HIGH PERFORMANCE COMPUTING FOR IRREGULAR ALGORITHMS AND APPLICATIONS WITH AN EMPHASIS ON BIG DATA ANALYTICS

A Thesis
Presented to
The Academic Faculty

by

Oded Green

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computational Science and Engineering

Georgia Institute of Technology
May 2014

HIGH PERFORMANCE COMPUTING FOR IRREGULAR ALGORITHMS AND APPLICATIONS WITH AN EMPHASIS ON BIG DATA ANALYTICS

Approved by:

Professor David A. Bader,
Committee Chair
School of Computational Science and Engineering
Georgia Institute of Technology

Professor David A. Bader, Advisor
School of Computational Science and Engineering & School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Richard Vuduc
School of Computational Science and Engineering
Georgia Institute of Technology

Professor Srinivas Aluru
School of Computational Science and Engineering
Georgia Institute of Technology

Professor Polo Chau
School of Computational Science and Engineering
Georgia Institute of Technology

Professor Bo Hong
School of Electrical and Computer Engineering
Georgia Institute of Technology

Yitzhak Birk
Electrical Engineering Department
Technion

Date Approved: 13 March 2014

To my parents, Itzhak and Esther, and my twin brother, Yoav,

you have always made me shoot to the stars.

To my wife, Ania,

you are my Northern Star and guide me through the worst of storms.

ACKNOWLEDGEMENTS

I am thankful for the support of my family throughout my graduate studies. Your support means the world to me and greatly simplified my life.

I want to thank my friends and fellow graduate students. You helped me walk this long and windy road. I am especially thankful to Rob McColl and David Ediger who helped me transition from Windows to Linux and had the patience to show me the basics - it is not easy to teach an old dog new tricks and you were successful.

To my friends, Lluis Miquel Munguia, Aakash Goel, Rajath Prasad, Adam McLaughlin, Xing Liu, Piyush Sao, Marat Dukhan, Aparna Chandramowlishwaran, Aftab Patel, and Zhaoming Yin, thanks for the many discussions (over lunch, coffer, or even a glass water). I learned a lot from you all.

To my advisor, Prof. David A. Bader, thanks for giving me the opportunity to study a new field. I am especially grateful to David for giving me as much leeway as I needed to start any new research problem.

I want to thank my committee: Professor Rich Vuduc, Professor Srinivas Aluru, Professor Polo Chau, and Professor Bo Hong. I had many interesting discussions with you all - some of the computer related and some not. I enjoyed them all.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

As a PhD student my research has focused on designing algorithms for Social Networks Analytics (SNA), primarily graph based analytics, and finding new approaches to applying these to massively-threaded systems. Throughout this endeavor, we have constantly been on the lookout on the impact of the algorithm and application on the architecture and vice-versa. Though our focus was not algorithm and architecture co-design, understanding the architecture has been useful in achieving better utilization and in fitting the algorithm to the architecture. This is fundamental for high performance computing.

While my work has focused on SNA, I had a chance to work on several different problems. It was interesting to see how different problems are in fact connected; and via several "paths" . For example, Merge Path [115], a parallel merge algorithm that I had started to develop prior to my arrival at Georgia Institute of Technology, was in fact a building block for clustering coefficients which requires sorted list intersection. Sorted list intersection is a similar operation to merging. As such, Merge Path [115] and its GPU counterpart, GPU Merge Path, [73] can be used to increase performance and load balance the computation of clustering coefficients. This insight, made me realize that is crucial for me to always be on the watch for the connections between different algorithms and data structures. On more than occasion I found that an optimization for algorithm A may also be applicable to algorithm B.

1.1 Thesis Content

This chapter is mostly a motivation for my work. I start off with a brief introduction to Big Data. This will be followed by a discussion of graph/network properties that played a pivotal role in understanding the dynamics of real world data. This will be followed by a discussion on the challenges of dealing with dynamic (aka streaming) data sets. I will then briefly introduce a new dynamic graph data structure/representation for dynamic graphs.

Chapters 2 and 3 discuss our contribution for the computation of Betweenness Centrality. Betweenness Centrality (BC) [67] is a widely used analytic used for finding key players in a social network. In Chapter 2, we present a new algorithm for computing BC for dynamic graphs that is an extension of Brandes's [30] static graph algorithm. Our dynamic algorithm is several hundred times faster that the static algorithm for edge insertion and deletion. This chapter discusses both exact and approximate computations of betweenness centrality with an additional emphasis on parallelization. At this time, our algorithm is the only dynamic betweenness centrality algorithm to support both approximation and parallelization. The work in Chapter 2, was in fact preceded by two of our publications [77] and [74]. In [77], we did not deal with either parallelization or approximation. This was partially due to the increased storage complexity of the dynamic approach. This required us to reevaluate the data structures used by Brandes's algorithm. Which led to the development of a new approach for computing betweenness centrality [74] - Chapter 3. This approach can be applied to multiple flavors of betweenness centrality and it increases scalability, reduces memory footprint, improves cache usage, increases problem size that can be handles, and can potentially offer better loading balancing (implementation dependent). Our algorithm also proved to be twice as fast (per core) then previously published algorithm for x86 systems. We implemented several BC algorithms on two shared-memory systems including a 40-core Intel x86 system and the Cray XMT2 [93]. Our new approach can be applied to additional betweenness centrality algorithms: parallel[15, 104], distributed [58], and approximate [17].

Chapters 4 and 5 present several novel optimizations for the efficient computation of Clustering Coefficients. Clustering coefficients is another widely analytic thats state how tightly bound vertices are based on the number of closed triangles that they belong [139]. In Chapter 4 we present a new approach to compute clustering coefficient using vertex covers. This approach avoids counting the same triangle multiple times and can be added to the well known lexicographical approach that reduces the times a triangle is counted by a factor of two. We showed that our algorithm can be extended to count larger circuits as well. From this optimization, we were able to better understand the load imbalance caused by straightforward parallelization of clustering coefficient that is due to power-law distribution of the edges. In Chapter 5 we present two unique load balancing techniques for clustering coefficients - these approaches trade off accuracy of the load-balance with storage requirements. Our new algorithms are several times faster than previous implementations. Further, our two optimizations can be combined, thus achieving even higher speedups.

Chapter 6 shows a new data structure and algorithm for tracking connected components in dynamic graphs. This algorithm takes into consideration the small-world property and shrinking diameter properties and shows that it is possible (and most likely beneficial) to track the connected components using $O(1)$ memory for each vertex. Our algorithm can easily keep up with the fastest update rates that current social networks produce on a shared-memory system.

Chapters 7 and 8 present a novel approach for parallel merging of sorted data. We dubbed this approach Merge Path. Chapter 7 discusses a visual and intuitive approach for parallel merging two sorted arrays. While the resulting partition and the computational complexity are similar to those of certain previous algorithms. The insights gained from the new approach have allowed the design of a synchronization-free, cache-efficient merging (and sorting) algorithm. The first of its kind. In Chapter 8 we extend the Merge Path concept to the GPU and create the first of a kind GPU merging algorithm. A brief summary of our merging results: $32X - 35X$ speedup on a 40 core system, first of its kind cache-aware merge, and a $50X$ speedup on NVIDIA's Fermi GPU over an Intel core. In fact, merging and adjacency list intersection (which is a key building of clustering coefficients) are similar operations. Thus, Merge Path can be extended to adjacency list intersection if needed.

Chapter 9 presents anew algorithm for computing the 2D estimated covariance matrix which avoids many redundant multiplications and additions without increasing memory requirements or communication costs. The sequential algorithm that is over $40X$ faster than the straightforward implementation. Our new algorithm is highly scalable due to low storage requirements and no communication costs.

1.2 Big Data Graph Analytics

Graph analytics and data structures have received a lot of attention for the last 50 years. Nonetheless, there are still many research opportunities in optimizing them. These optimizations include improving performance (per core), increasing parallel scalability, and dealing with dynamic and larger data sets. All these are important in this era of Big Data.

Big Data has been used to represent a wide range of topics (and such has become some what ubiquitous). Recently it was defined in [52] as the 3 V's: “"Big data is high Volume, high Velocity, and/or high Variety information assets that require new forms of processing to enable enhanced decision making, insight discovery and process optimization.” This definition is useful as it presents the key challenges associated with Big Data. A fourth V has also been added, Veracity.

- *Volume* - the size of the data is so massive that in most cases only single iteration over the data is possible or a single iteration over the incoming edges is possible. Further, the size of the data limits the amount of memory that an algorithm can use as the data utilizes a significant part of the memory. For larger data

- *Velocity* - the rapid updates to the data means that the recomputing an analytic is simply not feasible as by the time a recomputation has complete, it is in fact no longer relevant.

- *Variety* - each data set will have its unique properties such as the distribution of edges, rate of updates, size of the data set. Algorithms need to be designed taking these into consideration.

- *Veracity* - one considering a data set, it is necessary to ask is there an actual truth hidden within the data and if it can be extracted by an algorithm.

The above presented the generic challenges that Big Data analysts deal with daily when designing new metrics and analytics. This thesis focuses on the topic of social networks analysis for massive graphs with millions of vertices and billions of vertices, such as Facebook and Twitter. The size of these massive graphs present numerous algorithmic challenges, especially given that the networks are constantly updated by events such as new friendships and the joining of new members. In many cases , these events do not impact the entire network, but rather they impact a small local area of the network.

Static graphs algorithms such as connected-Components [129], clustering coefficients [139], single-pair-shortest-path [45, 66, 137], modularity [111], and betweenness centrality [67, 30] are used for graph analysis. Some of these algorithms require significant computational resources.

This has led to the development of algorithms for dynamic graphs, a.k.a. streaming graphs. My work has focused on the creation of these types of algorithms. I will present in this thesis several algorithms that I have developed for dynamic graphs.

The first is betweenness centrality and the second is connected components for streaming graphs. I will also briefly discuss an optimization made for computing dynamic clustering coefficients.

In addition to the dynamic algorithms, my work has also focused on improving performance of several analytics - both sequential and parallel implementations. These optimizations include modifying data structures, reducing computational overhead, and improving load-balancing.

1.3 Social Network Properties, Challenges, Insights, Opportunities

Before moving to the chapters in the thesis that present my algorithmic optimizations, I will present several real-world graph properties that were instrumental in designing the new algorithms. These properties present challenges; however, understanding these properties offered us insights and opportunities to overcome the hardship. Several of our optimizations in fact can be applied to more general type networks , such as communication and transportation networks, that only have a subset of the properties.

- Small World property - the first work to discuss the small-world property is due to Milgram [109]. Milgram suggests that people within the United States are not likely to be separated by more than six steps. This was later reconfirmed in [138] for additional networks.
  - *Challenge* - algorithms requiring three to four hops from a given a vertex will may require to access millions to billions of vertices. This is not computationally scalable. We do note that vertices one or two hops away is currently feasible.
  - *Insight* - if we consider a BFS-like traversal, it is likely that there will be many edges between vertices that are at an equal distance from the

root (edges between vertices in the same level). These edges do not have shortest paths going through them.

  - *Opportunity* - the insight greatly motivated our work on dynamic betweenness centrality as we questioned the change to the underlying graph due to an edge insertion or deletion . Our hope was that there would be a large number of edge updates in the same level of a BFS-like tree, which in turn requires little to zero updating of the data structures.

- Shrinking graph diameter - in [100] it is shown that as graphs evolve over time, their diameter decreases and the graphs become denser.

  - *Challenge* - as the diameter decreases, algorithms requiring vertices at three to four away become even more computationally demanding.
  - *Insight* - if we consider a BFS-like tree with a given, many of the new edges will connect vertices that are an equal distance to the root. As such, these edges will not have any shortest paths going through them.
  - *Opportunity* - for betweenness centrality, if a new edge does not create any new shortest paths. For dynamic connected components we developed a new data structure that benefits from the graph becoming denser.

- Power law distribution - in [22, 31, 63] it is shown that many networks follow a power law distribution for the number of adjacencies a vertex. This means that few vertices have many adjacencies and many vertices have a small number of adjacencies.

  - *Challenge* - parallel algorithm that partition the workload based on vertex list can potentially suffer from workload imbalance as a single vertex (or a set of vertices) with a a large adjacency set can cause a single thread to be the execution bottleneck.

  - *Insight* - by using a parallel prefix summation we can estimate the workload for a specific phase in the algorithm and offer better load-balancing. For massively parallel systems like the Cray XMT, the workload imbalance is not initially felt because of the large number of supported threads. As the thread count is increased for a specific data set, the workload imbalance is increased.
  - *Opportunity* - we recognized this workload imbalance for both BFS and Clustering Coefficients [138]. We were able to show that the parallel prefix summation and the additional partitioning is not computationally demanding. Further, we show that the additional overhead introduced is preferable over the naive partitioning that the causes the workload imbalance.

- One massive connected component- In Broder *et al.* [32] the authors show that World Wide Web (WWW) has one connected component that contains 90% of the vertices in the graph.
  - *Challenge* - a single update to the graph may require updating the entire connected component which is almost equivalent to doing a full static graph recomputation.
  - *Insight* - for most networks, it is unlikely that we will have deal with connecting or disconnecting two different connected components of size $O(V)$. Typically, a component of size $O(1)$ will be connected with a large component.
  - *Opportunity* - the reality is that many analytics only require doing "local" updates that are within a small proximity of the modification. For example, when connecting two components when one of them is the large component and the other is a small component, only the small component needs to be traversed.

- High velocity updates - social networks are constantly changing, whether it be new relationships between or new members joining the network, these updates can potentially arrive at such high frequencies that it is not possible to compute the analytic from scratch.
    - *Challenge* - updates come in at a faster pace than a full recompute can be handled.
    - *Insight* - maintaining state in between updates can help reduce touched unaffected parts of the graph.
    - *Opportunity* - create algorithms and data structures that can be updated at a fast pace. Further, do only the bare essential computations that offer new insights on the analytic. Avoid redundant computations.
- Sparsity - many real networks are considerably sparse with an average degree that is a constant $O(1)$ or logarithmic $O(log(V))$. The average degree (relationships) for Facebook [7] is 189 [136], which is very small compared to the number of people in the network.
    - *Challenge* - algorithms that skim over all possible vertex pairs are wasteful as the $|E| << |V|^2)$. Algorithms that are matrix-multiplication based are also costly.
    - *Insight* - do not add layers of abstraction to the graph. Use the edge list as best as possible for a specific analytic and its implementation.
    - *Opportunity* - clustering coefficients can be computed in several fashions. The time complexity of one approach is dependent on the vertex with the highest adjacency in the power of two. By adding a pre-algorithm computation phase, we were able to remove redundant computations between $15\% - 40\%$.

1.4 Dynamic Graph Challenges

The type of events that occur for dynamic graphs include edge insertion, vertex insertion, edge deletion, and vertex deletion. Algorithm that support both edge insertion (incremental) and edge deletion (decremental) are considered fully dynamic algorithms. If the algorithm supports one of these operations, it is partially dynamic.

Streaming graph algorithms present several challenges which include, 1) Correctness - the algorithm should output an exact state of the desired analytic. We note that for some analytics, approximate values are also acceptable. 2) Parallelism - as system resources are increased it is also desirable that the analytic utilize the new resources so that updates can be dealt with at the rate they come in. Synchronization and communications need to be minimized. 3) Time complexity - obviously if the time complexity of the dynamic graph algorithm is greater than the time complexity of the static graph algorithm, it is better to reuse the static graph algorithm. As such it is the complexity of the dynamic algorithm should not exceed of the static algorithm. 4) Batching - given the high rate in which events occur, these events are grouped into units known as batches. On the one hand, batches offers an opportunity to increase parallelism by dealing with them concurrently; on the other hand this requires finding the appropriate synchronization methods.

1.5 STINGER - A Dynamic Graph Data Structure

Effective implementation of a dynamic graph requires an efficient data structure for representing the graph. STINGER is such a data structure [18, 54]. STINGER is a compromise between massive storage and fast updates of an adjacency matrix and the minimal storage and static nature of CSR (Compressed Sparse Row) representation. Further, STINGER is designed for parallelism such that multiple threads can read and update the graph concurrently [55].

For dynamic graph implementations, updating the graph representation must be included as part of the execution time. For a CSR representation, this might require recomputing the prefix summation array and can thus become a dominant factor in the update time. In our implementation, the update is included in the execution. STINGER can support over 1M graph updates per second [55].

STINGER is free and open source software co-developed by our group. STINGER has been used to implement a variety of dynamic graph algorithms including clustering coefficients, community detection, connected components [107], and betweenness centrality [77].

1.6 My Publications

Refereed Journal Papers:

1. O. Green, and Y. Birk, "Scheduling Directives: Accelerating Shared-Memory Many-Core Processor Execution", to appear Parallel Computing

Refereed Conference Papers:

1. O. Green, L.M. Munguia, D. Bader, "Load Balanced Clustering Coefficients", ACM Workshop on Parallel Programming for Analytics Applications (PPAA), PPoPP, Orlando, Florida, 2014
2. R. McColl, O. Green, D. Bader, "A New Parallel Algorithm for Connected Components in Dynamic Graphs", IEEE International Conference on High Performance Computing, Hyderabad, India, 2013
3. O. Green, Y. Birk, "A Computationally Efficient Algorithm for the 2D Covariance Method", ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis, Denver, Colorado, 2013
4. O. Green, D. Bader, "Faster Clustering Coefficient Using Vertex Covers", 5th ASE/IEEE International Conference on Social Computing, Washington DC, 2013
5. O. Green, D. Bader, "Faster Betweenness Centrality Based on Data Structure Experimentation", 13th International Conference on Computational Science, 2013
6. O. Green, and Y. Birk, "Scheduling Directives for Shared-Memory Many-Core Processor Systems", International Workshop on Programming Models and Applications for Multicores and Manycores, PPoPP, Shenzhen, China, 2013

7. O. Green, R. McColl, D. Bader, “A Fast Algorithm for Incremental Betweenness Centrality”, ASE/IEEE 4th International Conference on Social Computing, Amsterdam, Holland, 2012

8. O. Green, R. McColl, D. Bader, “GPU Merge Path - A GPU Merging Algorithm”, ACM 26th International Conference on Supercomputing, Venice, Italy, 2012

9. S. Odeh, O. Green, Z. Mwassi, O. Shmueli, Y. Birk, “ Merge Path - Parallel Merging Made Simple”, Multithreaded Architectures and Applications (MTAAP) Workshop, IEEE 26th International Parallel & Distributed Processing Symposium (IPDPS), 2012

Non-refereed papers

1. “Detecting Insider Threats in a Real Corporate Database of Computer Usage Activity” , 19th ACM SIGKDD international conference on Knowledge Discovery and Data Mining, 2013

2. O. Green, L. David, A. Galperin, Y. Birk, “Efficient parallel computation of the estimated covariance matrix”, arXiv, 2013

3. L. David, A. Galperin, O. Green, Y. Birk, “Efficient parallel computation of the estimated covariance matrix”, IEEE 26th Convention of Electrical and Electronics Engineers in Israel (IEEEI), 2010

CHAPTER II

PARALLEL APPROXIMATE BETWEENNESS CENTRALITY FOR DYNAMIC GRAPHS

This chapter is an extension of the paper: O. Green, R. McColl, D. Bader, "A Fast Algorithm for Streaming Betweenness Centrality", ASE/IEEE 4th International Conference on Social Computing, Amsterdam, Holland, 2012.

The appearance of large networks (including social, communication, and transportation) has created a need for fast business intelligence analytics that can deal with the frequent updates that occur to the network without hindering real-time monitoring. Static graph algorithms permit capturing information for different snapshots of the network, though in practice this approach is undesirable as by the time the computation has completed the information is no longer relevant. In contrast, dynamic graph algorithms can provide significantly faster update times as they can benefit from "local" changes that require fewer modifications - allowing for faster update times. The above is especially true for the computationally demanding betweenness centrality analytic which is used to find key players in a network based on the number of shortest paths that these are on. In this work we present a novel algorithm for computing betweenness centrality for dynamic graphs. As our algorithm extends a widely used a static graph algorithm it can benefit from existing optimizations. We show how to apply approximation and parallelization to our algorithm that allows for further reduction of the execution time. Using multiple real-world networks, taken from the DIMACS 10 challenge, we show that our dynamic graph algorithm is several orders of magnitude faster than the static graph algorithm (up to 7990X faster). The output of the dynamic algorithm is equal to the output of the static graph algorithm.

2.1 Introduction

Betweenness centrality has been widely used for social network analysis since its introduction by Freeman [67]. Betweenness centrality finds key players in the network based on the All-Pairs Shortes-Paths (APSP). Several applications that have used betweenness centrality include: community detection [71], brain network analysis [119], and finding bottlenecks in communication networks [36].

With the introduction of massive social networks such as Facebook and Twitter that have over 100 million users and thousands of updates per second, betweenness centrality has required numerous algorithmic optimizations which include parallelization (shared-memory and distributed systems at different parallel granularities), approximation, reduction in storage complexity, and the creation of dynamic graph algorithms . All these are important as betweenness centrality is computationally demanding.

In this work we show how to compute betweenness centrality for dynamic graphs using parallel and approximation optimizations in an approach that extends Brandes's algorithm [30]. The similarity of the static graph and dynamic graph algorithms allows applying existing optimizations.

Our contributions are as follows:

- Design and implementation of a dynamic graph algorithm. Our dynamic graph algorithm maintains state in between iterations.
    - The dynamic graph algorithm is correct for both exact and approximate betweenness centrality.
    - We improve the storage complexity for the exact algorithm to $O(V^2)$ from its previous complexity, $O(V \cdot (V + E))$ [77]. The improved storage complexity is due to a performance optimization by [74].

  - Our algorithm supports both edge insertions and deletions - both will be discussed. Proofs in this chapter will be for the insertion; however, these proofs require little modifications to support deletions.

- We are the first to show a parallel approximate betweenness centrality dynamic graph algorithm. Extending the dynamic graph algorithm to support parallelization and approximation:

  - Our algorithm supports the different parallel granularities of previous implementations.
  - The storage complexity of of the approximate dynamic algorithm is $O(K \cdot V)$, where K are the number of roots that will be used in the approximation. For real world graphs $K << V$ - which allows analyzing large graphs.

- We show performance results for our coarse-grain parallel implementation of the static graph and dynamic graph algorithms:

  - We show that the coarse-grain parallel approach can be implemented on a shared memory many-core system - until recently this was not possible because of storage requirements. Our implementation achieves better scalability than previous fine-grain approaches is it does not require atomic instructions and has low communication costs.
  - We show that the dynamic graph algorithm is faster than using the static graph algorithm. The dynamic graph algorithm can be up to $7990X$ faster than the doing a recomputation using the static algorithm.
  - We show that the speedup of the dynamic graph algorithm is dependent on the number of traversed edges. This is a key issue for load balancing of this algorithm.

The remainder of the chapter is organized as follows. Section 2 presents the related works. Section 3 introduces our new algorithm and the data structures we maintain. We discuss both edge insertions and deletions. The formal proofs for the algorithms in this section can be found in Appendix A. In Section 4 we show performance results of our new algorithm. This includes a comparison of the dynamic graph algorithm with the static graph algorithm and parallel scaling results. Section 5 summarizes our contributions and presents additional avenues of research for future work.

2.2 Related Work

Using notations from [67], the number of shortest paths between two vertices s and t is denoted as $\sigma_{s,t}$ and the number of shortest paths between two vertices s and t that go through v is denoted as $\sigma_{s,t}(v)$. The betweenness centrality value of a vertex v is:

$$C_B(v) = \sum_{s \neq t \neq v} \frac{\sigma_{st}(v)}{\sigma_{st}}. \tag{1}$$

2.2.1 Brandes's Algorithm

Brandes [30] presents a fast algorithm for computing betweenness centrality based on a dependency accumulation technique which accesses the vertices in the reverse order of the BFS (Breadth First Search) traversal. The dependency accumulation essentially reduces the number of additions in the pair-wise summation as it avoids doing any additions between vertices that are not on the shortest paths. At this time, Brandes's algorithm has the best known time complexity for computing betweenness centrality of $O(V \cdot (V + E))$. The storage complexity of Brandes's algorithm is $O(V + E)$. In recent work it was improved to $O(V)$ [74].

Brandes's algorithm for computing betweenness consists of four stages. The first two stages, Stage 0 and Stage 1, are data structure initialization stages, where Stage 0 is a global initialization where the betweenness centrality value of each vertex is initialized to zero and is executed exactly once. The remaining stages will be executed once for each vertex. Stage 1 initializes the data structures that will be used in Stages 2 and 3.

Stage 2 is a BFS traversal from a given root. The BFS traversal in this algorithm includes several modification that are not found in a regular BFS [45] that simply finds the depth of the vertices: 1) each vertex maintains a list of the parents in the level above it [1] and 2) once all the neighbors of a vertex have been traversed, it is placed in a stack that is used in Stage 3. The stack maintains the reverse ordering of the queue.

Stage 3 computes the betweenness centrality values by using the dependency accumulation technique. Brandes defines the pair-dependency for a pair of vertices s, t is defined as follows:

$$\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}}. \tag{2}$$

In [30], Brandes proves the following recursive relationship (where $P_s(w)$ is the parent list for vertex w in the tree with s as its root):

$$\delta_s(v) = \sum_{\{w|v \in P_s(w)\}} \frac{\sigma_{sv}}{\sigma_{sw}}(1 + \delta_s(w)). \tag{3}$$

The immediate outcome of this is that it is no longer necessary to sum all the pair-dependencies as they follow a recursive relation. In addition to this, it is possible to compute each of the $\delta_s(w)$, by computing the shortest path from the root, s, to the rest of the graph using a single source shortest path algorithm.

2.2.1.1 Complexity Analysis

The time complexity of the BFS is $O(V+E)$. The time complexity of the dependency accumulation is $O(V+E)$ - as it is bound by number of traversed parents $O(E)$ and vertices $O(V)$. For the exact algorithm, the BFS and dependency accumulation is

[1]In [104], the parent list is swapped for a list of children. This is useful for parallel implementations. Further, the children list approach is preferable for directed graphs

Algorithm 1: Brandes's algorithm for computing betweenness centrality [30]. The algorithm consists of four stages.

```
Stage 0 - global initialization
$C_B[r] \leftarrow 0,\ r \in V$ ;
for $r \in V$ do
    Stage 1 - local initialization
    $S \leftarrow$ empty stack; $Q \leftarrow$ empty queue;
    $P[w] \leftarrow$ empty list, $w \in V$;
    $\sigma[t] \leftarrow 0,\ t \in V$; $\sigma[r] \leftarrow 1$;
    $d[t] \leftarrow \infty,\ t \in V$; $d[r] \leftarrow 0$;
    enqueue $r \rightarrow Q$;
    Stage 2 - BFS traversal
    while $Q$ not empty do
        dequeue $v \leftarrow Q$;
        push $v \rightarrow S$;
        for all neighbor $w$ of $v$ do
            // $w$ found for the first time
            if $d[w] = \infty$ then
                enqueue $w \rightarrow Q$;
                $d[w] \leftarrow d[v] + 1$;
            if $d[w] = d[v] + 1$ then
                $\sigma[w] \leftarrow \sigma[w] + \sigma[v]$;
                append $v \rightarrow P[w]$;
    Stage 3 - dependency accumulation
    $\delta[v] \leftarrow 0,\ v \in V$;
    while $S$ not empty do
        pop $w \leftarrow S$;
        for all $v \in P[w]$ do
            $\delta[v] \leftarrow \delta[v] + \frac{\sigma[v]}{\sigma[w]}(1 + \delta[w])$;
        if $w \neq r$ then
            $C_B[w] \leftarrow C_B[w] + \delta[w]$;
```

computed for each vertex. As such, the total time complexity is $O(V^2 + V \cdot E)$.

The storage requirements for computing betweenness centrality is the sum of the storage requirements of the stack, queue, arrays, and the parent lists. All the data structures except for the parent lists require $O(V)$ memory. The parent list requires $O(E)$ memory as the maximum number of parents a vertex is bound by the number of edges it has. As such Brandes's algorithm has a storage complexity of $O(V + E)$.

In [74], the authors show an additional optimization that shows how to avoid maintaining the parent list, reducing the memory complexity to $O(V)$. This new approach is called the neighbor-traversal approach.

2.2.2 Parallel Algorithms and Implementations

The performance and storage requirements of parallel algorithms are highly implementation dependent. In [135] a taxonomy of parallel betweenness centrality is presented. This taxonomy divides betweenness centrality implementations based on their parallel granularity: 1) coarse-grained, 2) medium-grained, and 3) fine-grained. In fact the storage complexity of the algorithm is dependent on the parallel granularity. Table 1 presents the storage complexity for multiple betweenness centrality algorithms.

Until recently, the storage requirements of the coarse-grain approach, $P \cdot (V + E)$ where P is the number cores used, limited its scalability which made the fine-grain and medium-grain approaches preferable. With the reduction, in storage requirements of the coarse-grain approach is also feasible for large thread counts. Deciding on the actual parallel implementation is system dependent yet there is a clear trade-off between storage vs. synchronization (atomic instructions and locks) requirements.

Bader & Madduri [15] showed the first parallel implementation for computing betweenness centrality. In Madduri *et al.* [104] the parent list is swapped for a children list which removes the need for locks in the dependency accumulation stage, though atomic instructions are still used in their newer algorithm. Both of these are fine-grain parallel algorithms and are implemented on massively parallel systems. A medium-grain parallel algorithm can be found in [135].

Parallel betweenness centrality is also part of the following software packages: SNAP [2, 16], GraphCT [1, 57], LIGRA [130], and GALOIS [116]. In [134] optimizations computing betweenness centrality are shown for the IBM Cyclops64. GraphCT is a software package for the Cray XMT system.

2.2.2.1 Neighbor-Traversal

In [74], a new approach for computing betweenness centrality is presented. This approach suggests removing the parent list data structure and traversing all the neighbors of a vertex while looking for neighbors in the level above (i.e. parents)[2]. The observations made in that paper is that the parent list is the only data structure used in the betweenness centrality that requires $O(E)$ storage and can limit the parallel scalability of the algorithm. From a performance perspective, the authors show that despite doing the additional traversals, the execution time on a single core is reduced. This phenomena is explained by the parents lists dominating the cache and kicking out the remaining data structure from the cache. The reader is referred to [74] for more details.

This new approach makes the coarse-grain approach practical. This chapter also presents performance results for the coarse-grain approach and shows its scalability on several shared-memory many-core systems. Further, this approach be applied to additional parallel algorithms, including the fine-grain and medium-grain approaches.

Table 1 presents the storage requirements of multiple flavors of betweenness centrality algorithm with the parent lists and using the neighbor-traversal approach. The storage requirements are dependent on the type of parallelism, approximation, and the selection of static or dynamic computation.

2.2.3 Approximate Algorithms

The high computational requirements of betweenness centrality make analyzing large networks with millions to billions of vertices near impossible on actual systems. Approximation techniques have been created that reduce the computational requirements (making the analysis feasible) at the cost of accuracy. In[17] it is suggested that

[2]In [74] a discussion on directed/undirected can be found. There is also a discussion on the difference between the parent list and the children list for this approach.

Table 1: Memory bounds for different betweenness centrality algorithms. P denotes the number of cores used by each algorithm. K denotes the number of roots used in the the approximation. The dynamic algorithms do not depend on the number of cores as these data structures are maintained as part of the computation.

Algorithm	Graph	Previous	Neighbor-traversal [74]
Exact [30]	Static	$O(V+E)$	$O(V)$
Approximate [17]	Static	$O(V+E)$	$O(V)$
Parallel fine-grain [15, 104]	Static	$O(V+E)$	$O(V)$
Parallel coarse-grain [74]	Static	$O(P \cdot (V+E))$	$O(P \cdot V)$
Exact [77]	Dynamic	$O(V \cdot (V+E))$	$O(V^2)$
Approximate	Dynamic	$O(K \cdot (V+E))$	$O(K \cdot V)$
Parallel fine-grain approximate	Dynamic	$O(K \cdot (V+E))$	$O(K \cdot V)$
Parallel coarse-grain approximate	Dynamic	$O(K \cdot (V+E))$	$O(K \cdot V)$

a subset of roots, K, be selected. These K vertices will be the roots in Brandes's[30] algorithm. The time complexity for the approximation is $O(K \cdot (V+E))$. The approximation can be combined with the parallel algorithms [15, 104, 135, 74]. The HPCS SSCA [4] specifications recommends using $K = 256$ roots.

In [70], multiple approximation methods are discussed that reduce the bias towards the vertices closer to the roots. These methods use different weighting function to reduce the bias. This approach is different than [17] in the way it chooses its roots. The algorithm in [70] is iterative with one root betweenness centrality computed every iteration. In each iteration a root is randomly selected.

2.2.4 Dynamic Graph Algorithms for Betweenness Centrality

In recent time, three different dynamic graph algorithms were designed and implemented for betweenness centrality: Green *et al.* [77], Lee *et al.* [97], and Kas *et al.* [89]. These algorithms take very different approaches for updating the analytic, yet all three are focused on computing the exact values of betweenness centrality. None of these techniques discuss parallelization or approximation which is an increasing concern in an era where networks can potentially have millions (or billions) of vertices. Green *et al.* [77] is extended in this paper to support both approximation and parallelization; as such we do not extend the discussion on this algorithm in this subsection.

The algorithm in [97] reduces the amount of work required to update the graph by figuring out which vertices are not affected by the update. It then only update the vertices that will have a change in the centrality values. This is done by decomposing the graph into bi-connected components and finding which bi-connected components are affected by the update. It seems that this approach is mostly successful (performance-wise) with very sparse networks as in denser networks a single update is likely to affect the betweenness centrality value of multiple vertices. For real-world graphs with an increased density , the update is likely to touch a significant part of the graph, thus lower speedups are attained.

The algorithm of [89] maintains the all-pairs shortest-path as the updates occur. They too maintain state in between iterations to reduce the computational overhead - their storage complexity is $O(V^2)$. They use a different all-pairs shortest-paths algorithm than the one suggested by Brandes yet they use parent list in their implementation to traverse the necessary vertices. Their implementation is JAVA based using open-source software package GraphStream[3] and they compare their algorithm to Brandes's algorithm. At publication time, the GraphStream implementation of Brandes's algorithm uses the parent list approach with linked lists - meaning every time a new parent found is found it does a dynamic memory allocation which is a time consuming operation. The optimization from [74] would improve the performance of the static graph and dynamic implementation as it would remove the overhead of dynamic memory allocation. An extended discussion of the cost of memory allocation in betweenness centrality can be found in [74] which compared several different implementations of betweenness centrality with and without memory allocation - these were done in C and not in JAVA.

While some algorithms consider edge updates there are algorithms that consider vertex updates. A vertex update (in which multiple edges are inserted/deleted) can in

[3]http://graphstream-project.org/

fact be treated as a serialization of insertions/deletions. The key benefit of supporting these updates is that they can potentially increase the speedup as certain traversals might be avoided multiple times. From a social network perspective, such as in the case of Facebook, when a vertex joins the network it joins without friends and friends are added one at a time. Vertex deletions might be more relevant in the real world as person might die.

2.2.5 Additional Results

BFS is an important building block for many algorithms, which is another reason that it is a core kernel for the Graph 500 benchmark [110]. Beamer *et al.* [26] showed a novel and efficient approach for doing a BFS traversal that potentially reduces the number of traversed edges by changing the “direction” of the traversal such that unfound children check if they have any potential parents. The unfound children are found by doing a linear search through the distance array, size $O(V)$, for a specific level/depth in the tree. The benefit of this approach is that once a parent is found, it is no longer necessary to traverse its remaining edges, hence a reduction in the number of traversed edges. While the number of operations is potentially increased, using this approach potentially offers a $10X$ speedup over the standard BFS because of a reduced number of edge traversals. This algorithm is useful for finding connecting components. Unfortunately, this optimization can not be applied to betweenness centrality. The reason being, betweenness centrality requires finding all the paths to the root and not “a” path to the root.

Several distributed algorithms for betweenness centrality have been created. In [34] a distributed software package is introduced that allows one to implement graph algorithms using algebraic operators. In [59] a space efficient distributed algorithm is given.

In [120] a GPU implementation of static graph betweenness centrality is presented that duplicates vertices, which they refer to as virtual vertices, in order to load balance the work better amongst the GPU's many threads. The virtual vertices slightly increase the storage requirements, but, this is not a significant increase. This algorithm would also benefit from the neighbor-traversal optimization of [74]. In [121] an algorithm is presented for updating closeness centrality in dynamic graphs.

In [117] several heuristics are shown that allows contracting vertices together if they are "structurally equivalent", meaning that they play the same role in the network and are on similar shortest paths.

2.2.6 Real World Graph Properties

In recent studies, it has been shown that many real world networks share certain properties. These include small world diameter [138, 13] which is originally due to Milgram [109], power law distribution of the vertex degrees [63], and a single massive connected component[31] in the graph.

These properties present both a challenge and opportunity in the algorithm design. The power law distribution can in fact cause workload imbalance and reduce performance for the fine-grain and medium-grain parallel algorithm (the is not an issue for the coarse-grain). The small world might imply that newly inserted edges will not create many new shortest paths and thus will not require updating the analytic.

2.3 Dynamic Graph Betweenness Centrality

In this section we discuss our algorithm for computing betweenness centrality for dynamic graphs. We then extend our algorithm to support approximation and parallelization. We start off by explaining the dynamic exact betweenness centralit algorithm from Green *et. al* [77]. This will include a spatial and time complexity analysis. We then show that approximation can be applied to this approach allowing for our dynamic graph approximate algorithm to give the same results as the static graph approximate algorithm. Given the vast literature on parallel algorithm and the fact that our algorithm is based off of Brandes's algorithm, we will only briefly discuss the parallelization. We note ahead of time that our algorithm can in fact be parallelized in multiple fashions: coarse-grain, medium-grain, and fine grain.

Both insertions and deletions of edges are discussed in this chapter. We focus on insertions; however, the similarity between the insertion and deletions is not significant and does not require significant updating of the pseudo code or proofs. We will make note of the difference throughout the chapter. Prior to the algorithm discussion, we present the data structures that are needed by the algorithm. A more formal discussion with proofs can be found in the Appendix.

2.3.1 BFS Tree Data Structure and Other Structures

The key difference between the static graph algorithm and our dynamic graph algorithm is that our algorithm maintains a "state" between the updates. This state allows us to modify only the relevant vertices that are affected by the update. For most cases, the update only affects a small percentage of the vertices in the graph which means that a full static graph recomputation is wasteful as many values will remain the same . The difference between the static and dynamic approaches requires a different initialization stage. For simplicity, our proofs will focus on undirected unweighted graphs; however, they can be augmented for remaining graph types.

For each root a BFS-like tree is maintained (for the exact case there will be V BFS-like trees for the approximate there will be K BFS-like trees). The fields used by the BFS-like tree can be found in the upper half of Table 2. This BFS-like tree maintains the distance of the vertices that the root is connected to, the number of shortest paths the root has to these vertices, and the dependency accumulation value. Each of these values is maintained in array of size $O(V)$. Thus, the storage requirement of each BFS-like tree is $O(V)$. As K such roots are maintained as part of the approximation

Table 2: Notations used in the chapter.

Field Name	Description
$\sigma[v]$	Number of shortest paths from vertex v to the root.
$d[v]$	Specifies the distance of vertex v to the root. Initially the distances of all vertices from the root are set to ∞.
$\delta[v]$	Dependency accumulation value of vertex v.
$P[v]$	Parent list of vertex v. The optimization of [74] removes the need to store this list. It is still used conceptually.
$\hat{\sigma}[v]$	New shortest paths vertex v has following update. Only vertices that have had a change in the number of shortest paths will use this field.
$\hat{\delta}[v]$	Dependency accumulation value of vertex v that are part of the update. Even vertices that do not have shorter paths to the root can have this value updated because of its recursive nature.
$dP[v]$	The changes in the number of shortest paths to the root for vertex v following the update. This value is propagated downwards as part of the BFS traversal starting at the lower vertex.
$t[v]$	Denotes if vertex v has been found as part of the update process. Can have one of the following values (which specify at what point the vertex was found): $Not-Touched$, $Down$, and Up.
T_s	BFS-like tree with s as its root.

process, the storage requirement of our new method is $O(K \cdot V)$[4]. For exact dynamic betweenness centrality, where $K = V$, the storage requirement is $O(V^2)$ which might be too costly on some systems. However, when $K << V$ as in the HPCS SSCA2 [4] standard, the storage requirements of our approximate algorithm are affordable.

2.3.2 Insertion and Deletion Scenarios

There are five insertion scenarios and five deletion scenarios. Assume that the edge $e = (u, v)$ is the new or deleted edge.

2.3.2.1 The Insertion Scenarios

1. Same level insertion - the new edge connects vertices in the same level BFS-like tree, as depicted in Fig. 1.

2. Adjacent level - the new edge connects vertices in two adjacent levels in the BFS-like tree, as depicted in Fig. 2.

3. Non Adjacent levels aka "Pull-up" - the new edge connects vertices that prior to the insertion are separated by at least two levels, as depicted in Fig. 3. As the vertices are in the same connected component, a path exists between the vertices prior to the insertion. After the insertion, the vertices are in adjacent

[4]If the parent lists are maintained then the storage complexity goes up to $O(K \cdot (V + E))$. This has been shown to be costly and ineffective in scaling in [74].

levels.

4. Connecting two disconnected components - prior to the insertion the vertices do not have a path connecting them as they belong to different connected components, as depicted in Fig. 4. After the insertion, the connected components are joined into a single connected component.

5. Edge insertion in a different connect component - the new edges connects two vertices that are in a different connected component than the root, as depicted in Fig. 5.

2.3.2.2 The Deletion Scenarios

1. Same level deletion - the deleted edge connects vertices in the same level BFS, as depicted in Fig. 1.

2. Adjacent level with extra parent/s - the deleted edge connects vertices in two adjacent levels in the BFS-like tree, as depicted in Fig. 2. The lower vertex has at least one additional parent in the level above it after the deletion.

3. Adjacent level without extra parent - prior to the deletion the vertices are connected by a path of length one and the lower vertex does not have any additional parents in the level above it, as depicted in Fig. 3. The edges have an additional path connecting them, as such, they are still in the same connected component. However, the lower vertex moves down in the BFS-like tree and is further away from the root.

4. Disconnecting a single connected component - similar to the previous deletion scenario, the lower vertex has a single parent in the level above, as depicted in Fig. 4. Unlike the previous scenario, there are no additional paths connecting these two vertices. Therefore, the vertices now belong to two different connected components.

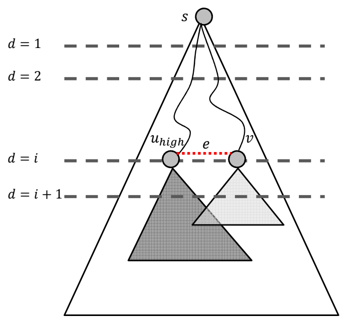


Figure 1: The insertion of edge $e = (u, v)$ connects two vertices that are on the same level in the BFS-like tree of root s. This insertion does not create any new shortest paths. In the case of edge deletion, no shortest paths are removed.

5. Edege deletion in a different connected component - the deleted edge is between vertices in a different connected component, as depicted in Fig. 5.

It is not surprising that the insertion and deletions complement each other. Note that a similar update taxonomy can be found in Shiloach and Even [127].

2.3.3 Same Level

For both the insertion and the deletion, other than updating the graph representation no additional computations are required. There are no new shortest paths from the root to any vertex that goes through that edge, Fig. 1. Intuitively, consider some path that does go through the new edge (in the insertion case) on its way to the root. A simple modification to that path allows creating an alternative shorter path that does not go through this edge (both vertices in the same level have different paths to the root). As such, inserting and removing these edges does not require updating any betweenness centrality values. Since these updates have computational complexity of $O(1)$, they are highly desirable.

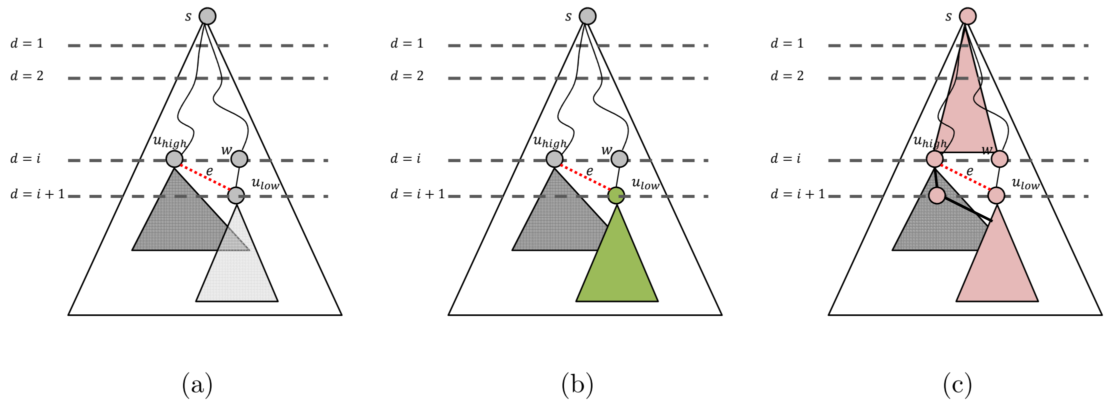


Figure 2: (a) The insertion of edge $e = (u_{high}, u_{low})$ connects vertices that are in adjacent levels before the insertion. (b) A BFS traversal is started at u_{low} as the shortest paths above this vertex are not affected by the insertion. (c) Shows the vertices that are affected by the dependency accumulation - this includes vertices that were not found in the BFS traversal beginning at u_{low}. The difference between the insertion and the deletion, is that the vertices found in the BFS traversal (green triangle) will have fewer paths to the root instead of additional paths to the root.

2.3.4 Adjacent Level Insertion and Adjacent Level Deletion with Extra Parent

In this scenario, prior to the update the vertices are one level apart. This is also the distance after the update. One vertex will be considered the higher vertex and one will be the lower vertex. The higher vertex is closer to the root than the lower vertex. For both the insertion and the deletion, the "structure" of the BFS-like tree does not change following the update as none of the vertices move in the tree (all distances to the root are the same as they were before the update), Fig. 2.

The number of shortest paths to the lower vertex has from the root changes (increases for insertions and decreases for deletions). Note, that none of the vertices above the lower vertex have any new shortest paths to the roots. This means that $\sigma[v]$ is not updated for any of the vertices above the lower vertex.

The immediate result of this observation is that it is not necessary to start the BFS traversal at the root as no new paths will be found until the lower vertex is reached. As such, it is possible to start the BFS traversal at the lower vertex, Fig. 2 (b). As the traversal starts at the lower vertex, only vertices that have a path to the root that goes through the lower vertex will be found in this BFS traversal. This means that only a fraction of the vertices and edges of the graph are traversed, which reduces the actual computational requirements. While a BFS traversal from the root is possible, it will discover the same number of shortest paths and distances that the BFS-like tree maintains. As such, there is no benefit in doing the full traversal.

In the dependency accumulation, once again only a fraction of the vertices are traversed. While the BFS traversal stage only required traversing vertices starting at the lower vertex, the dependency accumulation requires updating any vertex that has had one of its children update. Therefore, any vertex that has had a change in the number of paths to the root, will require updating all the vertices in the levels that are its way to the root, Fig. 2 (c).

The key difference between in the pseudo code for the insertion and the deletion is the change in the number of paths to the roots for the vertices found in the traversal starting at the lower vertex. For the insertion, there is an increase in the number of shortest paths. For the deletion, there is a decrease in the number of shortest paths.

2.3.5 Non Adjacent Level Insertion and Adjacent Level Deletion without Extra Parent

In this scenario, prior to the insertion the vertices are at least two levels apart. The insertion will pull-up the lower vertex to one level below the higher vertex and may cause additional vertices to be pulled up (a transition from Fig. 3 (a) to Fig. 3 (b)).

In case of a deletion, the vertices are one level apart and the lower vertex does not have additional vertices in the level above. This will cause the lower vertex to drop down in the tree, which in return can cause additional vertices to move down in the tree (a transition from Fig. 3 (b) to Fig. 3 (a)).

Following the insertion, all of the paths that the lower vertex has to the root must

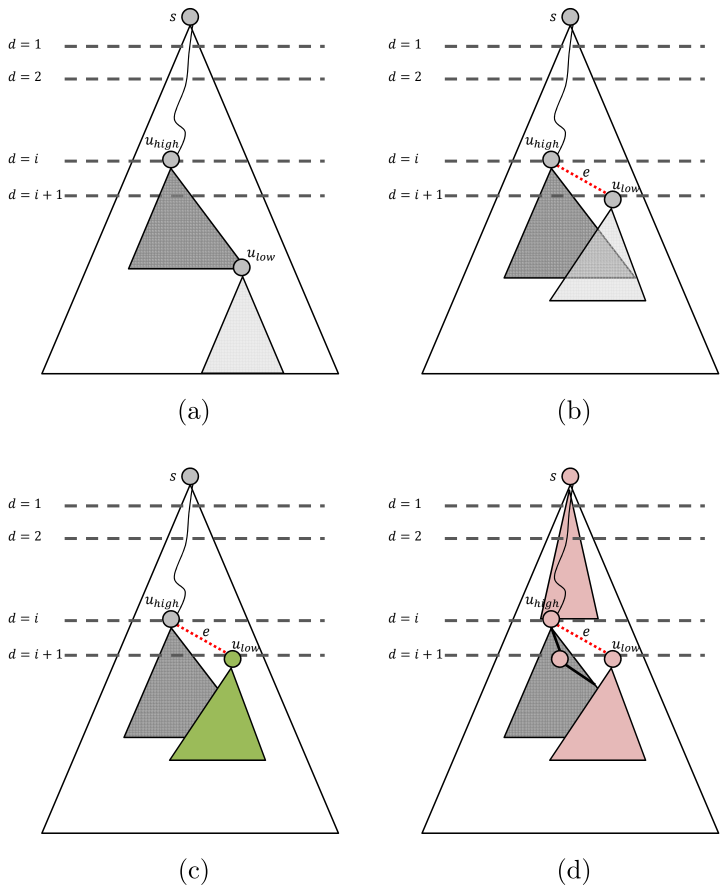


Figure 3: (a) The BFS-like tree prior to the insertion of edge $e = (u_{high}, u_{low})$. (b) The BFS-like tree after the insertion. Note that u_{low} has a single parent in the level above it. For deletions, swap between (a) and (b). (c) The BFS traversal starts at u_{low} and can possibly "pull-up" vertices that are closer to the root following the insertion. (d) The dependency accumulation can find vertices that were not in the BFS traversal.

go through the higher vertex as it is the only parent it has. Once again, all the vertices above the lower vertex do not have new paths to the root, which permits starting the BFS traversal at the lower vertex, Fig. 3 (c). Unlike the previous scenario where all the vertices stayed the same distance from the root, in this scenario multiple vertices can move closer to the root. For any vertex that moves, its paths to the root will be through its new parents in the level above it. Some vertices may stay the same distance from the root, yet, they may lose children from the level below, gain new parents, or get new neighbors. These may require updating the number of paths to

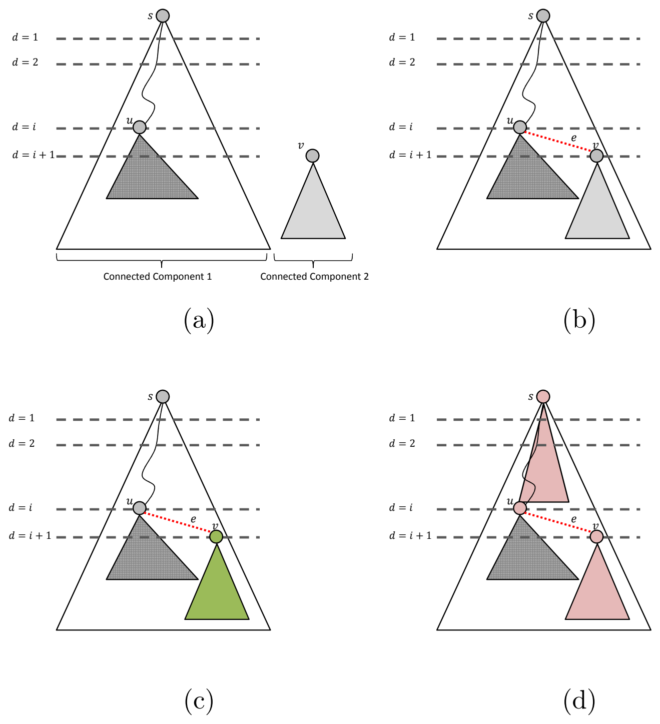


Figure 4: (a) Prior to the insertion, the vertices in different connected components. (b) After the insertions, the vertices are in the same connected component. All the paths between vertices that were previously in different connected components go through the inserted edge. For deletions, (b) is before the deletion and (a) is after the deletion. (c) and (d) are for insertions. (c) The BFS traversal starts at the vertex that just got connected to the first connected component. Only vertices in the second component will be found in the BFS traversal. (d) The dependency accumulation will go through all the vertices of the second connected component and work its way back through the newly connect edge and all the way up to the root.

the root and the dependency accumulation value. Thus, even if a vertex stays in its place, its dependency accumulation value requires updating, which entails updating all the vertices up to the root in a recursive fashion, Fig. 3 (d).

Again only a fraction of the vertices and edges will be traversed in both the BFS traversal and dependency accumulation in comparison with the static recomputation.

2.3.6 Connecting Components Insertion and Disconnecting Components Deletion

Fig. 4 depicts the scenario that the inserted edge connects two different connected components. Fig. 4 (a) depicts the BFS-like tree before the insertion and Fig. 4 (b) depicts the BFS-like tree after the insertion. Note that all the path between vertices that were previously in different connected components must go through the newly inserted edge. In the case of a deletion, swap the roles of Fig. 4 (a) and Fig. 4 (b) and note that all the shortest paths between the vertices in the new components went through the deleted edge.

As it turns out, this scenario is a sub-case of the Non-Adjacent level insertion, where the sub-tree that is "pulled-up" simply does not overlap with the remaining tree. The previous distance of the vertices from the second connected component was ∞ and now that distance is finite. For an unweighted graph, the maximal distance between two vertices is $V - 1$, which allows using an number greater than this to signify infinite distance. We do not extend the discussion as this is not a crucial issue.

2.3.7 Insertion and Deletion in Different Connect Component

The last insertion/deletion case is similar to the first, Same-level. The new edge is inserted into a different connected component - in fact, within that component one of the four previous cases can be applied. However, the current root does not have a path to any of the vertices as they are in different components, which means that are no paths have been updated and no computations are needed. An additional reason that this scenario is similar to the Same-level scenario is that the distance of both vertices from the root is the same: ∞. Using connected component algorithms [127, 128, 107] can also help detect this scenario.

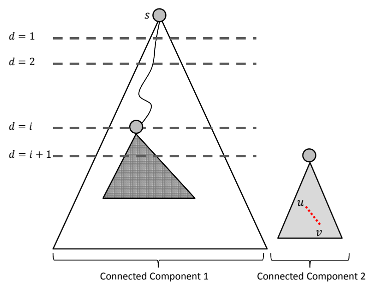


Figure 5: Edge insertion or deletion occurs in a different connected component than that of the root.

2.3.8 Additional Algorithmic Considerations and Optimizations

In the previous subsections we showed that an edge insertion and deletion falls into a small subset of scenarios for any given root. This in fact allows us to show that the output of dynamic graph algorithm and static graph algorithm are equivalent. This in turn means that we can use the static graph approximation approach of [17], which selects a subset of roots out of $K \subseteq V$, for the dynamic graph approach. If our new algorithm gives the correct output for all roots in the exact computation it will give the correct output for a subset of roots as well.

As our algorithm is an extension of Brandes's [30], it can benefit from the existing parallel approaches and optimizations. These approaches have different implementation concerns, pros, and cons. We refer the reader back to Section 2.2.2 for this discussion. In our presentation of both Brandes's algorithm and our own algorithm, we did not change the parent update approach to the children update approach [104] of the dependency accumulation which is important for parallel algorithm as it reduces the need for locks in the dependency accumulation stage. Further, for directed graphs this change is important and necessary for using the neighbor-traversal approach - parents know who the children are but the children don't necessarily know who their parents are because of the directed edges. Additional discussion on the benefits of the children approach can also be found in [74].

In the above, we considered unweighted graphs for our dynamic graph algorithm. Our algorithm can in fact be extended to support weighted graphs as well with a slight modification to the insertion/deletion scenarios. One example might be, instead of an adjacent level insertion, an edge insertion creates an additional path of equal weight to the root. For simplicity we have focused on unweighted graphs and do not extend the discussion on weighted graphs.

2.3.9 Complexity Analysis

For two of the scenarios, Same-level and different connected components, no work is required to update the data structure. The only thing that is required is to check the level of the vertices in a specific tree. This is an $O(1)$ operation.

All the insertion and deletion scenarios that require updating the data structure consist of the same two phases of Brandses algorithm: BFS traversal and dependency accumulation. These have a time complexity of $O(V + E)$, which is the same time complexity for a given root in the static graph algorithm.

For the exact computation, where there are V roots, the computational complexity is bounded by $O(V^2 + VE)$ similar to the one given by Brandes. For the approximate case, with K roots, the computational complexity is bounded by $O(K \cdot (V + E))$. In practice, the number of operations is smaller as there will be same-level insertions and fewer edges and vertices will be traversed for the cases requiring updates.

2.4 Results

For testing purposes we use the graphs from the 10th DIMACS Implementation Challenge [6]. The DIMACS 10 data set includes different types of networks: clustering, collaboration networks, road networks, random networks, and more. The subset of graphs that we used can be found in Table 3 and Table 4. The graphs in Table 4 are larger networks and will be discussed in a subsection 2.4.4. All the graphs are static.

For the graphs from Table 3, after the graph is read from the disk, 1000 edges are

Table 3: Graphs from the 10th DIMACS Implementation Challenge used in our scaling experiments. The execution times are for a single thread.

Name	Graph Type	$\|V\|$	$\|E\|$	Static (sec.)	Dynamic (sec.)
audikw1	Matrix	943,695	38,354,076	710	10.3
ecology1	Matrix	1,000,000	1,998,000	194	4.10
as-22july06	Clustering	22,963	48,436	1.22	0.0194
astro-ph	Clustering	16,706	121,251	1.52	0.0103
cond-mat-2005	Clustering	40,421	175,691	3.872	0.0163
hep-th	Clustering	8361	15751	0.292	0.00342
preferentialAttachment	Clustering	100,000	499,985	24.05	0.0298
smallworld	Clustering	100,000	499,998	24.81	0.0531
citationCiteseer	Collaboration Networks	268,495	1,156,647	75.531727	0.117694
coAuthorsDBLP	Collaboration Networks	299,067	977,676	68.006336	0.0558
coPapersDBLP	Collaboration Networks	540,486	15,245,729	367.486846	0.236555

randomly selected and removed from the graph. These 1000 edges are then inserted one at a time into the graph, updating the betweenness centrality values using our new parallel dynamic approximate approach. Unless mentioned otherwise, we use the HPCS SSCA2 [4] standard of $K = 256$ roots. The roots are randomly selected using a uniform distribution.

Our tests were conducted on a system consisting of 4 Intel $E7-8870$ processors. Each processors consists of 10 cores clocking at $2.4GHz$. Each core has $32KB$ L1 cache and $256KB$ L2 cache. All cores on the same processor share a $30MB$ L3 cache. The system has a total of 256GB of DRAM. While these processors support Hyperthread-ing, we did not use this feature and tested our algorithm on a maximal of 40 cores. We use cores and threads interchangeably.

Our implementation of both the dynamic graph and static graph algorithm uses the coarse grain approach which divides the roots of the approximation equally amongst the cores. Our implementation is OpenMP based and uses the OpenMP dynamic scheduler. Both implementations use the STINGER[18] dynamic graph representation. Changing the CSR implementation to support STINGER, simply required replacing the "for-loops" with several STINGER built in macros for adjacency list traversal.

For the static graph algorithm there was little difference between the static scheduler in which the roots are near-equally divided to the cores and the dynamic scheduler in which each core receives a different root when it has completed processing the previous root. As many graphs have a single large connected component, it is likely that most cores will receive an equal number of roots ensuring a near equal number of traversed edges which is key component in the load-balancing of the coarse-grain implementation.

For the dynamic graph algorithm, there was a difference in the performance. As we will show in this section, the dynamic graph algorithm is more sensitive to workload imbalance as each root will have a different number of edges to traverse. As a heuristic, before doing the $parallel - for$ loop on the roots, we place all the roots that have non-adjacent level insertion as the starting roots and these are followed by the adjacent level insertions. While an adjacent level insertion can have more work than a non-adjacent level insertion, in most cases this is not true - the non-adjacent level insertion typically has more work. By putting, the heavier tasks first, it is also less likely that there will be an execution tail where on a small number of cores are being utilized. The reality is that some of insertions are so time consuming that some cores will process a single root while others will process a larger number of less demanding roots. This will in fact motivate doing fine-grain parallel betweenness centrality for dynamic graphs.

2.4.1 Speedup Analysis of Dynamic Over Static

Fig. 6 shows the speedup of the dynamic graph algorithm over the static graph algorithm using a single core for the graphs in Table 3. Each insertion is timed. We present six different speedup bars for each network (from the left to the right): 1) the average speedup, 2) the minimal speedup, 3) the 25th percentile speedup, 4) the median speedup, 5) the 75th percentile speedup, and 6) the maximal speedup. Note

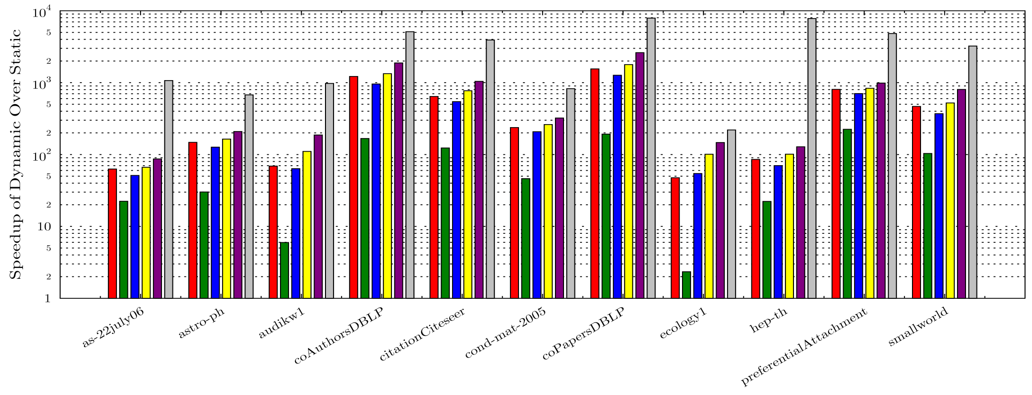


Figure 6: Six different speedup bar for each graph (from the left to the right) for single thread execution: 1) average, 2) minimal, 3) the 25th percentile, 4) median, 5) the 75th percentile, and 6) maximal.

the ordinate is log-scale.

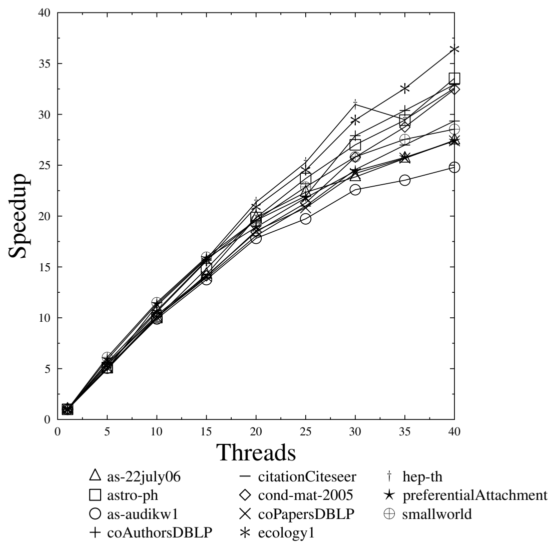
The speedup of the dynamic graph algorithm over static graph algorithm is computed using the following formula, where P denotes the number of cores:

$$Speedup_P = \frac{StaticTime_P}{DynamicTime_P} \tag{4}$$

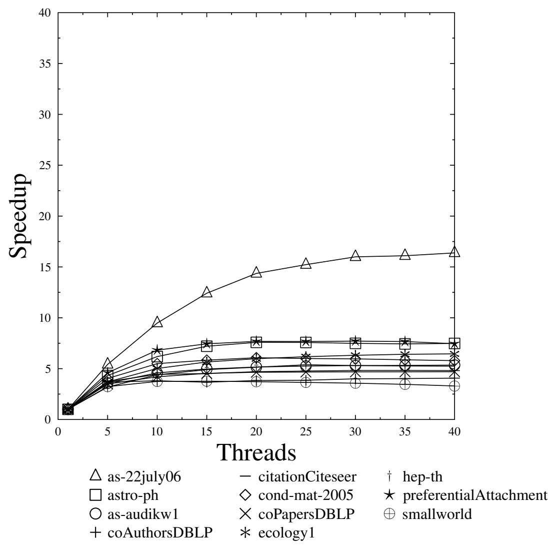
The results in Fig. 6 use $P = 1$ cores.

Note the following observations from Fig. 6:

- Both the minimal and maximal speedups are well distanced from the average speedups. For some of the cases there is an order of magnitude between the average and either of the extremes:
  - For the audikw1 network, there is a $10X$ difference between the minimal speedup and average speedup.
  - For the audikw1 network, there is a $163X$ difference between the minimal speedup and maximal speedup.
- The difference between the 25th percentile and 75th percentile is around $2X - 3X$.



(a) Static graph algorithm.



(b) Dynamic graph algorithm.

Figure 7: Strong scaling of the coarse-grain parallel implementations: (a) static graph algorithm and (b) dynamic graph algorithm. The approximation uses $K = 256$ roots. Each of the curves is for different network.

- The average speedup for most of networks (regardless of their type) is above $50X$.
- For six out of the eleven graphs the maximal speedup is over 1000 and for five out of the six graphs the speedup is over $3000X$.

For all the graphs and all the insertions our dynamic algorithm is faster than the static algorithm. This might be expected since the theoretical complexity bound of the static graph algorithm is an upperbound for our algorithm; however, from a practical perspective, our algorithm can potentially require twice as many floating-point operations as it computes the difference between the iterations. Nonetheless, our dynamic graph algorithm outperforms the static graph algorithm for all the insertions that we tested.

2.4.2 Strong Scaling

Fig. 7 depicts the scaling of the static graph algorithm and dynamic graph algorithm.

Specifically, the speedup is computed as follows:

$$Scaling = \frac{Time_1}{Time_P} \tag{5}$$

Note that the $Time_P$ is the maximal execution time of any of the P threads used.

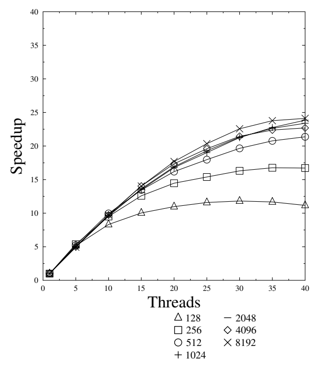
The static graph algorithm achieves significant speedups, Fig. 7 (a), in the range of $25X - 37X$. These speedups are higher than previous results on x86 systems that used the fine-grain approach and are due to the reduced communication cost. While the cores might not receive an equal number of roots and these roots might be in different connected components, based on these speedups - the load balancing is satisfactory.

Fig. 7 (b), depicts the scaling of our dynamic graph algorithm using coarse-grain parallelism. Our algorithm has good speedup up to 5 cores, after which the speedup declines. There are several causes for this result. The amount of work per thread is considerably smaller for the dynamic graph algorithm, especially for the smaller networks. However, the preeminent cause for the decay in the speedup is workload imbalance where a single root can become the execution bottleneck. These will be addressed in the following subsections.

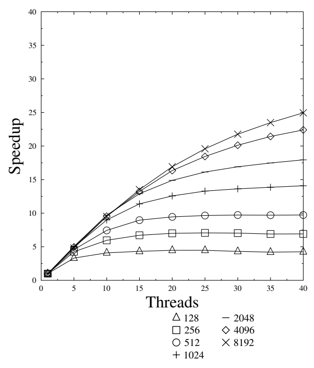
2.4.3 Load Balancing

Fig. 8 shows the scaling of the dynamic algorithm for different root counts. As more roots are used for computing betweenness centrality, the scalability of the dynamic algorithm increases. Recall from (4), the speedup is dependent on the time it takes the thread with the most amount of work to complete. Increasing the total amount of work allows for better partitioning using the coarse-grain approach. This reduces the likelihood of one thread becoming the execution bottleneck. For $astro-ph$ increasing the number of roots from 256 to 8192 increased the speedup from $7X$ to $25X$.

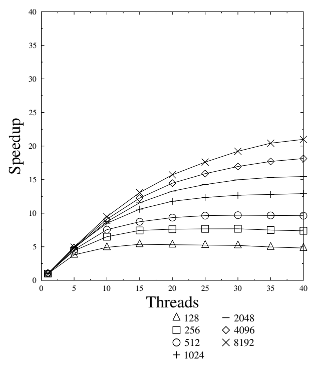
To further validate these results, we compare the ratios of the execution time and the ratios of the number of traversed edges for the thread with the most time



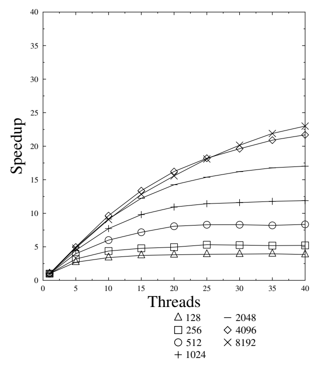
(a) as-22july06



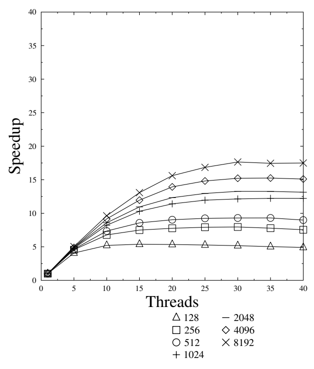
(b) astro-ph



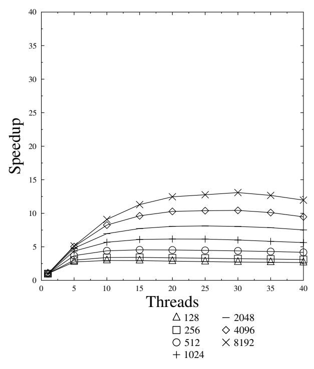
(c) cond-mat-2005



(d) hep-th



(e) preferential attachment

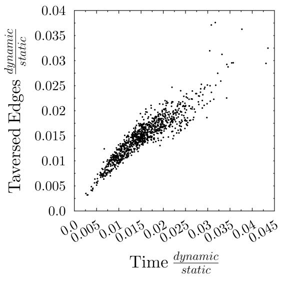


(f) small world

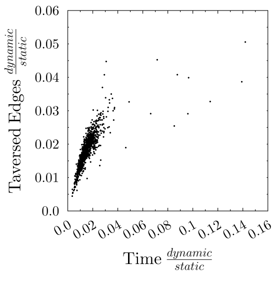
Figure 8: Scaling of the dynamic graph algorithm. The multiple curves represent the speedup for different a number of roots as part of the approximation process. The increase in the number of roots increases the amount of work each thread and allows for better load-balancing which allows for better scaling and speedups.

and largest number of traversed edges. For each graph, we compare these ratios for multiple thread counts: $1, 10, 20, 30$, and 40. These can be found in Fig. 9. The ordinate is the ratio of traversed edges and the abscissa is the ratio of the execution time. Both ratios are for the dynamic graph algorithm in comparison to the static graph algorithm.

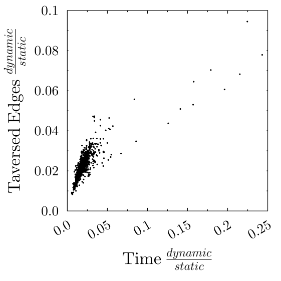
From these sub-figures a clear correlation between the execution time and the traversed edges can be seen. Further, analysis of the execution times and number of roots executed by each thread shows that for many of these cases the execution bottleneck is a single root that traverses more edges than multiple roots combined.



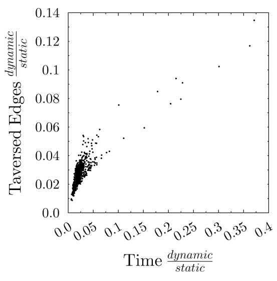
(a) 1 threads



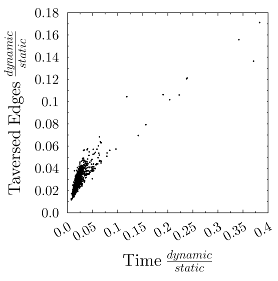
(b) 10 threads



(c) 20 threads



(d) 30 threads

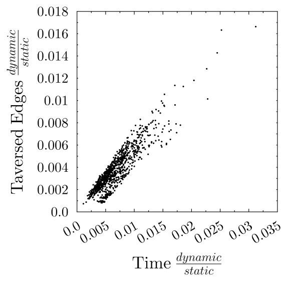


(e) 40 threads

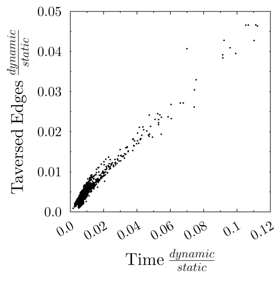
Figure 9: The subfigures show the relationship between the execution of the algorithms and the number of traversed edges for each of the insertions. The abscissa is the ratio in the execution time of the dynamic graph algorithm and the static graph algorithm (with an equal number of threads). The ordinate is the ratio of traversed edges of the dynamic graph algorithm and the static graph algorithm (with an equal number of threads). We show the results for $1, 10, 20, 30$, and 40 threads. These scatter plots show that there is a relationship between execution time and traversed edges. This caption is relevant for Fig. 10 - Fig 14. Results in this figure are for as-22july06 network.

Observe the values of the ordinates for the fourth and fifth columns (30 threads and for 40 threads, respectively). Notice that the maximal values of the ordinate increases with the introduction of new threads. This means that using more threads will not resolve the load-balancing issue.

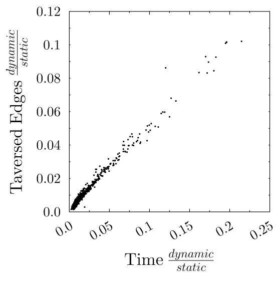
The workload imbalance might be solvable using a fine-grain approach or estimating the amount of work per root and load balancing the work based off this estimation. This load-balancing issue is not within the scope of this work.



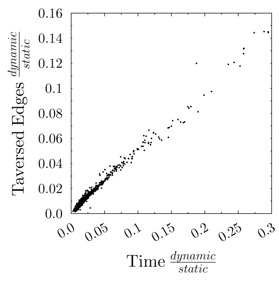
(a) 1 threads



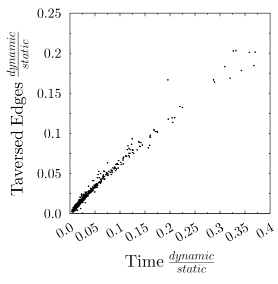
(b) 10 threads



(c) 20 threads

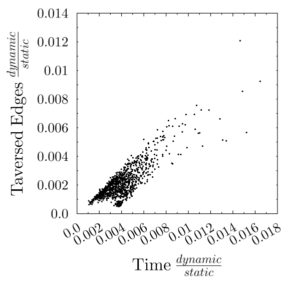


(d) 30 threads

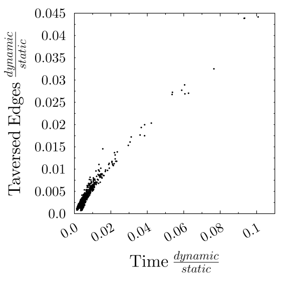


(e) 40 threads

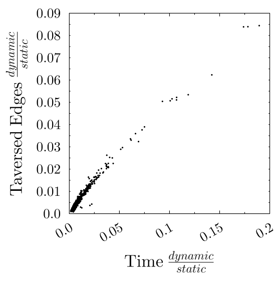
Figure 10: astro-Ph network.



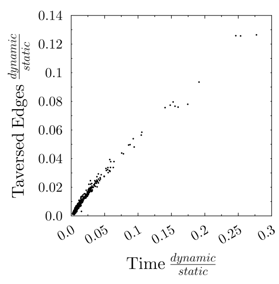
(a) 1 threads



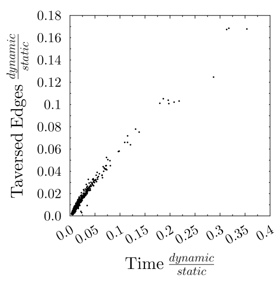
(b) 10 threads



(c) 20 threads

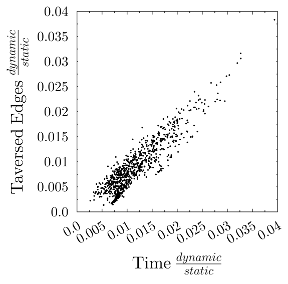


(d) 30 threads

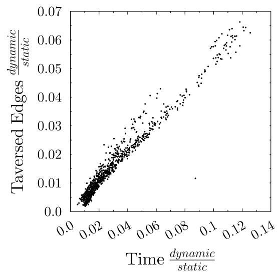


(e) 40 threads

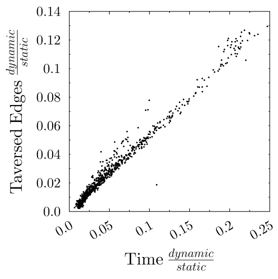
Figure 11: cond-mat-2005 network.



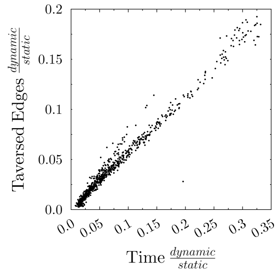
(a) 1 threads



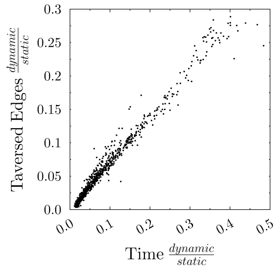
(b) 10 threads



(c) 20 threads

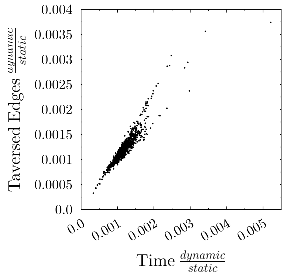


(d) 30 threads

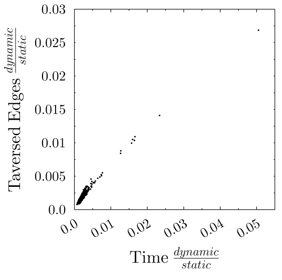


(e) 40 threads

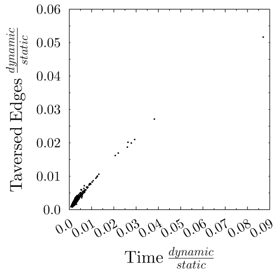
Figure 12: cond-mat-2005 network.



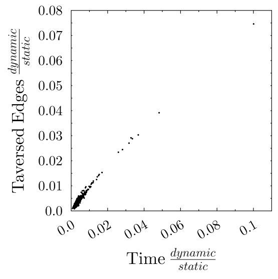
(a) 1 threads



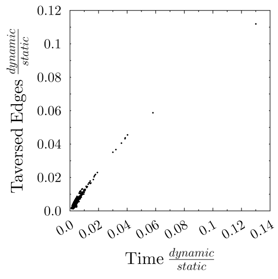
(b) 10 threads



(c) 20 threads

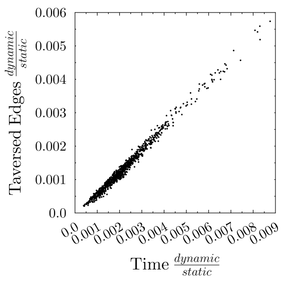


(d) 30 threads

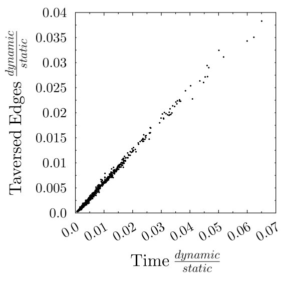


(e) 40 threads

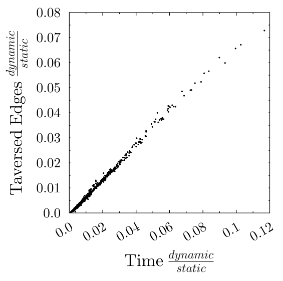
Figure 13: preferentialAttachment network.



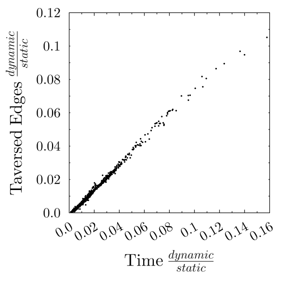
(a) 1 threads



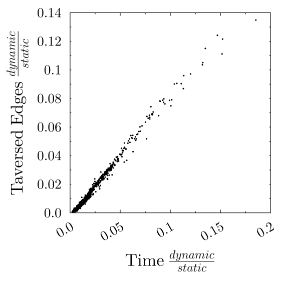
(b) 10 threads



(c) 20 threads



(d) 30 threads



(e) 40 threads

Figure 14: small-world network.

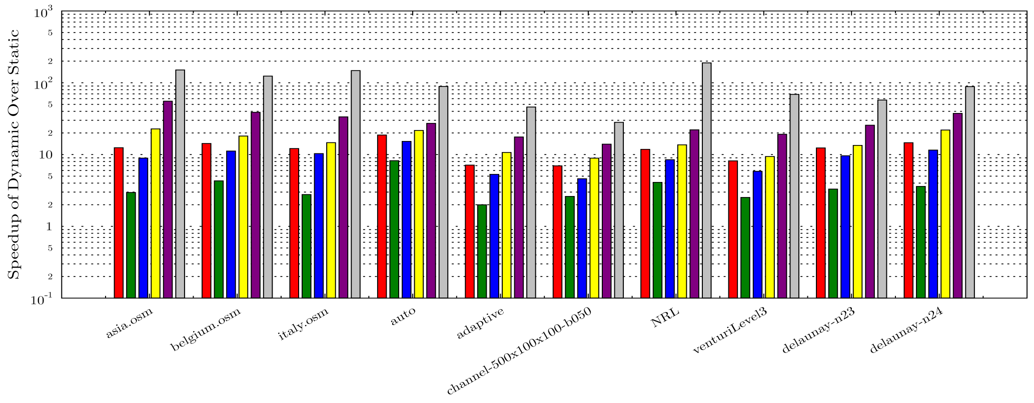
2.4.4 Large Network Analysis

We now present performance analysis of our new algorithm for larger graphs from the DIMACS 10 challenge. These graphs, found in Table 4, are considerably larger than the graphs used earlier, taken from Table 3. The size of these graphs limits some of scaling testing that can be done. We present performance results for both the static graph and dynamic graph algorithm using 1 core and all 40 cores in the system. Instead of doing 1000 insertions, only 50 insertions are done because of the high computational requirements. Note that the largest of these graphs has 16 million vertices and 50 million edges. Even with the $O(K \cdot V)$ storage requirement, our dynamic algorithm had enough memory. Note that many of these graphs have a large diameter and are highly sparse.

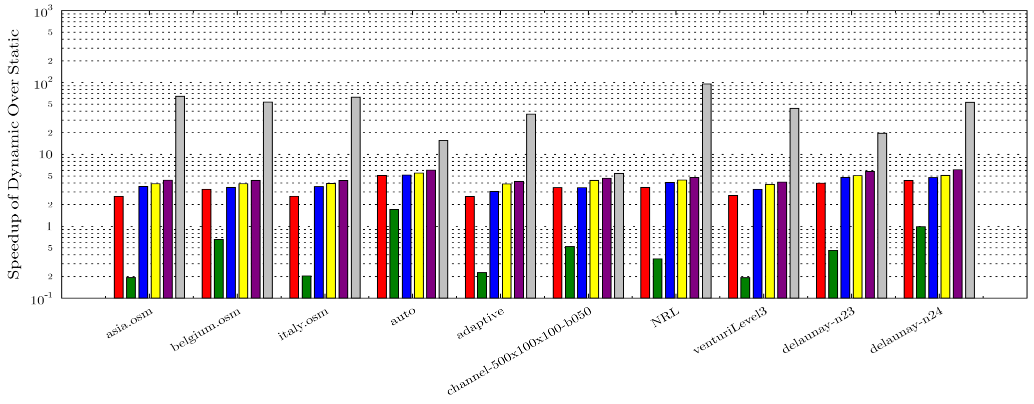
Fig. 15 shows the speedup of our dynamic graph algorithm over the static graph algorithm for 1 thread and 40 threads. Once again we present six different speedup bars for each network (from the left to the right): 1) the average speedup, 2) the

Table 4: Larger graphs from the 10th DIMACS Implementation Challenge. The execution times are for a single thread.

Name	Graph Type	$\lvert V\rvert$	$\lvert E\rvert$	Static (sec.)	Dynamic (sec.)
asia.osm	Street	11,950,757	12,711,603	2212	177
belgium.osm	Street	1,441,295	1,549,970	258	18.3
italy.osm	Street	6,686,493	7,013,978	1243	102
auto	Walshaw	448,695	3,314,611	178	9.56
adaptive	Numerical	6,815,744	13,624,320	1370	192
channel-500x100x100-b050	Numerical	4,802,000	42,681,372	1919	275
NLR	Numerical	4,163,763	12,487,976	1097	92.9
venturiLevel3	Numerical	4,026,819	8,054,237	814	99.8
delaunay-n23	delaunay	8,388,608	25,165,784	1845	149
delaunay-n24	delaunay	16,777,216	50,331,601	3747	256



(a) 1 thread



(b) 40 threads

Figure 15: Six different speedup bar for each graph (from the left to the right): 1) average, 2) minimal, 3) the 25th percentile, 4) median, 5) the 75th percentile, and 6) maximal.

minimal speedup, 3) the 25th percentile speedup, 4) the median speedup, 5) the 75th percentile speedup, and 6) the maximal speedup.

The sequential speedup analysis of the dynamic graph algorithm over the static graph algorithm is depicted in Fig. 15 (a). For these networks, the dynamic graph algorithm is always faster than doing a full static recompute. For many of these graphs, for the 40 core execution (Fig. 15 (b)) the dynamic graph algorithm performs worse than the static graph algorithm. This is in fact due to the workload imbalance that discussed earlier. The static graph algorithm as we showed has considerably good load balancing, whereas the dynamic graph algorithm needs better load balancing. Better load balancing of the dynamic graph algorithm would ensure performance greater or equal to the static graph algorithm. For the average speedup and everything above the 25th percentile, the dynamic graph algorithm outperforms the static graph algorithm.

2.5 Conclusions

In this chapter we showed a novel algorithm for computing betweenness centrality for dynamic graphs. Given that our algorithm extends Brandes's algorithm, we show that the algorithm supports both parallelization and approximation. While we placed more emphasis on edge insertions in the chapter, we also discussed edge deletions and showed the similarities between the two.

We then proceeded to show that our algorithm can analyze graphs with millions of vertices and edges (which is something that was previously not done by other dynamic graph betweenness centrality algorithms). This is followed by a speedup analysis of the dynamic graph algorithm over the static graph algorithm and we show that the speedup can be as high as $7790X$. Further, an analysis of the workload partitioning is given for a coarse-grain implementation and we believe that a fine-grain implementation of dynamic graph approximate betweenness centrality will resolve this workload imbalance. There are most likely additional load balancing opportunities which allow creating a better parallel implementation of this algorithm.

An additional open-ended challenge is creating an algorithm that can do batches of updates. Vertex insertion/deletion can be considered a specific case of batching. Yet, the real benefit maybe from avoiding the same traversal multiple times.

2.6 Formal Proofs for Insertions

In the above, we discussed the four insertion cases and presented the intuition behind their correctness. Here we prove the correctness for the insertions - the proofs for the deletions can be derived from this. For simplicity, assume that that new edge $e = (u, v)$ is inserted into the graph $G = (V, E \cup \{e\})$ and that all the BFS-like trees have been computed and are correct from the previous insertion. Note, that when the insertion scenarios are computed, the edge has been already inserted into the graph. The number of trees that we maintain between the insertions is dependent on the accuracy of our computation: exact or approximate.

While the graph has been updated and includes the edge, the BFS trees (denoted as T_s) have not been updated. We show that our four insertion cases cover the update process correctly. For simplicity and without the loss of generality we assume that u is closer to the root than v.

2.6.1 Same Level Insertion

Same level insertions do not require any additional computation for T_s, Fig 1. This is simply due to the fact that no new shortest paths are created. Recall that $|d_s(u) - d_s(v)| = 0$.

Lemma 1 *Given an edge $e = (u, v)$ such that $d_s(u) = d_s(v)$, no shortest paths go through e.*

Proof 1 *Assume by contradiction that for some vertex w there is a shortest path between s and w that goes through e. This path is denoted by vertices $p_s, p_2, ..., p_u, p_v, ...,$*

p_w. *Obviously, v has a path to s as well, this path is denoted by $\hat{p_s}, \hat{p_2}, ..., p_v$. By creating an alternate path $\hat{p_s}, \hat{p_2}, ..., p_v, ..., p_w$ we have created a shorter path in contradiction with the assumption.* ■

Lemma 2 *The BFS-like tree structure is maintained and the betweenness centrality values are updated correctly for same level insertion.*

Proof 2 *Following Lemma 1, no new shortest paths are created, which means that there are no changes in the dependency accumulation values. This means that no computations are required and the betweenness centrality values do not change.* ■

2.6.2 Adjacent Level Insertion

Prior to the insertion $d(u_{low}) = d(u_{high})+1$, Fig. 2 which is also the case following the insertion. Pseudo-code for updating the betweenness centrality values can be found in Algorithm 2. None of the vertices above u_{low} will have new shortest path to the root. u_{low} will have additional paths to the root through u_{high}.

Lemma 3 *Only vertices found in a BFS traversal starting at u_{low} (and are below u_{low}) will have new paths to the root.*

Proof 3 *Assume by contradiction that some vertex w has a new shortest path to the root, s, through u_{low} and that w is not found in a BFS traversal starting at u_{low}. w can not be above u_{low} as it would have a shorter path to the root. If w does not have a path to the root through u_{low}, then no paths are created. As such the w has a path to the root through u_{low}. Because w has a shortest path to the root via u_{low} it has some ancestral path to u_{low}. As such, this path will be found during the BFS traversal in contradiction to the assumption.* ■

Corollary 1 *If the number of shortest paths to the root has changed for a vertex w then for all the parents of w,$v \in P[w], \delta[v]$ needs to be updated. This change is denoted using $\hat{\delta}[v]$.*

Algorithm 2: Insertion of a new edge in a specific BFS tree where the vertices are in adjacent levels prior to the insertion.

```
Stage 1 - local initilization
$Q_{BFS} \leftarrow$ empty queue;
for $level \leftarrow 1\, to\, V$ do
    $Q[level] \leftarrow$ empty queue;
$dP[v] \leftarrow 0, v \in \forall V$;
$t[v] \leftarrow$ Not-Touched $, v \in \forall V$;
$\hat{\sigma}[v] \leftarrow \sigma[v], v \in \forall V$;
enqueue $u_{low} \rightarrow Q[d[u_{low}]]$;
enqueue $u_{low} \rightarrow Q_{BFS}$;
$t[u_{low}] \leftarrow$ Down;
$dP[u_{low}] \leftarrow \sigma[u_{high}]$;
$\hat{\sigma}[u_{low}] \leftarrow \hat{\sigma}[u_{low}] + dP[u_{low}]$;
Stage 2 - BFS traversal starting at $u_{low}$
while $Q$ not empty do
    dequeue $v \leftarrow Q$;
    for all neighbor $w$ of $v$ do
        if $d[w] = (d[v] + 1)$ then
            if $t[w] =$ Not-Touched then
                enqueue $w \rightarrow Q_{BFS}$;
                enqueue $w \rightarrow Q[d[w]]$;
                $t[w] \leftarrow$ Down;
                $d[w] \leftarrow d[v] + 1$;
                $dP[w] \leftarrow dP[v]$;
            else
                $dP[w] \leftarrow dP[w] + dP[v]$;
            $\hat{\sigma}[w] \leftarrow \hat{\sigma}[w] + dP[v]$;
```

Observations from Algorithm 2:

- Only vertices below u_{low} will update $\hat{\sigma}$.
- The number of new paths will be maintained in the array dP, where $dP[v]$ is the number of new shortest paths to v.
- As vertices can be found in the dependency accumulation, an additional queue is required for the vertices found in the level above. In the pseudo-code we maintain a single queue per level. An actual implementation requires only two additional queues and a slight modification of the code that ensures that the level is not changed until all vertices in the current level (including those found in

Algorithm 2: Cont.

```
Stage 3 - modified dependency accumulation
$\hat{\delta[v]} \leftarrow 0, v \in \forall V$; $level \leftarrow V$;
while $level > 0$ do
    while $Q[level]$ not empty do
        dequeue $w \leftarrow Q[level]$;
        for all $v \in P[w]$ do
            if $t[v]$ =Not-Touched then
                enqueue $v \rightarrow Q[level-1]$;
                $t[v] \leftarrow$ Up;
                $\hat{\delta}[v] \leftarrow \delta[v]$;
            $\hat{\delta}[v] \leftarrow \hat{\delta}[v] + \frac{\hat{\sigma}[v]}{\hat{\sigma}[w]}(1+\hat{\delta}[w])$;
            if $t[v] =$ Up $\wedge(v \neq u_{high} \vee w \neq u_{low})$ then
                $\hat{\delta}[v] \leftarrow \hat{\delta}[v] - \frac{\sigma[v]}{\sigma[w]}(1+\delta[w])$;
            if $w \neq r$ then
                $C_B[w] \leftarrow C_B[w] + \hat{\delta}[w] - \delta[w]$;
    $level \leftarrow level - 1$;
$\sigma[v] \leftarrow \hat{\sigma}[v], v \in \forall V$;
for $v \in V$ do
    if $t[v] \neq$ Not-Touched then
        $\delta[v] \leftarrow \hat{\delta}[v], v \in \forall V$
```

the dependency accumulation) are traversed. Vertices found in the dependency accumulation stage will be marked as touched on the way UP.

- The dependency accumulation stage will traverse a greater or equal number of vertices and edges than the BFS traversal.

To compute the dependency accumulation value, we differentiate between vertices found in the BFS traversal and vertices found in the recursive movement. Note that $\hat{\delta}[v]$ is computed in a similar fashion to $\delta[v]$ with the following modification - use $\hat{\sigma}$ instead of σ. For vertices, that were found in the BFS traversal, their dependency accumulation values are computed from scratch as discussed above using $\hat{\delta}[v]$ and $\hat{\sigma}[v]$.

$$\hat{\delta}[v] \leftarrow \hat{\delta}[v] + \frac{\hat{\sigma[v]}}{\hat{\sigma[w]}}(1 + \hat{\delta[w]}). \tag{6}$$

For the vertices, found in the dependency accumulation only a partial recomputation is needed. Consider a vertex w that has just been found on the way up. w can potentially have additional children below it that have not been found - for these children the dependency accumulation value going through these edges are unchanged and do not require recomputing. For the edges do have a change, the previous value needs to be reduced:

$$\hat{\delta}[v] \leftarrow \hat{\delta}[v] - \frac{\sigma[v]}{\sigma[w]}(1 + \delta[w]). \tag{7}$$

The previous value, $\delta[v]$, is no longer needed, thus, it should be decremented from the centrality value (this is correct for all vertices except for the root as discussed in [30]):

$$C_B[v] \leftarrow C_B[v] + \hat{\delta}[v] - \delta[v]. \tag{8}$$

Lemma 4 *The BFS-like tree structure is maintained and betweenness centrality is updated correctly for an adjacent level insertion.*

Proof 4 *In Lemmas 3 we show that the shortest path count is maintained. As there are no vertices that move in the BFS-like tree following the insertion, the distances are the same and correct. Based on Eq. (8) and Lemma 1 the betweenness centrality metric is update correctly.* ■

2.6.3 Non-Adjacent Level Insertion

This subsection presents the modifications needed for updating the BFS-like tree for an edge insertion that causes vertices to move up the tree as $|d_s(u) - d_s(v)| \geq 2$. From Fig. 3, it is obvious that at least one vertex, u_{low}, changes it position and moves closer to the root. The insertion can in fact pull-up additional vertices - we sketch the necessary steps.

Following the edge insertion, $e = (u_{high}, u_{low})$, we know for a fact that the vertex u_{low} will move up the tree and will be one level below u_{high}. As a consequence of this pull-up, additional vertices might be pulled-up as well. For all neighbors of u_{low} we will check if a new shortest path has been creating due to the pulling up of u_{low}. These neighbors have two options, either they will be pulled-up the BFS-like tree or they will stay as they are. These neighbors will be traversed from a BFS traversal starting at u_{low} and distances from the root will be updated if needed.

Lemma 5 *The BFS-like tree structure is maintained and betweenness centrality is updated correctly for an edge insertion that causes movement.*

Proof 5 *In the process of the BFS traversal starting at u_{low}, each vertex that is encountered will fall into one of the following categories:*

- *If a vertex has moved - its paths to the root are through its parent that causes its movement and its previous paths to the root are no longer relevant as these are no longer above the vertex. In addition to moving the vertex to its new level, it is also necessary to update the previous parents of this vertex as they now have fewer children going through as this is part of the dependency accumulation. This will require require the dependency accumulation value of the entire path up to the root. These vertices will use the formulation of Eq. (6) for computing the dependency accumulation.*

- *If a vertex has not moved, one of the following occurs to the vertex:*

  - *New paths to the root through new parents (its old parent are still relevant), Eq. (6) is used to compute their dependency accumulation value.*

  - *Fewer children below as these might have been pulled up, Eq. (6) is used to compute their dependency accumulation value.*

  - *The vertex keeps the same number of children and same number of parents. For this, as there is no change, it is not necessary to place it into the queue. Such vertices might be found during the dependency accumulation stage.*

Using the explanations and proofs from the Adjacent level insertions, we can update the dependency accumulation of the vertices found on the reverse traversal using Eq. (7).

As such, all the values of the BFS-like tree are maintained and the update correct.

■

CHAPTER III

FASTER BETWEENNESS CENTRALITY

This chapter is an extension of the paper: O. Green, D. Bader, "Faster Betweenness Centrality Based on Data Structure Experimentation", 13th International Conference on Computational Science, 2013.

Betweenness centrality is a graph analytic that states the importance of a vertex based on the number of shortest paths that it is on. As such, betweenness centrality is a building block for graph analysis tools and is used by many applications, including finding bottlenecks in communication networks and community detection. Computing betweenness centrality is computationally demanding, $O(V^2 + V \cdot E)$ (for the best known algorithm), which motivates the use of parallelism. Parallelism is especially needed for large graphs with millions of vertices and billions of edges. While the the memory requirements for computing betweenness are not as demanding, $O(V + E)$ (for the best known sequential algorithm), these bound increase for different parallel algorithms. We show that is possible to reduce the memory requirements for computing betweenness centrality from $O(V + E)$ to $O(V)$ at the expense of doing additional traversals. We show that not only does this not hurt performance it actually improves performance for coarse grain parallelism. Further, we show that using the new approach allows parallel scaling that previously was not possible. One example is that the new approach is able to scale to 40 x86 cores for a graph with 32M vertices and 2B edges, whereas the previous approach is only able to scale upto 6 cores because of memory requirements. We also do analysis of fine-grain parallel betweenness centrality on both the x86 and the Cray XMT.

To avoid repetition, the reader is referred to the introduction and related work sections of Chapter 2.

3.1 Data Structure Experimentation

3.1.1 Relevant Terminology

In Brandes's algorithm [30] (Algorithm 1, Chapter 2), each vertex maintains a list of all its neighbors that are in the level above it. We will refer to these list as either the parent list or the predecessor list, interchangeably, when referring to Brandes's algorithm. In Madduri *et al.* [104], vertices maintain a list of all the the neighbors that are in the level below it. We will refer to these list as either the children list or the successor list, interchangeably, when referring to this algorithm. Obviously, these lists are a subset of the adjacency for a given vertex. The final implementation of the algorithm is dependent on the selection of either the predecessor or successor approach. We refer the reader to both [30] and [104] for further reading. We note that for a sequential implementation, there is no preference to either of these approaches.

As these algorithms share many implementation properties, they do not require separate explanations on many common issues. As such we will refer to these lists as ancestry lists whenever the explanation is relevant for both the approaches. In Section 3.2 we will specify the exact approaches that we used for each test.

3.1.2 Ancestor-List implementation

It is well known that the implementation of a data structure can significantly influence the performance of an algorithm. In some cases, including the case of betweenness centrality, additional parameters about the data structure are known such as the maximal size it will grow and access pattern to the data structure. For Brandes's algorithm we know both of these. The length of each ancestor list is bounded by the vertex's incoming edges. The access pattern is also known - in the BFS stage, elements are pushed to the end of the list and in the dependency accumulation stage traverse the list from the beginning to the end.

Given this additional information, the ancestor lists can be implemented with an array. This is done by pre-allocating an array of $|E|$ element where each vertex, v, is allocated space in the array based on its adjacency. In the Results section we show that despite the increase in the memory requirements, the performance of the array based list is significantly better than the linked-list, as can be expected. Both these implementations have a memory bound of $O(V + E)$.

Obviously the lists can also be implemented using dynamic memory allocations such as a linked-list implementation. The linked-list approaches suffers from several performance pit falls. The first, insertion of an element into the list requires adding a new link to the end of the list. If the node is dynamically allocated, this requires a system call which is usually costly. If the link is added to the list by getting a 'new' link from a pool of preallocated link, then the pool needs to be the size of $O(E)$. As such, it is preferable to use the array based approach because of spacial locality and the reduced number of memory allocations.

3.1.3 Neighbor-Traversal

In this section we present an alternative to maintaining the ancestry lists. This approach reduces the memory requirements from $O(V+E)$ to $O(V)$, increases parallel scalability, reduces synchronization requirement and improves performance for sparse graph. We will see that this new approach, which we refer to as neighbor-traversal, is more computationally demanding the the ancestral based approaches, yet it does not increase the time complexity of the betweenness centrality algorithm of $O(V^2+V\cdot E)$. In the neighbor-traversal approach we eliminate the ancestry list altogether. As such, lists for the vertices are not created during the BFS traversal. Consequently, in the dependency accumulation stage all the neighbors of a given vertex are traversed rather than just the ancestors.

According to Brandes [30], each vertex that is found in the BFS traversal for the first time is placed in a stack and is popped out in the dependency accumulation in the reverse order that it was found. Each vertex is popped out exactly once. Further, each time an additional path is found between a parent and child it is added to its respective list. In the BFS traversal, all the neighbors of a vertex are traversed which has an upper-bound of $O(V+E)$ is given. Given that we only reduce the number of operations required for the BFS traversal, we do not increase the time complexity.

In the dependency accumulation stage, now that the parent lists are not maintained, it is necessary to traverse all the neighbors of a vertex and search for a parent or child. As vertices are accessed in the reverse order in which they were found, each vertex is accessed once and its neighbors are traversed once. This means that the dependency accumulation also has an upper bound time complexity of $O(V+E)$.

Storage Complexity Analysis

The only data structure that requires $O(E)$ memory in the computation of betweenness centrality is the ancestor list. As it is no longer required to maintain the ancestry lists/array, the memory bound of betweenness centrality goes down from $O(V+E)$ to $O(V)$. The remaining data structures only require $O(V)$ memory. As these memory requirements do not change, the memory requirements of betweenness centrality is reduced to $O(V)$.

The coarse-grain parallel algorithm requires that each core maintain a copy of the data structure required by Brandes' algorithm. Each core is responsible for computing the betweenness centrality values for an independent set of vertices. Thus, the storage complexity of the coarse grain granularity using the ancestry list is $O(P \cdot (V+E))$ given that each core needs $O(V+E)$ memory. Consequently, Tan *et al.* state that the coarse-grain parallelism is not scalable due to these memory requirements. Following the removal of the ancestry list, the memory requirement of the coarse-grain approach is reduced from $O(P \cdot (V+E))$ to $O(P \cdot V)$. As such, the coarse grain approach becomes practical and is more scalable than the ancestor approaches. This will become apparent in the Results section.

We now discuss the implications of our new approach on existing algorithms. The medium-grain [135] and fine-grain algorithms [15, 104] require atomic instructions to maintain the ancestry lists. In the first parallel implementation of betweenness centrality [15] the parents list is used. To update the betweenness centrality of a parent, in the dependency accumulation stage, it is necessary to acquire a lock on the parent. In [104] a lock-free algorithm for betweenness centrality is shown where the predecessor list is replaced with a successor list. This approach swaps the roles between the parents and children. For both these algorithms, it is possible to replace the ancestry lists with the neighbor-traversal approach. The benefit of using the neighbor-traversal over ancestry lists for these parallel granularities is atomic instructions are no longer needed for serializing the insertions of elements into the ancestry lists. We note that the neighbor-traversal approach can be implemented either using the predecessor approach or successor approach in which in the dependency accumulation stage the algorithm looks for parents in the level above or children in the level below, respectively. For undirected graphs, both methods will work fine, however, for directed graphs the parent approach can not be used in the dependency accumulation stage of the neighbor-traversal as a vertex will not have access to its incoming edge as is customary for many graph representations such as in the CSR (Compressed Sparse Row) representation. As such, we recommend using the successor approach when using the neighbor-traversal approach.

3.2 Results

We present experimental results of the traversal-based algorithm in this section on both the x86 architecture and the Cray XMT2. The x86 system used for testing is a multi-processor multi-core Intel server. The Intel system has four processors each containing a 10-core Intel Xeon E7-8870 with at 2.4 GHz clock rate. This gives a total of 40 physicals cores. As the E7-8870 supports Hyper-Threads, the system has a total of 80 logical cores; however, for our testing the Hyper-Threads was disabled. Each processor has 30MB of L3 cache shared by all the cores on that processor. This system has a total of 256 GB of DDR3 RAM clocked at 1066 MHz. The Cray XMT2 is the second generation of the Cray XMT [93] system. The XMT architecture is a shared memory system with a massive thread count. The XMT2 we used is the system at the Swiss National Supercomputing Centre that has 64 processors and 2 TB of main memory. Each of the processors contains 128 hardware streams. A different stream can be executed at every clock cycle. The massive thread count enables applications with irregular memory accesses to overcome the latency. In addition to this, the XMT offers low-overhead synchronization through atomic fetch-and-add instructions and full-empty bit memory semantics. As such the XMT is highly suited for graph problems.

3.2.1 Random Graphs

Recursive Matrix (R-MAT) [39] is a graph generator used to create synthetic scale-free graphs that follow properties found in real-world networks. For simplicity, we introduce R-MAT using an adjacency matrix (though this is conceptual). Initially, the adjacency matrix is empty, and edges are added one at a time. For each newly inserted edge, the adjacency matrix is divided into equal-size quadrants where each has a different probability of being selected. One quadrant is selected using a random number generator. This quadrant is recursively subdivided into smaller equal-size quadrants from which the next random selection is made. This process is repeated until each quadrant contains only a single element in the adjacency matrix. The last round randomly selects a single element and creates the corresponding edge. The probabilities assigned to the quadrants are designated a, b, c, and d where $a+b+c+d = 1$ and $0 \leq a, b, c, d \leq 1$. The R-MAT generator can create Erdős-Rènyi (ER) [61][62] random graphs when $a = b = c = d = 0.25$. ER graphs are random graphs with a uniform edge distribution. We use RMAT-I to denote an RMAT graph with 2^I vertices. For example an RMAT-24 graph has $2^{24} = 16M$ vertices.

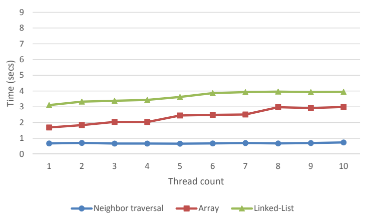
3.2.2 Coarse Grain Parallelism

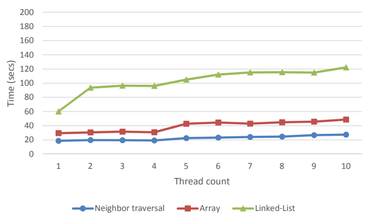
In this section we will take a look at experimental results of the coarse grain parallel neighbor-traversal implementation. The coarse grain approach is appropriate for the x86 architecture as it does not require synchronization (locks and atomic instructions) which is usually costly. Furthermore, this architecture does not benefit from increasing the number of threads beyond the number of physicals cores, as such, we limited the thread count by the maximal number of cores available.

On this system we compared the different algorithmic approaches for computing betweenness centrality: neighbor-traversals, parent list with an array implementation, and parent list using a linked list implementation. As both scalability and performance were the metrics of interest, we ran a variety of tests on the system. We ran the algorithms on multiple scales (vertex count), average edge degree, and thread count. All tests were conducted on undirected graphs.

The results in this subsection are for weak scaling, where each core is supplied with a constant amount of work, regardless of the number of the cores. This means that each of the cores was responsible for computing betweenness for the same number of roots.

For each graph size and average edge adjacency, we checked the scalability for a single processor (upto 10 cores) and for the entire system (all 40 cores). Tests on the single processor allow a speedup analysis that does not require dealing with memory subsystem issues such as cache coherency and cross processor communication. The memory contention becomes apparent for the higher thread counts. Note, that


(a) $|V| = 2^{18}, |E| = 64 \cdot |V|$.


(b) $|V| = 2^{22}, |E| = 64 \cdot |V|$.

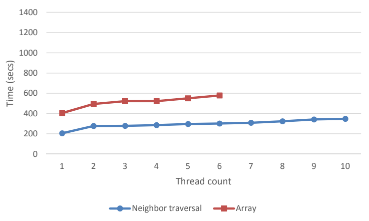
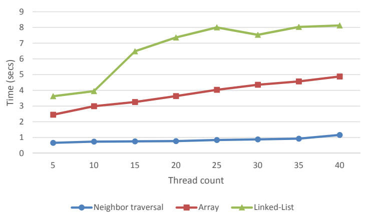
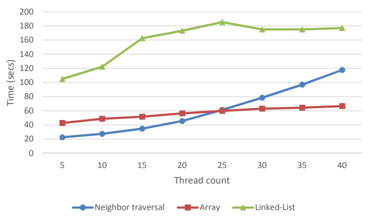

(c) $|V| = 2^{25}, |E| = 64 \cdot |V|$.

Figure 16: Weak scale testing for thread counts of 1 to 10. Lower is better.


(a) $|V| = 2^{18}, |E| = 64 \cdot |V|$.


(b) $|V| = 2^{22}, |E| = 64 \cdot |V|$.

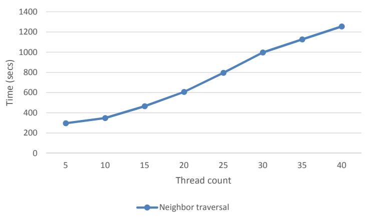

(c) $|V| = 2^{25}, |E| = 64 \cdot |V|$.

Figure 17: Weak scale testing for thread counts of 5 to 40 in multiples of 5. Lower is better.

Hyper-Threads option was not used for testing performance, however, the neighbor-traversal approach could scale to 80 threads for many of the test cases because of the considerably lower memory requirements.

We tested the algorithms on multiple graph sizes from RMAT-18 to RMAT-27. For all the graphs between RMAT-18 and RMAT-25, four different average edge degrees of $8, 16, 32$, and 64 were tested. For RMAT-26, average edge degrees of $8, 16$, and 32 were tested. For RMAT-27, average edge degrees of 8 and 16 were tested. All these, were tested for the above mentioned thread counts. We note that for both the array based approach and the linked list approach, there were instances in which the algorithm was not able to complete due to the memory requirements of these methods. Further, there were instances in which the algorithms timed-out. We elaborate on these cases.

We divide the test cases into 4 sets: small graphs (RMAT-18 - RMAT-19), medium sized graphs (RMAT-20 - RMAT-22), large graphs (RMAT-23 - RMAT-25), and extra large graphs (RMAT-26 - RMAT-27). We analyze each of these independently and in the following order: small graphs, large graphs, extra large graphs, and medium graphs. The order of the explanations moves from the simplest case to the more complicated case.

For the small graphs, all the different algorithms completed for all the different graph sizes, edge counts and thread counts. In Fig. 16 (a) and Fig. 17 (a) the execution times for the three different implementations are depicted. The small graph is a RMAT-18 graph with an average edge degree of 64. Note that for all thread counts the neighbor-traversal approach is faster than the other implementations.

For the larger graphs, there were many cases that the array approach algorithm was not able to scale to the to 30-40 threads due to memory constraints, as such the neighbor-traversal out preformed the array approach for these thread counts. Note that the amount of memory required for array approach was so large, that there was increased number of cache misses. The linked-list approach simply timed out even for the smaller thread counts. The neighbor-traversal scaled to all 40 cores for all the graph scales and average edge degreess. In Fig. 16 (c) and Fig. 17 (c) the execution times for the three different implementations are depicted. The large graph is a RMAT-25 graph with an average edge degree of 64. The array based approach was not able to scale beyond 6 cores, whereas, the neighbor-traversal approach scaled up to 40 cores. For the extra large graphs it is clear that the neighbor-traversal out preformed the other algorithms, simply as it was able to scale to all the thread counts. For the sake of brevity we, do not present an additional figures, yet, we note that such a figure would be similar with fewer points on the graph for the array approach.

For the medium size graphs a more qualitative explanation is presented given that for some of the test cases the neighbor-traversal approach was outperformed by the array approach. In Table 5 it is possible to see the cases that the array based approach

Table 5: The scenarios for the medium size graphs where the different approaches have better performance. The array based approach is denoted with $\circ$ and the neighbor-traversal is denoted as $\diamond$. The first column denotes the graph size and average edge adjacency. The remaining columns represent the thread count. The neighbor-traversal approach dominates the table.

(Size, Avg|E|)	1	2	3	4	5	6	7	8	9	10	15	20	25	30	35	40
(18,8)	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$
(18,16)	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$
(18,32)	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$
(18,64)	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$
(19,8)	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$
(19,16)	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$
(19,32)	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$
(19,64)	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$
(20,8)	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$
(20,16)	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\circ$
(20,32)	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\circ$	$\circ$
(20,64)	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\circ$	$\circ$
(21,8)	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\circ$
(21,16)	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\circ$	$\circ$	$\circ$
(21,32)	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\circ$	$\circ$	$\circ$
(21,64)	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\circ$	$\circ$	$\circ$
(22,8)	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\circ$	$\circ$
(22,16)	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\circ$	$\circ$	$\circ$	$\circ$
(22,32)	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\circ$	$\circ$	$\diamond$	$\diamond$
(22,64)	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\circ$	$\circ$	$\diamond$	$\diamond$
(23,8)	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$
(23,16)	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$
(23,32)	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$
(23,64)	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$
(24,8)	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$
(24,16)	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$
(24,32)	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$
(24,64)	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$
(25,8)	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$
(25,16)	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$
(25,32)	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$
(25,64)	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$
(26,8)	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$
(26,16)	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$
(26,32)	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$
(27,8)	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$
(27,16)	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$

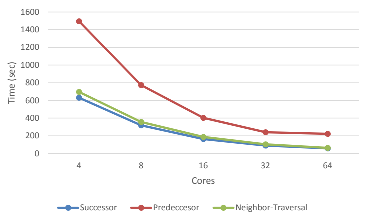
out performed the neighbor-traversal approach. These cases are marked with a red circle. For all remaining graph sizes, average edge degrees and thread counts the neighbor-traversal approach outperformed the array based approach.

In Fig. 16 (b) and Fig. 17 (b) the execution times for the three different implementations are depicted. The medium graph is a RMAT-22 graph with an average edge degree of 64. It is well worth noting that the array approach has the better performance only for the case that two or more processors (20 cores) are used. This can be explained by the fact the neighbor-traversal approach has a greater number of memory access and requires more data to be transmitted between different sockets. The increase in the number of processor present several communication limitations which include limited bandwidth, cache coherency, and non-uniform memory access (NUMA).

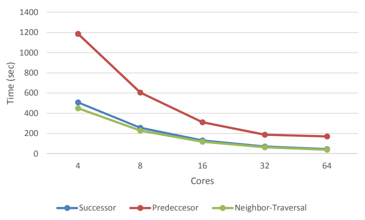
As such, an increase in communication (memory requests) can reduce the overall performance of the system as it happens for many of the medium sized graph scenarios when moving from 2 sockets (maximum of 20 threads) and upwards. However, as the graph becomes denser and the vertices have more parents in the adjacent level, the number of memory requests to the parent array increases. So does the number of cache misses due to contention. Cache misses can be just as expensive as over saturating the memory bandwidth and thus the performance of the array approach goes done and the neighbor-traversal approach starts outperforming the array approach again.

3.2.3 Fine Grain Parallelism

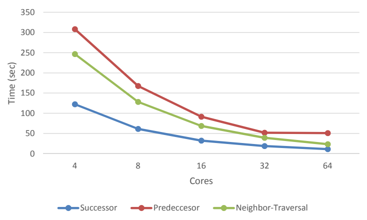
In this subsection we present performance results of the fine grain parallel betweenness centrality using both the ancestral approach and the neighbor approach on both the Cray XMT and the x86. For both these architectures we used and modified existing software packages: GraphCT [1] for the Cray XMT and Georgia Tech's SNAP [2, 16] for the x86.



(a) Execution for both BFS and dependency accumulation stages.



(b) Execution for BFS stage.



(c) Execution for dependency accumulation stage.

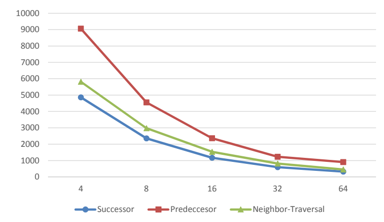
Figure 18: Execution time for approximate betweenness centrality on the Cray XMT2 for a considerably sparse graph. An RMAT-24 graph is used with an average edge degree of 8. 256 roots were used as defined in the HPCS SSCA [4] specifications.

3.2.3.1 GraphCT on the Cray XMT

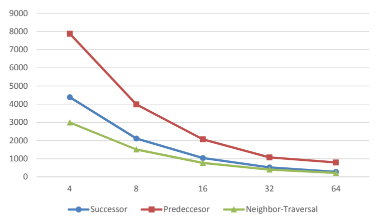
We compare the execution times of three variations of computing approximate betweenness centrality on the Cray XMT using fine grain parallelism: parent array approach [15], successor array approach [104], and successor neighbor-traversal approach. The first two are part of the GraphCT [1] software package.

For the XMT testing we used a RMAT-24 graphs with two different average edge degrees: 8 and 64. We refer to the first graph as the sparse graph and the second graph as the dense graph. For each graph 256 vertices were selected as roots as is specified in the HPCS SSCA [4] specifications. We timed the the two main stages of betweenness centrality, the breadth first search stage and the dependency accumulation stage. Fig. 18 and Fig. 19 present the timing the running times of the algorithms for the sparse and dense graph, respectively. The Successor approach in these figures refers to the [104] algorithm and the Predecessor approach refers to the [15] algorithm.

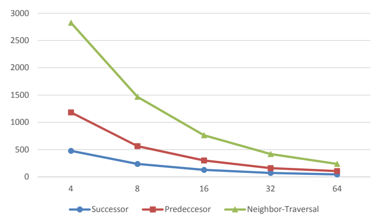
As can be seen, the new traversal algorithm out preforms the predecessor array and successor array based approaches in the BFS stage. The time spent in the parallel BFS dominates the execution time for both the sparse and dense graphs for all the implementations. On the other hand, the dependency accumulation stage in the predecessor array approach outperforms the dependency accumulation stage in the



(a) Execution for both BFS and dependency accumulation stages.



(b) Execution for BFS stage.



(c) Execution for dependency accumulation stage.

Figure 19: Execution time for approximate betweenness centrality on the Cray XMT2 for a slightly denser sparse graph. An RMAT-24 graph is used with an average edge degree of 64. 256 roots were used as defined in the HPCS SSCA [4] specifications.

neighbor-traversal approach. The initial intuition behind the reduced performance can be can be explained by the fact the neighbor-traversal approach requires more memory operations. However, that is only a partial explanation.

The main reason for the reduced performance is due to load balancing issues that occur because of the different number of adjacencies that the vertices have. A single vertex with an extremely high number of adjacencies can cause the imbalance. This was also a problem for the successor based approach [104].

By generating an ER random graph we are able to confirm the workload imbalance. As ER graphs have a uniform distribution of edges over the vertices, the workload is considerably balanced for both the BFS traversal and the dependency accumulation. As expected both stages achieved better scalability (near linear scalability). For the sake of brevity, we do not present these graphs. It is also worth noting that detecting these workload imbalances for a small number of cores is difficult.

3.2.3.2 SNAP on the x86

We compare the execution times of the coarse parent array based approach and the neighbor-traversal approach on the x86. For these tests we used the implementations taken from the Georgia Tech's SNAP [2, 16] graph package. For the neighbor-traversal approach, we simply removed the none relevant code. For these tests we used a

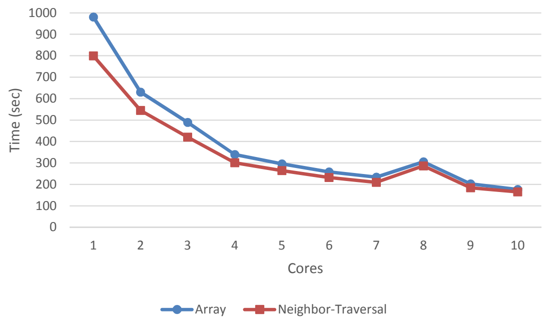


Figure 20: Execution time of the fine grain parallelism using the SNAP package.

RMAT-21 21 graph with an average edge degree of 8. Once again a total of 256 roots were selected, as specified in the HPCS SSCA [4] specifications. Fig. 20 depicts the run-time of the fine grain algorithms. For these implementations, the neighbor-traversal code outperforms the array based approach.

3.2.4 Additional Uses

In this subsection we discuss the benefit of using neighbor-traversal in additional betweenness centrality algorithms. Table 6 contains a list of different betweenness centrality algorithms and their storage complexity using both the ancestor array approach and the new neighbor-traversal approach. We differentiate between computation of betweenness centrality for static graphs and streaming graphs. Streaming graphs are graphs that dynamically change over time with updates such as edge insertion or deletion. Further, we differentiate between exact and approximate betweenness centrality.

Note that the storage complexity for all the algorithms in Table 6 has been reduced using the neighbor-traversal approach. Also note, that the storage complexity of these algorithms is no longer dependent on the $O(E)$. This is especially crucial for the exact streaming algorithm [77] which previously had a storage complexity of $O(V \cdot (V+E))$ and is now $O(V^2)$. This improvement is also be helpful for computing approximate streaming betweenness centrality, where K refers to the number of roots that will be

Table 6: Memory bounds for different betweenness centrality algorithms.

Algorithm	Ancestor-array based approach	Neighbor-traversal approach
Exact static [30]	$O(V+E)$	$O(V)$
Approximate static [17]	$O(V+E)$	$O(V)$
Exact streaming [77]	$O(V \cdot (V+E))$	$O(V^2)$
Approximate streaming	$O(K \cdot (V+E))$	$O(K \cdot V)$
Parallel fine grain [15, 104]	$O(V+E)$	$O(V)$
Parallel coarse grain	$O(P \cdot (V+E))$	$O(P \cdot V)$

used in the computation, giving the approximate algorithm a storage complexity of $O(K \cdot V)$.

3.3 Conclusions

In this chapter we presented an additional approach for computing betweenness centrality. While the new approach does more operations it does not increase the theoretical complexity. Furthermore, the new approach increases the scalability due to a reduced storage complexity and allows for analyzing larger graphs. The performance of the new algorithm outperforms the previous ones. We showed the increased performance on two different architectures, Intel x86 and Cray XMT, using two different parallel granularities: coarse grain and fine grain. For the x86 architecture we showed both weak scaling and strong scaling results.

An additional conclusion of our work, is that computing edge betweenness centrality, as suggested Newman and Girvan [111], using the coarse grain approach does not scale due to the storage requirements as each thread requires an $O(E)$ memory and thus a total of $O(P \cdot (V+E))$. Vertex betweenness centrality has a reduced storage complexity compared to edge betweenness centrality.

In conclusion we leave several open ended issues that we believed should be addressed in the hope of better understanding computation of betweenness centrality:

1. Is there a certain graph density from which it becomes beneficial to use the ancestral approach over the neighbor-traversal approach?

2. Is there a way to model the computation of betweenness centrality based on graph properties (without actually computing betweenness centrality) such that the number of memory accesses for a given algorithm can be approximated? This would allow for auto-tuning for betweenness centrality and perhaps for additional graph algorithms.

3. As we saw for the fine-grain parallelism scenario (on the Cray XMT) , the dependency accumulation in the successor approach suffers from an unequal workload balance. Is there a way to address this problem that would not require locks? Or if locks are used, can they be used such that the performance will not be affected too much?

CHAPTER IV

VERTEX COVER CLUSTERING COEFFICIENTS

This chapter is an extension of the paper: O. Green, D. Bader, "Faster Clustering Coefficient Using Vertex Covers", 5th ASE/IEEE International Conference on Social Computing, Washington DC, 2013.

Clustering coefficients, also called triangle counting, is a widely-used graph analytic for measuring the closeness in which vertices cluster together. Intuitively, clustering coefficients can be thought of as the ratio of common friends versus all possible connections a person might have in a social network. The best known time complexity for computing clustering coefficients for sparse graphs uses adjacency list intersection and is $O(V \cdot d_{max}^2)$, where d_{max} is the size of the largest adjacency list of all the vertices in the graph. In this work, we show a novel approach for computing the clustering coefficients in an undirected and unweighted graphs by exploiting the use of a vertex cover, $\hat{V} \subseteq V$. This new approach reduces the number of times that a triangle is counted by as many as 3 times per triangle. The complexity of the new algorithm is $O(\hat{V} \cdot \hat{d}_{max}^2 + t_{VC})$ where $\hat{d}_{max}$ is the size of the largest adjacency list in the vertex cover and t_{VC} is the time needed for finding the vertex cover. Even for a simple vertex cover algorithm this can reduce the execution time $10\% - 30\%$ while counting the exact number of triangles (3-circuits). We extend the use of the vertex cover to support counting squares (4-circuits) and clustering coefficients for dynamic graphs.

4.1 Introduction

Clustering coefficients is a graph analytic that states how tightly bound vertices are in a graph [138]. The tightness is measured by computing the number of closed triangles in the graph, which can then imply the small-world property. Computing the clustering coefficients has been applied to many types of networks: communication [125], collaboration [133], social [133], citation [99], and biological [23]. In social networks, one can think of the local clustering coefficients as the ratio of actual mutual acquaintances versus all possible mutual acquaintances. Applications using clustering coefficients include DIMES [125], which is a distributed platform used for providing an accurate, and comprehensive map of the Internet and automatic Web-Spam detection [27].

There are two types of clustering coefficients: global and local. The global clustering coefficient is a single value computed for the entire graph, whereas the local clustering coefficient is computed per vertex. Both are computed in a similar fashion. Without the loss of generality, we consider the global clustering coefficient in this work; nonetheless our approach is applicable for computing local clustering coefficients. Table 7 presents the notations used in this chapter.

Definition 1 *Formally, the global clustering coefficient is:*

$$CC_{global} = \sum_{v \in V} CC_v = \sum_{v \in V} \frac{tri(v)}{deg(v) \cdot (deg(v) - 1)}$$

There are several approaches for computing clustering coefficients:

1. Enumerating over all node-triples. This has an $O(V^3)$ upper bound complexity.

2. Using matrix multiplication. This has an $O(V^w)$ complexity where $w \leq 2.376$ [44].

3. Intersecting adjacency lists. This has an $O(V \cdot d_{max}^2)$ upper bound [123] where d_{max} is the vertex with largest adjacency.

In this chapter we show a novel and intuitive way to improve the adjacency list intersection approach that removes redundant list intersections, therefore reducing

Table 7: Notations in this chapter

Name	Description
CC_{global}	Global clustering coefficient
CC_v	Clustering coefficient for vertex v
$deg(v)$	Degree of vertex v.
$tri(v)$	Number of triangle that vertex v is in.
$\hat{V}$	A vertex cover of the graph, $\hat{V} \subseteq V$.
d_{max}	Vertex with maximal degree in the graph.
$\hat{d}_{max}$	Vertex with maximal degree from the vetex cover.
u, v, w, x	Vertices in the graph.

the run-time of the algorithm. We accomplish this by only intersecting the adjacency lists of vertices that are marked in an arbitrary vertex cover.

We then extend our method for computing a circuit of any length and illustrate this with circuits of length 4. We show that our new approach can also be applied to dynamic graphs.

The remainder of the chapter will be structured as follows. Section II discusses the related work and discusses real world graph properties and introduces vertex covers. In Section III, we present our new algorithm for counting 3-circuits (clustering coefficients), 4-circuits, and extend the vertex cover approach for computing clustering coefficients for dynamic graphs. In Section IV, we discuss our experimental methodology and present quantitative results. Finally, in Section V, we present our conclusions and discuss future work.

4.2 Related Work

In this section we review the literature that addresses the challenge of computing clustering coefficients for large dynamic graphs. These approaches take into account optimizations such as parallelization, approximation, and dynamic algorithms that make only local modifications to the graphs. These optimizations are crucial in order to analyze social networks such as Facebook [7] and Twitter [9] which can potentially have million of members and thousands of updates per second. The combination of these approaches allows scaling of the data set.

Clustering coefficients is an analytic that is relatively simple to parallelize because of the relatively large number of independent operations that can be executed. The exact parallel granularity that is used and whether it is local or global will define the synchronization methods required, which is important as the number of threads in a system increases and when full system utilization is desired. An additional benefit of computing clustering coefficients for dynamic graphs is that a given update requires modifying only local values.

We now discuss algorithms that have tackled the challenges mentioned above. We add that our new algorithm is an orthogonal optimization to the ones mentioned, meaning that our approach would also benefit from parallelization and the creation of a dynamic algorithm.

In [53], a massively multithreaded architecture, the Cray XMT2, is used for computing dynamic clustering coefficients for a graph with half a billion edges. The Cray XMT2 is a massively parallel system that supports several thousand light-weight concurrent threads with a Uniform Memory Architecture (UMA) that allows for fine-grain parallelism. To achieve high system utilization on the XMT2 for the dynamic case, the incoming updates are batched together and dealt with concurrently by the various threads. In [98] a GPU implementation of clustering coefficients is presented.

Approximation and dynamic data flows are the focus of [27, 35]. These show different techniques for approximating clustering coefficients. In [53] an additional approximate algorithm is given that is based on Bloom filters. The Bloom filters are created only for a subset of the vertices in the batch.

4.2.1 Vertex Covers

A Vertex Cover is a set $\hat{V} \subseteq V$ such that for all $(a, b) \in E$, either $a \in \hat{V}$ or $b \in \hat{V}$ (or possibly both). A minimal vertex cover is the smallest $\hat{V} \subseteq V$ meeting the requirement above. Finding a vertex cover is a well studied problem [45, 86, 95, 20, 84]. Finding the minimal vertex cover is known to be NP-Complete (NPC) [69, 91]. We do not extend the discussion on this as it not relevant for this chapter.

While finding the minimal vertex cover is intractable, there are numerous algorithms that can find some vertex cover in polynomial time, including linear time [45, 86, 95, 20]. In [95] a parallel algorithm for computing the vertex cover based on the A* formulation is given.

A theoretic PRAM algorithm is given in [86] where the vertex cover is found in iterative fashion using an Euler circuit. A total of $O(V + E)$ processors are required and the time complexity is $O(log^3(E))$.

4.3 Vertex Cover Clustering Coefficients

In this section we show a new algorithm for finding the clustering coefficients using vertex covers that avoid wasteful neighbor intersections.

4.3.1 Clustering Coefficients for Static Graphs

In the following subsections we discuss both theoretical and practical implications of the the vertex cover. For simplicity, assume that a vertex cover of $\hat{V}$ of $G = (V, E)$ is given.

Consider a triangle made up of three vertices u, v, w. These vertices may potentially have additional adjacencies; for our discussion these edges are not relevant. Now consider a vertex cover over this triangle. Fig. 21 depicts all legal vertex covers of a triangle. Note that a closed triangle, made up of $u, v, w \in V$, can be represented as six different tuples: $(u, v, w), (u, w, v), (v, u, w), (v, w, u), (w, u, v), (w, v, u)$. All these tuples need to be counted. Note, that by using a lexicographical sorting, only three of these tuples need to be counted. This however, does not change the relevance of our approach as our algorithm benefits from the lexicographical sorting as well.

Lemma 6 *At least two out of the three vertices that make up a triangle are within the vertex cover, $\hat{V}$. By contradiction, consider the case that only one vertex is in the vertex cover, $\hat{V}$, without the loss of generality, assume $v \in \hat{V}$ and $u, w \notin \hat{V}$. As such there is an edge $(u, w) \in E$ that is not covered, meaning that $\hat{V}$ is not a vertex cover.*
■

Lemma 7 *Given two vertices $u, v \notin \hat{V}$, there are no triangles in the graph in which both u and v are in the same triangle. This is immediate from Lemma 6.* ■

As a results of these two Lemmas, we only need to intersect the adjacency lists of two adjacent vertices when both vertices are marked in the vertex cover. That is, for $u, v \in V \wedge (u, v) \in E$ such that either:

1. $v \in \hat{V} \wedge u \notin \hat{V}$
2. $v \notin \hat{V} \wedge u \in \hat{V}$

the intersection between these vertices is not required.

In the original algorithm, this intersection is indeed computed. We now show that it is only necessary to intersect adjacency lists of adjacent vertices that are both in the vertex cover.

We denote C as the set of the intersections between the two vertices. If C is empty, these vertices do not have common neighbors. If C is not empty then for each $w \in C$ the following two scenarios can happen: 1) $w \in \hat{V}$ or 2) $w \notin \hat{V}$. Both these scenarios are depicted in Fig. 21.

Using the above observations we show that it is possible to compute the clustering coefficient by intersecting the adjacency lists of two vertices u, v such that $u, v \in \hat{V} \wedge (u, v) \in E$. We are required to show that all six tuples are accounted for. We start with the case that $w \in C \wedge w \in \hat{V}$, as such $u, v, w \in \hat{V}$. When intersecting u with v, w is found. When intersecting v with u, w is found again. The intersection

Algorithm 3: Pseudo-code for computing clustering coefficients given a vertex cover $\hat{V}$.

```
$triangles \leftarrow 0$;
for $v \in \hat{V}$ do
    for $u \in adj(v)$ do
        if $u = v$ then
            next $u$;
        if $u \in \hat{V}$ then
            $C \leftarrow intersect(v, adj(c), u, adj(v))$;
            for $c \in C$ do
                if $c \in \hat{V}$ then
                    $triangles \Leftarrow triangles + 1$;
                else
                    $triangles \Leftarrow triangles + 3$;
```

will be repeated for intersecting u with v, v with w and their respective reverse orders. For each of the triangles found, the global triangle counter is increased. All six tuples have been accounted for.

For the second scenario when $w \in C \wedge w \notin \hat{V}$, the only intersection computed is between vertices u and v and vertices u and v. w is not intersected with other vertices as it is not in the vertex cover. As such, some intersections are omitted. Yet, the omitted intersections can be accounted for by a simple counting correction where each intersection is counted as 3 different triangles. Given the two computed intersections, u with v and v with u, all the six triangles are properly accounted for.

Given the above, it is possible to create an algorithm for computing the clustering coefficients given a vertex cover that reduces the number of neighbor intersections. The pseudo-code for this is given in Algorithm 3. Note the algorithm assumes that a vertex cover has been computed. We extend the discussion on the time complexity of vertex covers in a following subsection.

4.3.2 Complexity Analysis

In this section we do a complexity analysis of our new algorithm and discuss the cost of computing the vertex cover in terms of time complexity. The time complexity of the

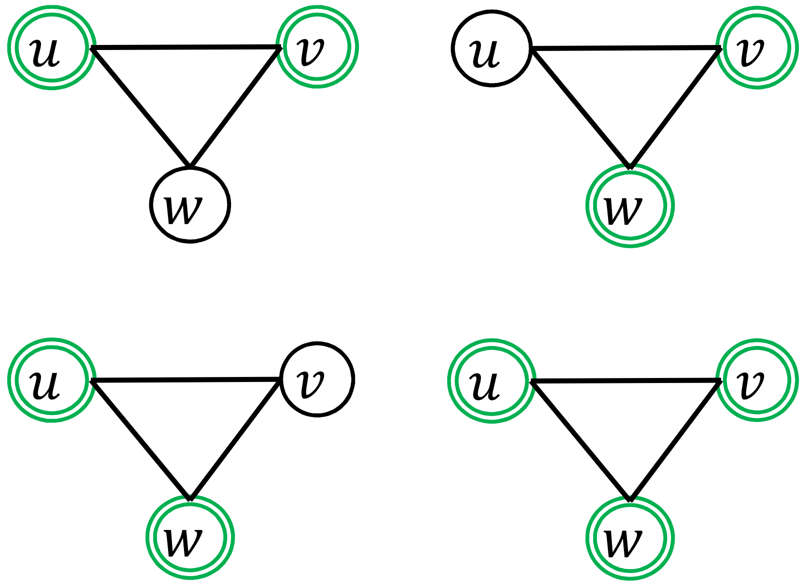


Figure 21: All the legal vertex covers of the triangle made up of u, v, w. The vertices in the vertex cover are marked with an additional circle.

original algorithm is $O(V \cdot d_{max}^2)$ where d_{max} is the size of the largest adjacency list in the graph. The key difference between our new algorithm and the original algorithm is that only a subset of vertices requires adjacency list intersection: $\hat{V}$. Further, only the adjacency lists of vertices that are within the vertex cover are intersected. As such the $O(d_{max}^2)$ is replaced with $O(\hat{d}_{max}^2)$, where $\hat{d}_{max}$ is the size of the largest adjacency list in the vertex cover.

Intuitively, the computational bottleneck of the original algorithm is the vertex with the largest adjacency list. This is because this vertex is intersected with all its neighbors d_{max} times and each intersection requires d_{max} operations. Conceptually this is still the same for the new algorithm, however, the bottleneck is no longer the highest degree vertex in the graph, but, rather the highest degree vertex in the vertex cover. This gives the new algorithm a time complexity of $O(Time_{VC} + \hat{V} \cdot \hat{d}_{max}^2)$, where $O(Time_{VC})$ refers to the time complexity of finding a vertex cover.

For the new algorithm to be considered optimal and not to add additional overhead to the existing complexity, the following requirements needs to be met: $O(Time_{VC}) \leq O(V \cdot d_{max}^2)$. For practical purposes, an even tighter bound is needed: $O(Time_{VC}) \leq O(V \cdot d_{max}^2 - \hat{V} \cdot \hat{d}_{max}^2)$. These requirements imply that the time spent computing the vertex cover should not be greater than the reduced run-time.

While a trivial vertex cover of $\hat{V} \equiv V$ can be found in $O(1)$, this is essentially the original algorithm. Linear time approximations yield reasonable smaller vertex covers than V in practice. For most linear-time vertex cover algorithms the condition is met almost trivially as $O(V + E)$ is smaller than $O(\hat{V} \cdot \hat{d}_{max}^2)$. This is not true for considerably sparse graphs, where $E \cong 2 \cdot V$, as very little work is required for computing the list intersections. We extend this discussion in Section 4.4.

4.3.3 Circuits of length 4 and higher

A circuit is defined as a simple path of vertices where each vertex and edge is traversed once with the first and last vertices in the path being the same. For example, a triangle is a 3-circuit. Abdo *et al.* [10] suggest using larger circuits than 3-circuits for the purpose of graph analysis. Specifically, they discuss the impact of "long-distance" relationships on the underlying graph structure.

A circuit of length four can be considered a *square* and a circuit of length five can be considered a *pentagon.* From a social network analysis standpoint, finding a square in the network given a specific player, v, is equivalent to finding common friends of any two of v's friends. This can be extended to find circuits of all sizes.

Fig. 22 depicts a subset of different vertex covers for a square. In all these vertex covers, u and x are part of the cover. All non-relevant edges have been removed. Note, in all the examples of Fig. 22 at least one set of vertices that are across from each other (i.e two-hops away) are in the vertex cover. This is mandatory for the vertex cover to be maintained. In our examples, these vertices are u and x. We refer to these vertex-pair as cross-vertices.

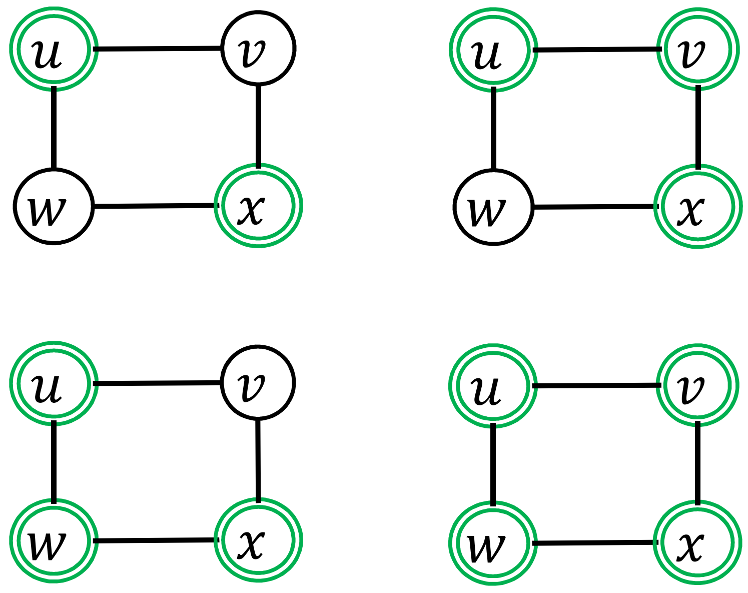


Figure 22: All legal vertex covers of the square made up of u, v, w, and x. Additional edges that are non-relevant have been removed. There are additional vertex covers for square. However, these follow one of the presented patterns. The vertices in the vertex cover are marked with an additional circle.

Finding circuits of length four entails taking all vertex pairs and intersecting their adjacencies. Given the adjacency set of two cross-vertices C, if $|C| > 2$ (meaning that the vertices share two or more neighbors) then a square exists. To be exact there are $|C| \cdot (|C| - 1)$ different 4-circuits. We will elaborate on this.

The pseudo-code for finding the number of 4-circuits using the vertex cover approach can be found in Alg. 4. The pseudo-code makes the proper counting correction. Using the cross-vertices observation, we reduce the computed intersection from "all vertex pairs" to "all vertex-cover pairs" without missing a 4-circuit. To get the original algorithm for 4-circuits, the pseudo code requires a simple modification: replace $\hat{V}$ with V in the first two loops and get rid of the counting correction code.

Given u, x are cross-vertices (not necessarily in the vertex cover), now consider the following scenarios when $w, v \in V$ and w, v are cross-vertices (both pairs must be

Algorithm 4: Pseudo-code for counting 4-circuits given a vertex cover $\hat{V}$.

```
$squares \leftarrow 0$;
for $u \in \hat{V}$ do
    for $x \in \hat{V}$ do
        if $u = x$ then
            next $x$;
        $C \leftarrow intersect(u, adj(u), x, adj(x))$;
        $In\hat{V} \leftarrow \{\}$;
        $notIn\hat{V} \leftarrow \{\}$;
        for $c \in C$ do
            if $c \in \hat{V}$ then
                $In\hat{V} \leftarrow In\hat{V} \cup \{c\}$;
            else
                $notIn\hat{V} \leftarrow notIn\hat{V} \cup \{c\}$;
            $squares \leftarrow squares + |C| \cdot (|C| - 1)$
            $|notIn\hat{V}| \cdot (|notIn\hat{V}| - 1)$ $+2 \cdot |notIn\hat{V}| \cdot (|In\hat{V}| - 1)$
```

cross vertices for a square to exist). The following scenarios occur:

1. $w, v \notin \hat{V}$. If they are part of the 4-circuit, then they can be found when intersecting the adjacency list of u, x and therefore this intersection can be avoided as it is redundant. If they are not part of a 4-circuit, but simply vertices that are two hops away, then there is no point intersecting their adjacencies allowing a reduced number of intersections.

2. Either $w \in \hat{V} \wedge v \notin \hat{V}$ or $v \in \hat{V} \wedge w \notin \hat{V}$. Without the loss of generality, assume the second case. Once again, for w, v to be part of a 4-circuit, $u, x \in \hat{V}$ cross vertices need to be adjacent to them. For the same reasons as in 1), redundant intersections can be avoided.

3. $w, v \in \hat{V}$. It is possible that they are part of a 4-circuit. If $u, x \in \hat{V}$ are cross vertices, then they are part of a 4-circuit. Therefore, their adjacencies need to be intersected. As such, the 4-circuit will be found for the intersection of w with v, v with w, u with x, and x with u.

   If $u, x \in \hat{V}$, it is possible to swap roles between w, v and u, x and go back to 1)

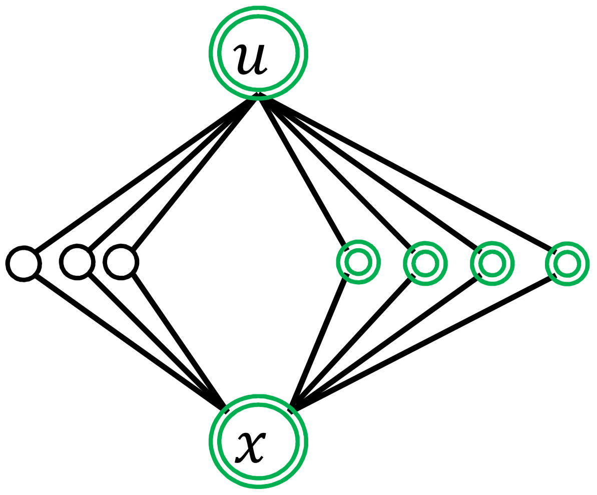


Figure 23: The cross-vertices (u, x) and their common neighboring vertices found in adjacency intersection. These neighbors are divided into groups: those in the vertex cover and those not in the vertex cover. The vertices in the vertex cover are marked with an additional circle.

which states that the only intersections that need to be computed are between the vertices in the cover.

The above essentially states the counting corrections that need to be made.

Assume that $u, x \in \hat{V}$, meaning that they are potentially cross-vertices and need to have their lists intersected. In the process of intersecting their adjacencies lists, the set C consists of two subsets, the first set consists of vertices in the vertex cover and is denoted $in\hat{V}$, and the second set consists of vertices that are not in the vertex cover, denoted as $notIn\hat{V}$. Fig. 23 depicts the intersections of two vertices u and x. The vertices belonging to $in\hat{V}$ have been marked with a double circle.

The number of ordered pairs in the intersection is $|C| \cdot (|C| - 1)$, each one of them specifies a different 4-circuit. Remember, that u will be intersected with x and x will be intersected with u and these give different 4-circuits [1].

Because not all vertex pairs have their lists intersected, a counting correction needs

[1]This can be avoided if lexicographical sorting is used.

to be taken into account. Consider v, w common neighbors of the ordered pair (u, x). If $v, w \in \hat{V}$, then the ordered pair (v, w) will have its neighbors intersected and then u, x will be found. Therefore, no correction needs to be done.

If $v, w \notin \hat{V}$, then the ordered pair (v, w) will not have its adjacencies intersected, therefore, requiring a counting correction as if the adjacencies of the ordered pair (v, w) were intersected and found u, x as its common neighbors. This adds two additional 4-circuits because of the ordering of the neighbors. The ordered pair (w, v) also will not have its neighbors intersected; however, this counting correction is taken into account by the ordered pair (x, u). Therefore, for the 4-circuit consisting of vertices u, v, w, x all eight 4-circles are accounted for. The counting correction needed because of vertices that are not in the vertex cover is $|notIn\hat{V}| \cdot (|notIn\hat{V}| - 1)$.

For the case in which only one of w, v is in the vertex cover, the explanation from above repeats itself. Therefore, the number of cycles that need to be accounted for is essentially the number of ways to order w, v: $2 \cdot |in\hat{V}| \cdot (|notIn\hat{V}| - 1)$.

In this subsection we showed how to find 4-circuits using the vertex cover and applying additional counting correcting techniques. These techniques can be further extended to include the general K-circuit case for $K \geq 3$. However, the general K-circuit case is outside the scope of this paper.

4.3.4 Clustering Coefficients in Dynamic Graphs

In this subsection we show that our new vertex cover clustering coefficients approach can also be applied to dynamic graphs. The two types of operations that we consider for dynamic graphs are edge insertions and edge deletions. Vertex insertion and deletion are simple. A vertex insertion places a new vertex in the graph without any edges. A vertex deletion can be serialized as a set of edge deletions.

For an inserted edge, $e = (u, v)$, into the graph the following three scenarios can arise:

1. $u, v \notin \hat{V}$

2. $(u \in \hat{V} \wedge v \notin \hat{V})$ or $(v \in \hat{V} \wedge u \notin \hat{V})$

3. $u, v \in \hat{V}$

The global and local clustering coefficients are handled slightly differently. Again, we focus on the global case.

For the first case, where both vertices are not in the vertex cover, one of the vertices has to be added to the vertex cover to ensure that all edges are covered. For simplicity assume that v is added to the vertex cover. The inserted edge can potentially close several triangles. The second case is similar to the first but does not require modifying the vertex cover. Obviously, for the third case when both vertices in the vertex cover, the vertex cover does not require any modification.

Edge deletions can be dealt with in a similar fashion. However, when deleting $e = (u, v)$, the first scenario in which $u, v \notin \hat{V}$ is not possible as this violates the vertex cover's properties.

4.4 Empirical Results

In this section we present empirical performance results of the new clustering coefficients algorithm for both triangle counting and square counting. In our tests we used an Intel i7-2600K system with 16GB of memory. The clock frequency is 3.4GHz. We used graphs taken from the 10th DIMACS Implementation Challenge on Graph Partitioning and Graph Clustering [19]. The graphs we used can be found in Table 8. The global clustering coefficient value for these graphs are presented in Fig. 24.

For benchmarking purposes, we used the clustering coefficient algorithm from [53]. This is a highly optimized CSR implementation of clustering coefficients. As such we have used this CSR clustering coefficient implementation as our baseline and have implemented our new algorithms based using the list intersection taken from the

Table 8: Graphs from the 10th DIMACS Implementation Challenge that are used in our experiments, grouped by type and sorted within each group by vertex count.

Name	$\|V\|$	$\|E\|$
Collaboration Networks		
coAuthorsCiteseer	227,320	814,134
citationCiteseer	268,495	1,156,647
coAuthorsDBLP	299,067	977,676
coPapersCiteseer	434,102	16,036,720
coPapersDBLP	540,486	15,245,729
Random Networks		
RMAT-16	65,536	2,456,071
RMAT-18	262,144	10,582,686
RMAT-20	1,048,576	44,619,402
matrix		
audikw1	943,695	38,354,076
ldoor	952,203	22,785,136
cage15	5,154,859	47,022,346
Road Networks		
luxembourg.osm	114,599	119,666
belgium.osm	1,441,295	1,549,970
road_central	14,081,816	16,933,413
road_usa	23,947,347	28,854,312
clustering		
celegans_metabolic	453	2025
email	1,133	5,451
polblogs	1,490	16,715
netscience	1,589	2,742
power	4,941	6,594
hep-th	8,361	15,751
PGPgiantcompo	10,680	24,316
astro-ph	16,706	121,251
cond-mat	16,726	47,594
as-22july06	22,963	48,436
cond-mat-2003	31,163	120,029
cond-mat-2005	40,421	175,691
smallworld	100,000	499,998
preferentialAttachment	100,000	499,985
caidaRouterLevel	192,244	609,066

implementation.

We use is a simple linear time greedy algorithm, $O(V + E)$ for finding the vertex cover. For each vertex v in the graph, we check if v has any neighbors that are not within in the vertex cover. If this is the case, v is added to the vertex cover. This algorithm by no means ensures a small or minimal vertex cover. Nonetheless, for our

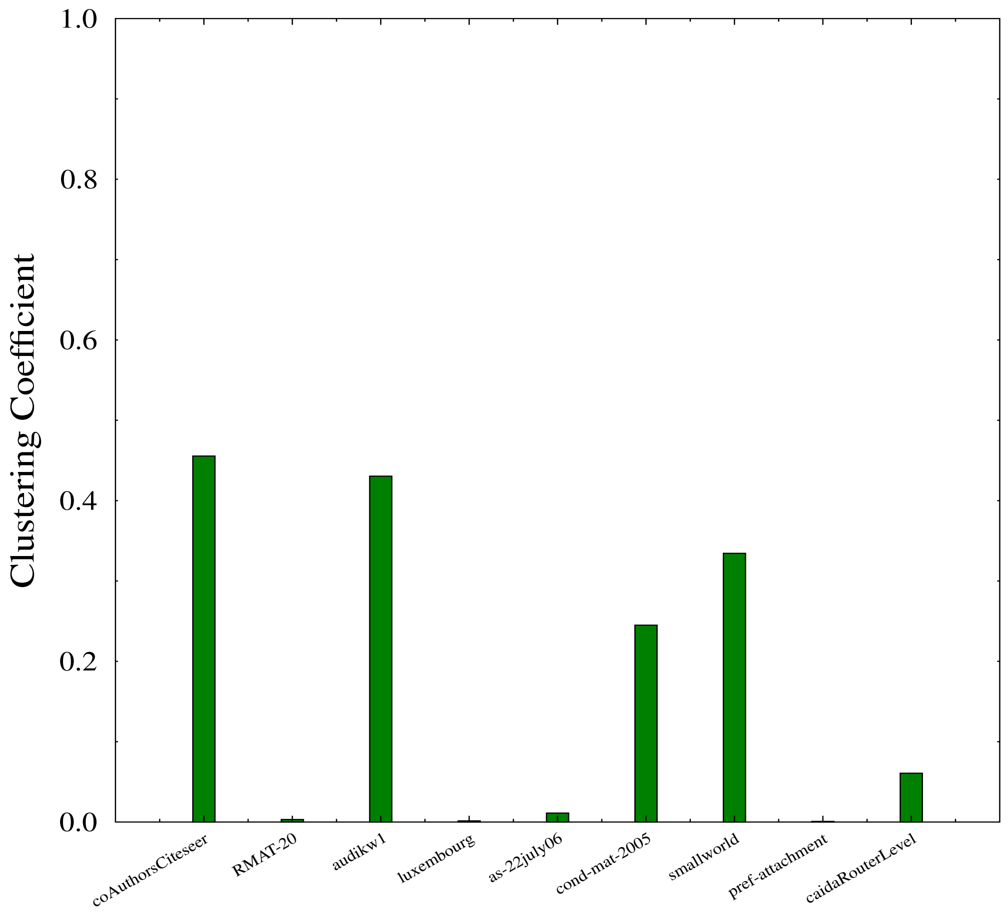


Figure 24: This chart shows the global clustering coefficient value for graphs from Table 8.

needs it is sufficient as it gave small enough vertex covers for us to see improvements.

4.4.1 Static Clustering Coefficients

The overhead to compute a reasonable vertex cover is negligible for many graphs. Fig. 25 depicts the portion of time needed to compute the vertex cover out of the total needed for computing the clustering coefficients. Note that the ordinate of Fig. 25 is a log-scale which shows the time spent computing the vertex cover out of the total

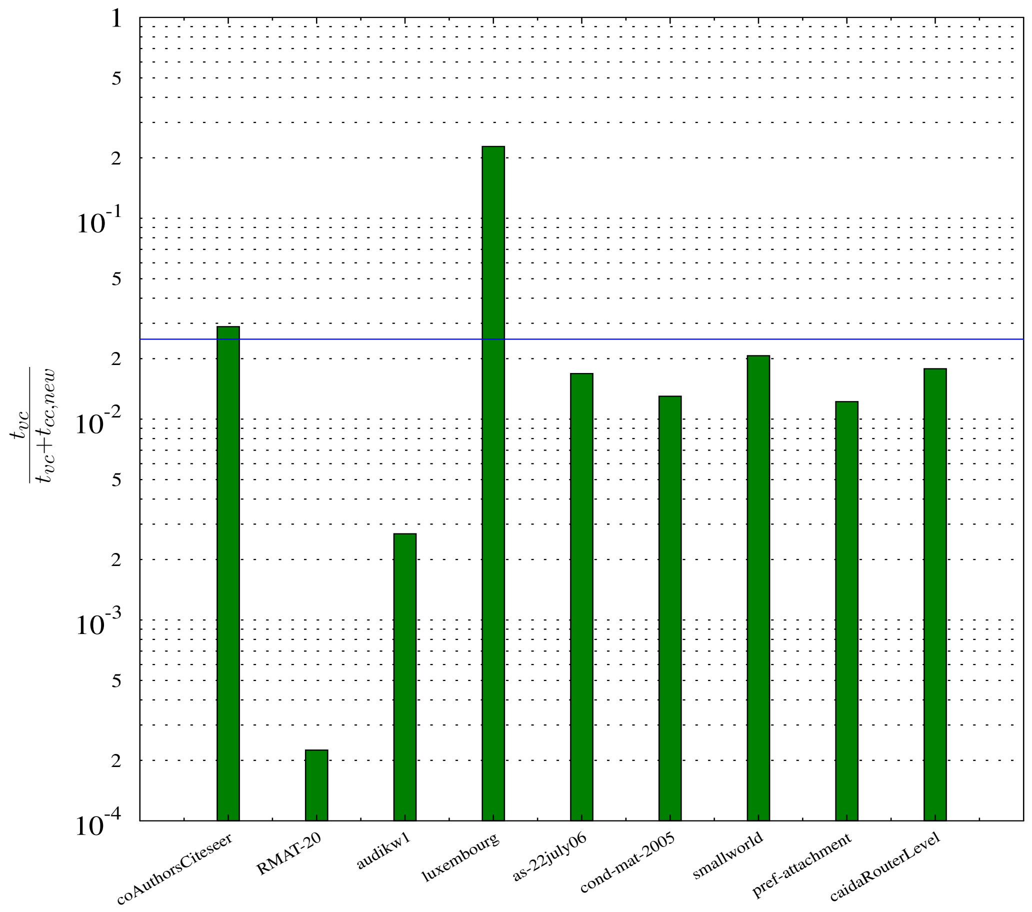


Figure 25: The ordinate presents the ratio of time spent finding the vertex cover as a function of the total time spent computing the vertex cover and the clustering coefficients. The ordinate is a logscale. An additional blue curve has been placed at 2.5%. The bars of about 2/3 of the graphs are below this curve.

time of the new algorithm (time for the vertex cover and the clustering coefficient algorithm that uses the vertex cover), and lower is better. In Fig. 25 an additional (constant) curve has been added at 2.5%. For many of the graphs, the vertex cover takes less than 2.5% of the total time to compute.

Fig. 26 depicts the size of the vertex cover in comparison with the entire vertex

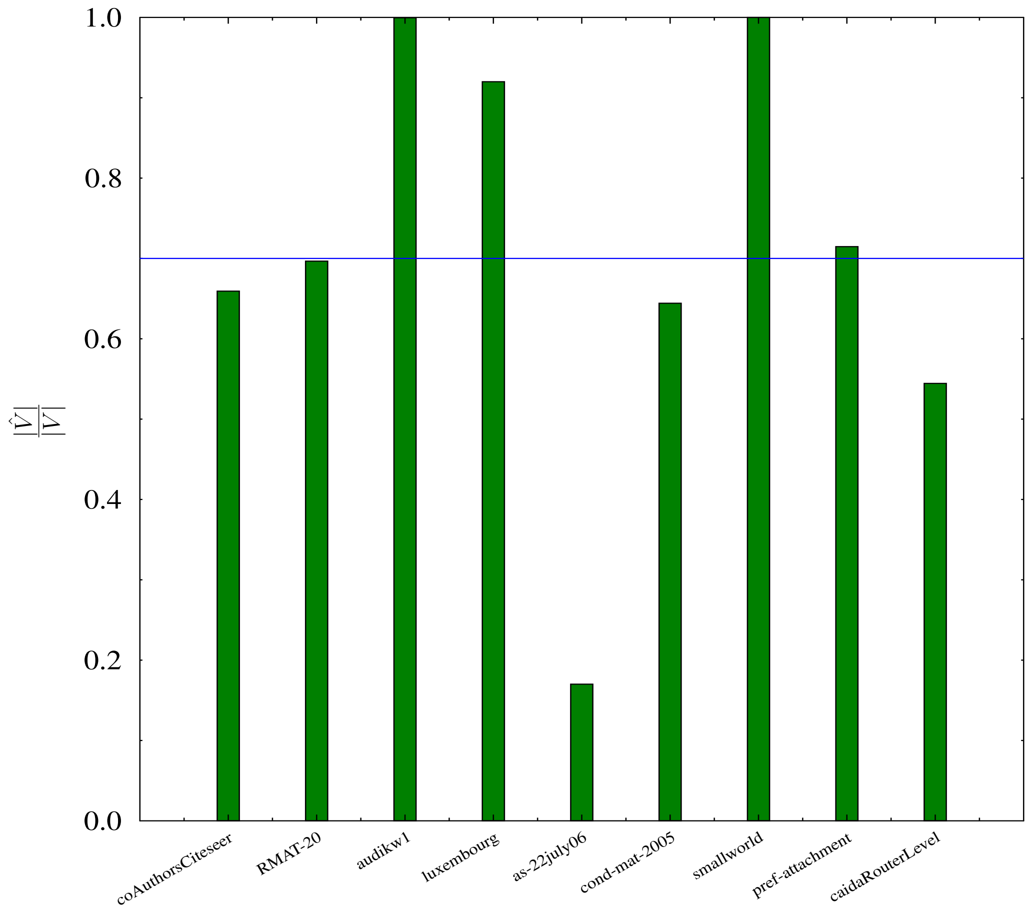


Figure 26: The ordinate is the size ratio of the vertex cover, $\hat{V}$, and the vertex set V. Note that $|\hat{V}| \leq |V|$. An additional blue curve has been added at 70%. The bars of about 3/5 of the graphs are below this curve.

set V. An additional constant curve has been added at 70%. For slightly more than half the graphs, the size of the vertex cover is less than 70% the size of the entire vertex set.

Fig. 27 depicts the ratio of the time it takes to compute the global clustering coefficients using our new method over the original algorithm. For this figure, lower is better as this means that the new algorithm takes a fraction of the original run-time.

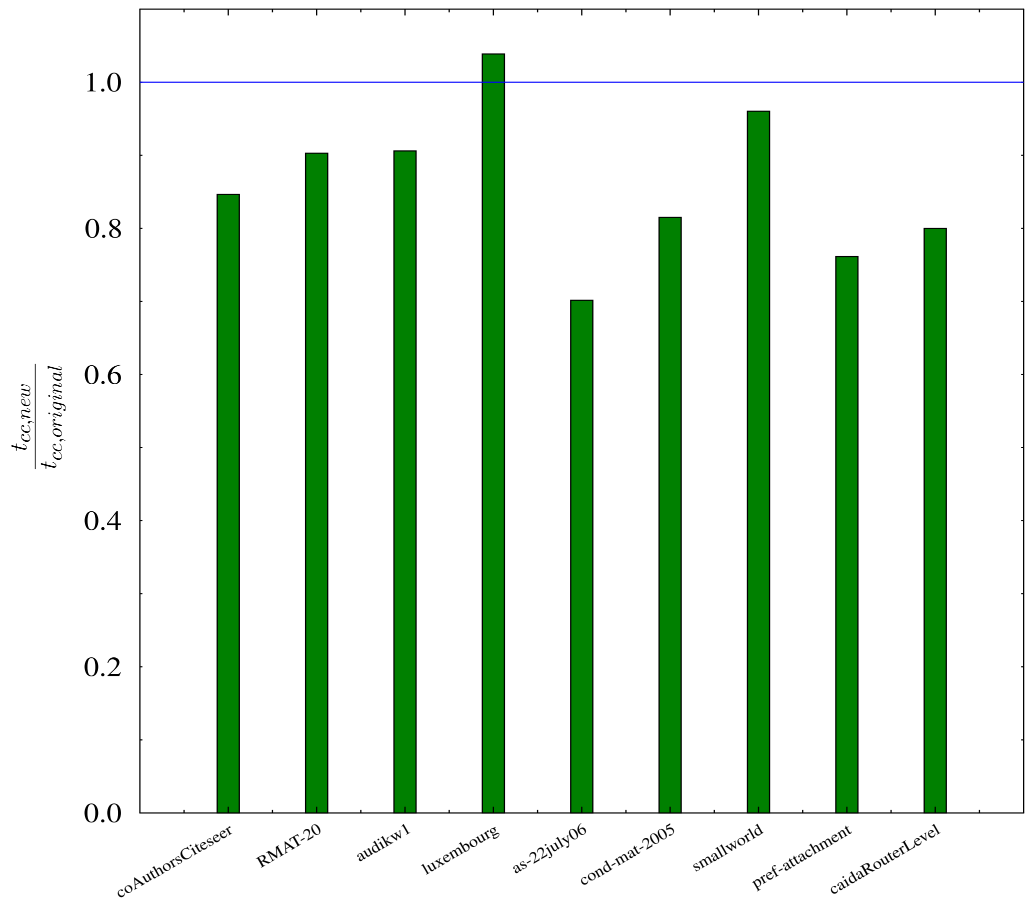


Figure 27: The ordinate is the time ratio of the new vertex cover-based clustering coefficients algorithm and the original clustering coefficient algorithm. The time of the new algorithm includes both the time needed to find the vertex cover and the time for computing the clustering coefficients. All bars below the blue curve occur when the new algorithm is faster, as such lower is better.

An additional blue curve has been added to the figure to state when the performance of the two algorithms is the same. There are several graphs that do not benefit from our new approach. These are graphs for which the vertex cover is roughly the size of $|V|$. A different vertex cover (by a different algorithm) might possibly reduce the size of the vertex cover and allow for shorter run times. However, this was not the focus

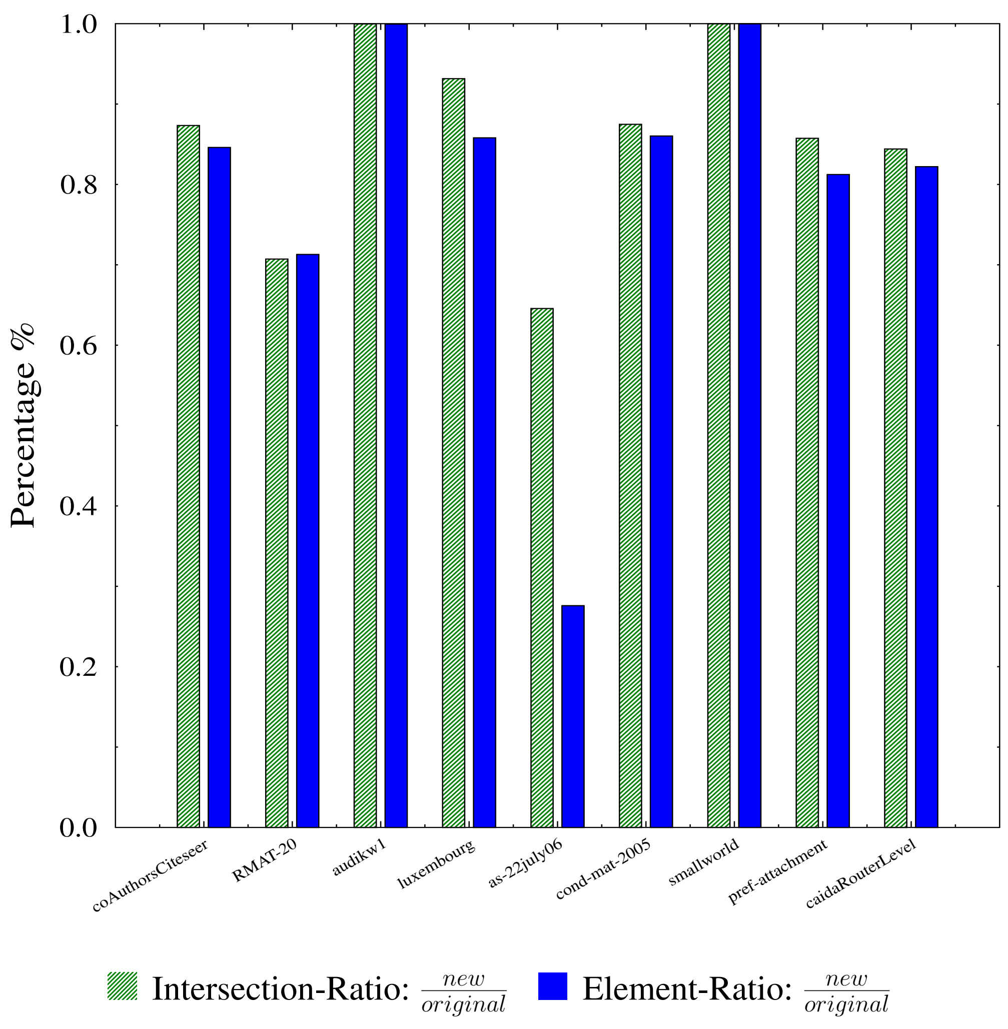


Figure 28: The ordinate is the ratio of the number of intersection that are necessary by the new algorithm in comparison with the original algorithm.

of our work.

For six graphs that we tested, the time for finding the vertex cover took over 10% of the total time. Out of these six graphs, four are road networks. The reason the vertex cover takes such a high percentage of the total time is due to the considerable sparsity of these graphs, where$|E| \leq 2 \cdot |V|$. These graphs have low clustering coefficients and the amount of intersections needed is considerably small. It is not surprising that for these graphs the total time of our new algorithm is greater than the time using the original approach as our vertex cover added overhead. Fortunately, this overhead is negligible.

In Fig. 28 two different bar charts are shown, both are ratio values from 0 to 1, such that the ratios are between our new algorithm and the original one. The green patterned bars are the ratio between the number of adjacency lists intersected using the new method and the respective number using the original method. The blue solid bars are the ratio between the actual number of elements intersected for both methods. These two ratios are slightly different. The first ratio states how many intersections need to be computed. The second ratio states how many comparisons need to be done as part of the list intersection. The second ratio takes into account that the lists are not equal length. This is expected in graphs following power-law distribution [22, 31, 63]. For graphs with a uniform distribution of edges, such as the Erdös-Rényi ([61, 62]) random graph model, it is very likely that these ratios would be similar. Further this non-uniformity plays a critical part of the time complexity of the algorithm, specifically d_{max}.

As expected, the graphs that had larger vertex covers (when $|\hat{V}| \cong |V|$) did not have a reduction in the number of intersections or the number of elements intersected. The graphs with the smaller vertex covers had fewer intersections, fewer comparisons, and reduced runtime.

As the number of intersections of our algorithm is bounded by the number of intersections of the original algorithm (which computes all the intersections) these ratios are smaller than one.

In the context of performance, the smaller these ratios are the better the performance will be as these ratios are correlated to the number of list intersections that need to be done. As such, the lower the ratios, the better.

4.4.2 Static 4-Circuits

In this subsection we present performance results for finding 4-circuits using the algorithm presented in Section 4.3.3. A key challenge of the 4-circuit counting is the increased time complexity of the algorithm. As such, for the 4-circuits counting we present results for the smaller graphs that belong to the clustering subset of graphs taken from Table 8. We note that finding the 4-circuits for larger graphs can be parallelized to further reduce the running time.

Fig. 29 depicts the time spent computing the vertex cover vs. the time to compute the vertex cover and find the 4-circuits. For almost all the graphs, the time to compute the vertex cover is less that 0.1%. For several graphs the vertex covers takes less than 0.001% of the total time. We use the same vertex cover algorithm as before and the running times are the same as before. As the circuits get longer the time spent on finding the vertex cover becomes a smaller fraction of the total running time.

Fig. 30 depicts the time ratio between the new algorithm (including time spent on finding the vertex cover) and the original algorithm for computing 4-circuits. The blue curve at $y = 1$ denotes when the execution time of both algorithms is the same. Once again, any bar below the blue curve means that the new algorithm is faster than the original algorithm. For half the graphs the new algorithm is 30% faster than the previous algorithm which is due to fewer intersections. Fig. 31 depicts the ratio of the number of comparisons made during the list intersections for the new and original algorithms.

4.5 Conclusions

In summary, in this chapter we design and implement a new method for computing exact clustering coefficients using vertex covers. This method reduces the number of list intersections and avoids unnecessary element comparisons. The two key differences between our new algorithm and the original approach are as follows: 1) our

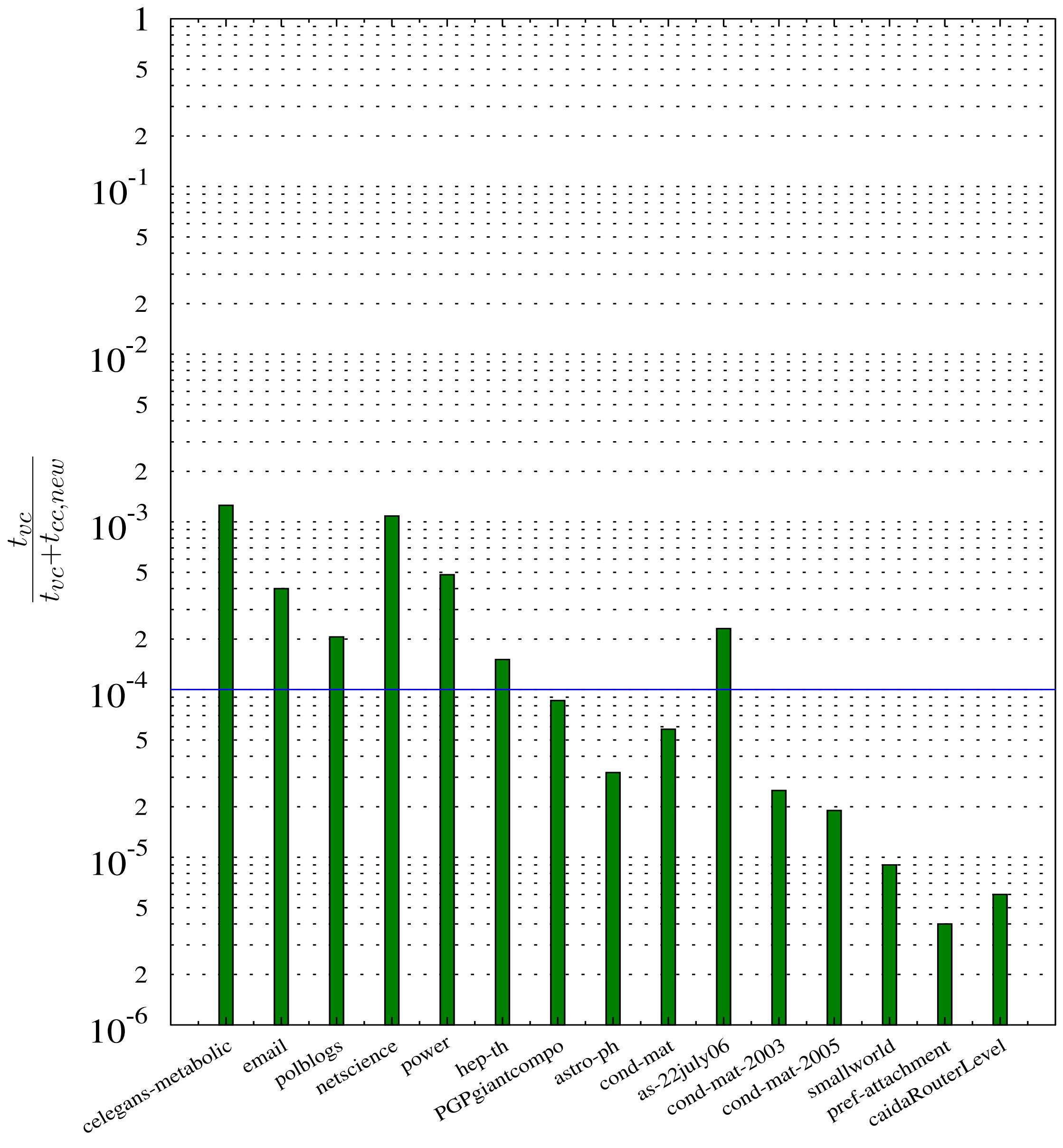


Figure 29: The ordinate presents the ratio of time spent finding the vertex cover as a function of the total time spent finding all the 4-circuits. Note that the ordinate is a log scale. The abscissa are the Clustering graphs from Table 8.

algorithm computes a vertex cover of the graph and 2) our algorithm applies a counting correcting technique that makes up for triangles that are not counted multiple times.

We show that the new algorithm is both exact (gives the same results as the original algorithm) and correct (by proofs). We then show that our new approach can also be used on circuits of length four and can be extended to longer circuits. All these are followed by performance analysis in which we showed that the new algorithm is indeed faster, $15\% - 20\%$ for the 3-circuit and $30\% - 40\%$ for the 4-circuit, than the previous algorithms.

While the focus of work was to show the viability of the vertex cover for clustering coefficients, we believe that the vertex cover approach might be applicable to additional social network analytics, perhaps community detection and modularity-based algorithms. Several open problems related to this work are:

1. Finding additional analytics that can benefit from the vertex cover.

2. Testing the sensitivity of the algorithm by changing the vertex cover algorithm. This includes using additional vertex cover algorithms or using the same vertex cover algorithm as we did that accesses the vertices in a different order (this should give a different cover).

3. Similar to the previous issue, is it possible to select a vertex cover algorithm based on properties of the input graph.

4. For the dynamic graph case, how does the vertex cover change over time? Does the vertex cover gradually increase in size over time until its the size of the vertex set or is the vertex cover near constant over the insertion?

5. Creating an efficient algorithm for finding a K-circuit for some given K using the vertex cover approach.

6. Is there an effective way to parallelize the new algorithm?

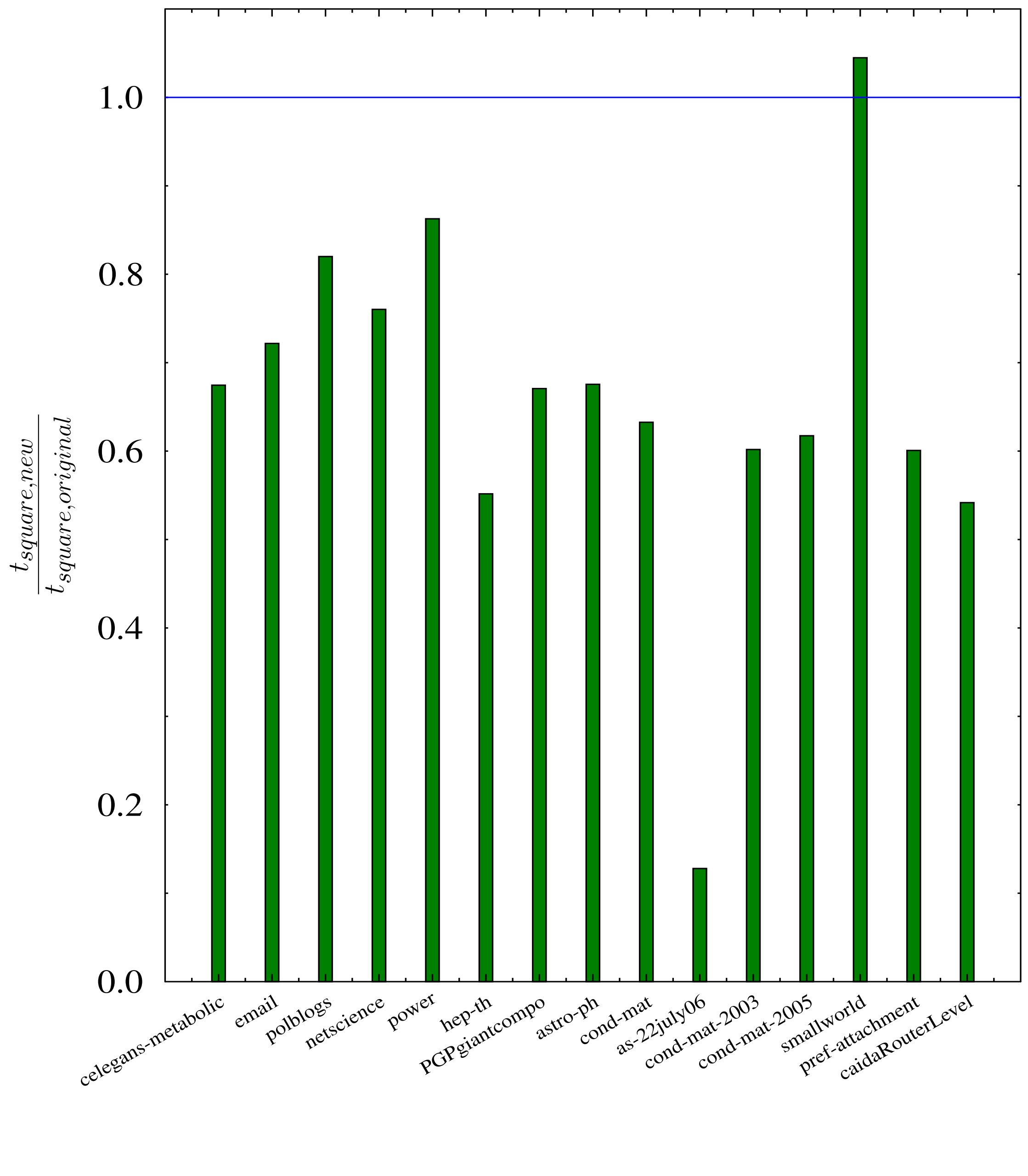


Figure 30: The ordinate is the time ratio of the new vertex cover based clustering coefficients algorithm and the original clustering coefficient algorithm. The time of the new algorithm includes both the time needed to find the vertex cover and the time for computing the clustering coefficients. The blue curve denotes equal run-times for the new and original algorithm. All bars below the blue curve state the new algorithm is faster, as such lower is better.

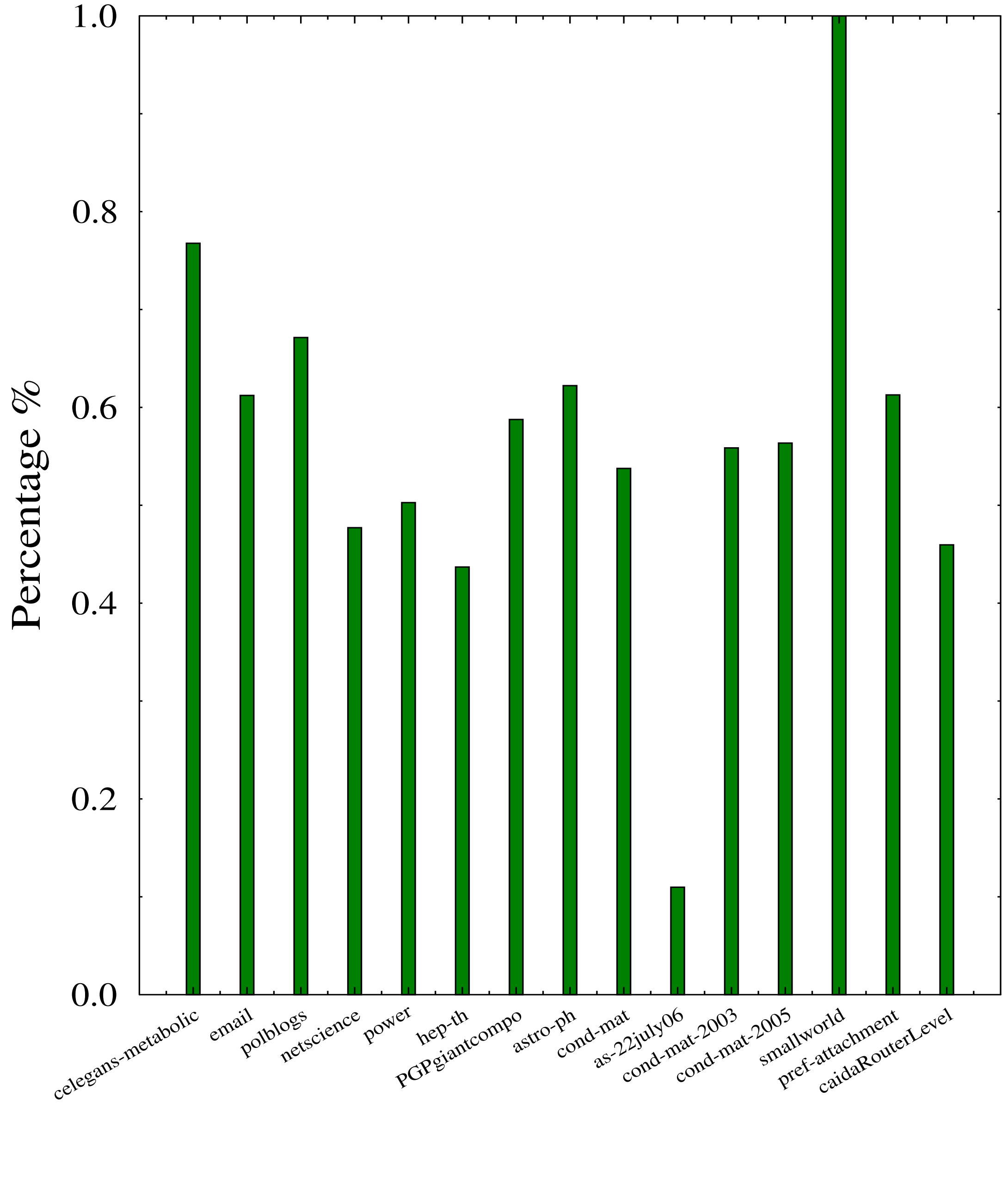


Figure 31: The ordinate is the ratio of the number of intersection that are necessary by the new algorithm in comparison with the original algorithm.

CHAPTER V

LOAD BALANCED CLUSTERING COEFFICIENTS

This chapter is an extension of the paper: O. Green, L.M. Munguia, D. Bader, "Load Balanced Clustering Coefficients", ACM Workshop on Parallel Programming for Analytics Applications (PPAA), PPoPP, Orlando, Florida, 2014.

Clustering coefficients is a building block in network sciences that offers insights on how tightly bound vertices are in a network. Effective and scalable parallelization of clustering coefficients requires load balancing amongst the cores. This property is not easy to achieve since many real world networks are scale free, which leads to some vertices requiring more attention than others. In this work we show two scalable approaches that load balance clustering coefficients. The first method achieves optimal load balancing with an $O(|E|)$ storage requirement. The second method has a lower storage requirement of $O(|V|)$ at the cost of some imbalance. While both methods have similar a time complexity, they represent a tradeoff between load-balance accuracy and memory complexity. Using a 40-core system we show that our load balancing techniques outperform the widely used and simple parallel approach by a factor of $3X - 7.5X$ for real graphs and $1.5X - 4X$ for random graphs. Further, we achieve $25X - 35X$ speedup over the sequential algorithm for most of the graphs.

5.1 Introduction

Clustering coefficients is a graph analytic that states how tightly bound vertices are in a graph [138]. The tightness is measured by computing the number of closed triangles in the graph, which can then imply the small-world property. Computing the clustering coefficients has been applied to many types of networks: communication [125], collaboration [133], social [133], and biological [23]. Clustering coefficients is used in a wide range of social network analysis applications. In such context, one can think of the local clustering coefficients as the ratio of actual mutual acquaintances versus all possible mutual acquaintances.

Clustering coefficients can be computed in two different variants: global and local. The global clustering coefficient is a single value computed for the entire graph, whereas the local clustering coefficient is computed per vertex. Both are computed in a similar fashion. Without the loss of generality, we consider the global clustering coefficient in this work, specifically when presenting pseudo code. Nonetheless, our approach is applicable to computing local clustering coefficients as well. Table 9 presents the notations used in this chapter.

We can formally define clustering coefficients as the sum of the ratios of the number of triangles over all possible triangles:

Clustering coefficients can be computed in multiple approaches [123]: enumerating over all node-triples, matrix multiplication, and intersecting adjacency lists. As many real world networks are considerably sparse, we focus on the last of these three approaches which has a time complexity of $O(|V| \cdot d_{max}^2)$ where d_{max} is the vertex with largest adjacency. The pseudo code for this approach can be found in Algorithm 5. Many real world networks have a skewed vertex degree distribution which present parallel load balancing challenges.

In this work, we show that it is possible to estimate the total amount of work required by the clustering coefficient algorithm in $O(|E|)$ steps.

We show two different and simple load balancing techniques: the edge-based approach and the vertex-based approach. These differ in the fact that the edge-based approach offers a better workload balance than the vertex-based approach. While this advantage is desirable, it comes at an increased spatial and computational cost. The edge-based approach requires an $O(|E|)$ memory and $O(|E|)$ operations that evenly split the work to the p processors. On the other hand, the vertex-based approach

Algorithm 5: Serial algorithm for computing the number of triangles.

```
$cc\_count \leftarrow 0$;
for $v \in V$ do
    for $u \in adj(v)$ do
        if $u = v$ then
            next $u$;
        $C \leftarrow intersect(v, adj(c), u, adj(v))$;
        $cc\_count \leftarrow cc\_count + \frac{|C|}{deg(v) \cdot (deg(v)-1)}$;
```

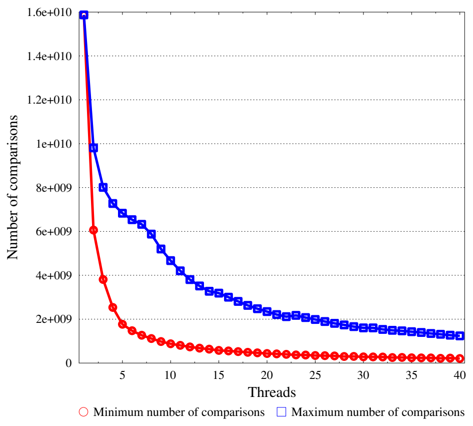
requires an $O(|V|)$ memory and $O(|E|)$ operations, which also are split among the processors. Both approaches can be executed in parallel.

While the load balancing may seem costly, for sparse graphs where $O(|E|) < O(|V| \cdot d_{max}^2)$. We show in Section 4.4 that this computation is negligible in time on an actual system for many real world sparse graphs.

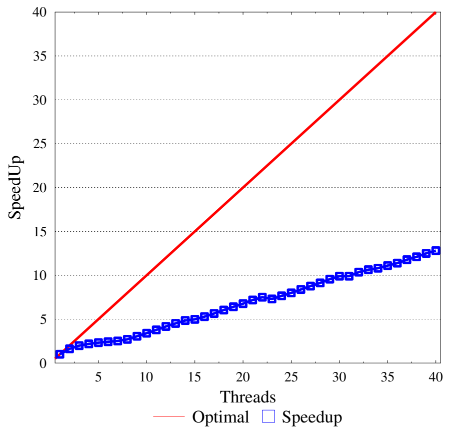
The remainder of the chapter will be structured as follows. This section discusses the challenges with computing clustering coefficients in parallel and briefly introduces our solutions. Section II discusses the related work and discusses real world graph properties and introduces vertex covers. In Section III, we present our two approaches for effective load balancing. In Section IV, we discuss our experimental methodology and present quantitative results. Finally, in Section V, we present our conclusions.

5.1.1 Parallel Clustering Coefficient Challenges & and Solutions

In [22, 31, 63], it was shown that several real world networks follow a power law distribution on the number of adjacent edges a vertex has. As the time complexity of the algorithm is dependent square of the vertex with the highest degree, d_{max}, a simple division of the vertices amongst the cores such that each core receives an equal number of vertices is not likely to offer a good load balancing. This is due to the fact that a single core might receive multiple high degree vertices. This can cause a single core to become the execution bottleneck. The focus of our work is to overcome this challenge.



(a)



(b)

Figure 32: (a) Load distribution among parallel processors. (b) Shows the maximum attainable speedup.

Fig. 32 depicts an example of the load balancing issues caused by straightforward division schemes when computing clustering coefficients for a graph with a non-uniform adjacency distribution. Fig. 32 (a) plots the minimum and the maximum number of comparisons performed by the different parallel processors. Assuming that the total amount of work required by the algorithm is known and is defined as $Work$, the maximal parallel speedup that can be attained for each algorithm is limited by the thread with maximum number of comparisons, max_t:

$$max_{speedup} = Work/max_t. \tag{9}$$

By this definition, uneven work distributions can affect scalability severely. The main ordinate of Fig. 32(b) depicts the ratio between the minimum and the maximum number of comparisons shown in Fig. 32(a). The secondary ordinate of Fig. 32(b) shows the maximal attainable speedup of the work distribution of the algorithm used to plot Fig. 32(a). In fact, it is the observation from above, that motivated the development of load balancing techniques that take into account the unique workload properties of clustering coefficients.

Table 9: Notations in this chapter

Name	Description
CC_{global}	Global clustering coefficient
CC_v	Clustering coefficient for vertex v
$deg(v)$	Degree of vertex v.
$tri(v)$	Number of triangle that vertex v is in.
d_{max}	Vertex with maximal degree in the graph.
P and p	Number of parallel processors.
V	Set of vertices in a graph.
E	Set of edges in a graph.
u, v	Vertices in the graph.
t	Thread number.
$Vertex\_adj$	Array of size $\|V\|$ containing the starting positions of the vertex adjacencies
$eWork$	Array of size $\|E\|$ used to accumulate the work estimation of every connected vertex pair.
$vWork$	Array of size $\|V\|$ used to accumulate the work estimation of every vertex.
$Pivots$	Array of size $P + 1$ that holds the starting and ending points of the work done by each processor.
t	Thread id.
cc_t	Local thread clustering coefficients value.
cc_{global}	Global clustering coefficients value.

5.2 Related work

Clustering coefficients was first introduced by Watts and Strogatz [138]. Since, it has become a common means to quantify structural network properties. In essence, clustering coefficients measures the tightness of neighborhoods in graphs. They can be computed for both dense and sparse graphs, yet, they offer more insights for sparse graphs, many of which have the small-world property. The Small-World property was first presented by Milgram [109] and suggested that people in the United States can be related in less than six steps of separation.

An additional graph property of significance is the power-law distribution of edges in a network. Graphs featuring this characteristic have a large number of vertices with low degrees and a small number of vertices with high degrees, see by Faloutsos *et. al.* [63] and Barabási and Albert [22].

Clustering coefficients for a given graph is often reduced to enumerating the triangles formed between every triplet of vertices. Schank and Wagner [123] present an extensive review on other various serial algorithms along with performance comparisons over both "real world" networks and synthetic graphs. Still in the context of serial algorithms, Green and Bader [75] propose a novel clustering coefficients algorithm that employs vertex covers in order to reduce the number of list intersections and the number of actual comparisons needed to compute the triangle enumeration. We show that our load-balancing techniques can be extended to this algorithm as well.

The increase in the network size from thousands of vertices to millions and possible billions in the foreseeable future and the relevance of dynamic graphs has brought about a need for effective computations of clustering coefficients. The advances for faster clustering coefficients algorithms focus in parallelization techniques as well as approximation schemes. Both improvements are orthogonal concepts: while parallelization reduces the computation time, approximation can offer insights on the closeness of vertices when the cost of computing the clustering coefficients is prohibitive. Special techniques can also be developed for dynamic graphs. Dynamic graph algorithms allow updating the analytic without doing a full recomputation every time the underlying network is modified. In practice, these optimizations are applied concurrently.

Several examples of algorithms using these optimizations can be found in the works of Becchetti *et. al.* [27], Bar-Yossef *et. al.* [21], and Buriol *et. al.* [35] where triangle counting approximation techniques are employed on streaming graphs as a measure to cope with large data sets.

Ediger *et. al.* [53] present both a parallel exact and parallel approximate algorithm for computing clustering coefficients for dynamic graphs. Their approach employs Bloom filters and they show results on the Cray XMT (a massively multithreaded architecture) for graphs with over a half a billion edges. Leist *et. al.* [98] provide multiple parallel implementations using several GPUs and IBM Cell-BE processors for smaller graphs.

While these algorithms tackled many of the different aspects of computing clustering coefficients, they do not deal with the inherent load imbalance that is typical for many graph algorithms using static scheduling techniques. In such scenario, the work is partitioned by the runtime and it is unaware of the properties of the application. As such, the application designer is responsible for dividing the work equally to the cores.

Both [24] and [108] consider the problem of workload imbalance for other graph algorithms. They use similar techniques to the ones that we use in this chapter, which are based on prefix summation. Prefix summation is a basic primitive that can also be efficiently parallelized. Blelloch [28] showed a PRAM parallel work-efficient algorithm. In [78] a GPU implementation of the prefix summation is introduced.

In [24], a Breadth First Search algorithm is presented. The vertices in each level are partitioned to the multiple cores based on the sum of the adjacent vertices. A prefix summation is employed followed by a binary search in order to elaborate the partitioning. The overhead of these two partitioning stages is negligible compared to the remainder of the BFS. In [108], scalable GPU graph traversals are presented that partition the traversal edges equally among the multiple gpu streaming processors.

Other load balancing mechanisms can be used on specialized architectures. Ediger *et. al.* [57] made use of the online scheduler of the massively multithreaded architecture of the Cray XMT for computing clustering coefficients. The scheduler is responsible for dispatching tasks when processors become available, thus achieving an effective parallelism. They show nearly perfect load balancing upto 64 XMT processors for several different graph types. Beyond the 64 processors, the speedup continues to grow but does not always scale perfectly - this is most likely due to workload imbalance.

5.3 Load Balanced Scalable Clustering Coefficients

We present two different techniques, which are conceptually similar and consist of two highly parallel phases. The first phase approximates the the expected amount of work. With the work estimation at hand, we proceed to partition the work to the cores. We proceed to show that each of these steps is itself balanced. In the second step, the adjacency lists are intersected by the multiple cores using a modified version of Algorithm 5.

The workload estimation process is based off of Algorithm 5 and consists of foreseeing the amount of work needed to intersect vertices and edge endpoints. For that purpose, we employ two basic work estimations, which are discussed in depth. Both estimations can be defined in terms of the amount work needed for intersecting the adjacency lists of two vertices $u, v \in V$.

For simplicity, we assume that the adjacency lists are sorted. Given the sorted lists, the upper bound for the adjacency list intersection is $deg(u) + deg(v)$ comparisons. This number represents the worst-case scenario for a adjacency list intersection. An actual list intersection might be cut short if all the elements of one of the lists are traversed. However, this cannot be detected without doing the intersection or further testing. Nonetheless, we find $deg(u) + deg(v)$ to be a fair estimation of the adjacency list intersection.

We can define $Work(v, u)$ as the number of comparisons needed for the intersection of the adjacencies of v and u:

$$Work(v, u) = deg(v) + deg(u). \tag{10}$$

We proceed to gather this estimation for the vertex endpoints of every edge in the graph. This is formalized in terms of the previous definition as:

$$Work(G(V, E)) = \sum_{(u,v)\in E} Work(u, v) = \sum_{(u,v)\in E} (deg(v) + deg(u)) \tag{11}$$

Such definition can also be expressed in terms of vertices and their adjacencies:

$$Work = \sum_{(u,v)\in E} Work(u,v) = \sum_{v\in V}\sum_{u\in adj(v)} deg(v) + deg(u). \tag{12}$$

Specifically, the number of comparisons required for a specific vertex is:

$$Work(v) = \sum_{u\in adj(v)} (deg(v) + deg(u)) =$$

$$deg(v)^2 + \sum_{u\in adj(v)} deg(u). \tag{13}$$

It is from (13) that the time complexity of clustering coefficients is in fact derived.

Computing either $\Sigma_{(v,u)\in E} Work(v,u)$ or $\Sigma_{v\in V} Work(v)$ allows us to to partition the work into near equal units to the multiple cores available. The first technique, which we refer to as the the edge-based approach, requires $O(|E|)$ memory and theoretically offers optimal partitioning assuming that all comparisons are executed[1]. The vertex-based approach reduces the memory requirement to $O(|V|)$ at the expense of non equal partitioning. In Section 4.4 we quantitatively compare these two approaches for real networks. While we have yet to discuss the algorithms in detail, we note that both algorithms have a similar upper bound time complexity, yet the accuracy of the partitioning will slightly change based on the storage complexity.

5.3.1 Edge-Based Approach

We present the first of our two approaches that partitions the work equally among the multiple cores. We refer to this method as the edge-based approach and its pseudo code can be found in Algorithm 6. In Table 9, a reference is given for the variables used by our methods. Overall, we distinguish two distinct computation phases: 1) work estimation and load balancing and 2) clustering coefficient computation.

Using an array of size $O(|E|)$ the expected number of comparisons for each edge is calculated based on expression (10). The first step in Stage 1 divides the edges

[1]As we discussed earlier, the list intersection might complete early based on the actual adjacencies.

Algorithm 6: Edge-Based algorithm

```
input  : Graph $G(V, E)$, number of processors $p$
output: Clustering coefficients value $cc_{global}$
For $t \leftarrow 1$ to $p$ do in parallel
    // Stage 1: Workload estimation
    $Pivots_t \leftarrow$ BinarySearch($Vertex\_adj, t \cdot |E|/p$);
    SynchronizationBarrier();
    for $v \leftarrow Pivots_t$ to $Pivots_{t+1}$ do
        for $\forall u \in adj(v)$ do
            $eWork_{(v,u)} \leftarrow deg(v) + deg(u)$;
    SynchronizationBarrier();
    ParallelPrefixScan($eWork$);
    $Pivots_t \leftarrow$ BinarySearch($eWork, t \cdot |E|/p$);
    SynchronizationBarrier();
    // Stage 2: CC calculation
    $ccl \leftarrow 0$;
    $E_t \leftarrow$ all edges between $Pivots_t$ and $Pivots_{t+1}$
    for $e = (v, u) \in E_t$ do
        VertexIntersection($v, deg(v), u, deg(u)$);
        $cc_t \leftarrow cc_t + \frac{|C|}{deg(v) \cdot (deg(v)-1)}$;
    $cc_{global} \leftarrow$ ParallelReduction($cc\_t$);
```

equally among the p cores such that each core receives $|E|/p$ edges. Assuming that the graph is given in a CSR representation, a binary search is conducted by each core into the vertex offset array. Such a vertex offset array is essentially a prefix sum array of the edge degrees in the graph. Hence, the binary search in the vertex offset array for the value $t \cdot |E|/p$ allows finding the vertex to which that that edge belongs to, for a given thread $t \in \{1, 2, ..., p\}$.

Due to the fine grained load balancing, several cores may intersect adjacency lists for the same vertex[2]. For simplicity, we assume that the sets of adjacencies assigned to the different processors do not overlap - this is a fair assumption given adjacency distributions of many real world graphs. However, if there is a vertex with a large enough degree that it dominates the execution time, it is possible to modify the algorithm such that the adjacencies of a vertex is shared by multiple cores.

Each core will compute $Work(v, u)$ for the set of edges it has been assigned. The results will be stored in the $Work$ array of size $O(|E|)$. Once this is completed, a parallel prefix sum is computed on this array in order to obtain the total amount

[2]This can occur when the ratio between the average vertex degree and number of cores is considerably small.

Algorithm 7: Vertex-based algorithm.

```
input  : Graph $G(V,E)$, number of processors $p$
output: Clustering coefficients value $cc_{global}$
For $t \leftarrow 1$ to $p$ do in parallel
    // Stage 1: Workload estimation
    $Pivots_t \leftarrow$ BinarySearch($Vertex\_adj, t \cdot |E|/p$);
    SynchronizationBarrier();
    for $v \leftarrow Pivots_t$ to $Pivots_{t+1}$ do
        $vWork_v \leftarrow 0$;
        for $\forall u \in adj(v)$ do
            $vWork_i \leftarrow deg(v) + deg(u)$;
    SynchronizationBarrier();
    ParallelPrefixScan($vWork$);
    $Pivots_t \leftarrow$ BinarySearch($vWork, t \cdot |E|/p$);
    // Stage 2: CC calculation
    $ccl \leftarrow 0$;
    for $v \leftarrow Pivots_t$ to $Pivots_{t+1}$ do
        $triangles = 0$;
        for $\forall u \in adj(v)$ do
            VertexIntersection($v, deg(v), u, deg(u)$);
            $triangles \leftarrow triangles + |C|$;
        $cc_t \leftarrow cc_t + \frac{triangles}{deg(v)\cdot(deg(v)-1)}$;
    $cc_{global} \leftarrow$ ParallelReduction($cc\_t$);
```

of work computed from the first vertex up to the current vertex. The last entry in the prefix array maintains the expected number of comparisons for the entire graph. Upon completion of the prefix summation, an additional binary search is executed per core into the prefix summation array. The binary search finds the partitioning points of the algorithm. The binary search might actually divide a specific list intersection. As discussed before, the algorithm does not create partitions that divide a single list intersection for simplicity. In reality, this is not a concern and we discuss this in Section 4.4. When the binary search is completed, the partitioning points for clustering coefficients are available and we can proceed to compute the clustering coefficients as part of Stage 2.

5.3.2 Vertex-based approach

We refer to our second load balancing technique as the vertex-based method. Its pseudo-code can be found in Algorithm 7. This approach reduces the storage requirement from $O(|E|)$ to $O(|V|)$ by performing the load balance at a vertex granularity. As a result, some imbalance might be introduced and a single vertex can become a

Table 10: Time complexities of both approaches

Stage	Edge-based approach	Vertex-based approach
Binary search in the offset array.	$O(log(\|V\|))$	$O(log(\|V\|))$
Workload computation per edge.	$O(\|E\|/p)$	$O(\|V\|/p)$
Workload prefix sum.	$O(\|E\|/p + log(p))$	$O(\|V\|/p + log(p))$
Binary search in the workload prefix sum array.	$O(log(\|E\|))$	$O(log(\|V\|))$

bottleneck of the algorithm. We will see in Section 4.4, that this imbalance does not reduce the total performance of the vertex-based approach. In comparison with the edge-based approach, this imbalance is minute.

Similarly to the previous technique, the load-balanced clustering coefficients calculation is comprised of two main stages: 1) the workload estimation and 2) the clustering coefficient calculation.

The amount of work is calculated using expression (13). In a first stage, each thread performs a binary search of the term $t \cdot |E|/p$ over the offset array of the graph CSR. As a result of this work division, each thread is then responsible for a non-overlapping set of vertices and computes the number of comparisons required for each vertex. The results are then stored in an array of size $O(|V|)$. Note that computing the expected number of comparisons needed by the algorithm requires the same number of the steps for both the edge-based and vertex-based approaches, with the key difference in the size of the array used. This is followed by a prefix summation over the $Work$ array. In the final step, a binary search is employed by each core to compute the partition points of the workload. As before, the vertices will be divided among the threads in such a way that their adjacency intersections will not be split.

5.3.3 Complexity analysis

As the amount of work per edge is maintained in array of $O(|E|)$, the spatial complexity of this approach is $O(|E|)$. The time complexity of the load balancing stage in the edge-based approach for each thread is decomposed as described in Table 10.

The work complexity is the time complexity multiplied by a factor of p cores.

Overall, the complexity is of $O(p \cdot log(|V|) + |E| + p \cdot log(|E|) + |E|)$ for the edge-based approach.

In the case of the vertex-based method the complexity is $O(p \cdot log(|V|) + p \cdot log(|E|) + |V| + |E|)$.

5.3.4 Vertex Cover Optimization

In this subsection we briefly discuss how our load balancing technique can be adapted to the vertex cover optimization presented in [75]. This optimization involves computing a vertex cover for the graph and doing the adjacency list intersection only when both vertices of the edge are in the vertex cover. They show that finding the vertex cover takes a small fraction of the total execution and that the requirement that both vertices of an edge be in the vertex cover can reduce the number of list intersections and number of comparisons. This optimization avoids counting the same triangle multiple times. Their optimization can be applied in addition to the lexicographical sorting which reduces the number of times triangles are counted by a factor of two.

To adapt the vertex cover to our algorithm, two modifications are required:

1. Parallel computation of the vertex cover $\hat{V}$

2. Apply the load balancing techniques discussed in this chapter to the vertex cover, $\hat{V}$, instead of the entire vertex set V.

Making these modifications allows creating a load balanced algorithm which avoids duplicate triangle counting.

5.3.5 Summary

We have shown two methods that load balance clustering coefficients. These approaches tradeoff accuracy for spatial complexity. For these methods to be considered as asymptotically optimal as a straightforward parallelization, the load balancing phase has to have lower time complexity than the actual clustering coefficient computation from (12). For many real graphs, including sparse graphs, this will be the case as:

$$O(p \cdot log(|V|) + |E| + p \cdot log(|E|) + |E|) < O(|V| \cdot d_{max}^2) \tag{14}$$

for the edge-based case and

$$O(p \cdot log(|V|) + p \cdot log(|E|) + |V| + |E|) < O(|V| \cdot d_{max}^2) \tag{15}$$

for the vertex-based case.

As a result. Both approaches will offer better performance and core-scaling. We discuss the overhead of this approach in Section 4.4 with respect to real graphs. Note that while the work complexity of clustering coefficient has not changed, the actual time complexity per core changes from $O(Work/p)$ to $\Theta(Work/p)$.

5.4 Results

In this section, we present the experimental performance results for both our new parallel load balanced algorithms. In our tests, we use a 4-socket 40 physical core multicore system made up of the Intel Xeon E7-8870 processor. Each core runs at a 2.40 GHz frequency and has 30 MB of L3 cache per processor. The system has 256 GB of DDR3 DRAM. We test our algorithms over a subset of graphs from the 10th DIMACS Implementation Challenge on Graph Partitioning and Graph Clustering [19]. The graph set used in the tests can be found in Table 11.

We compare the performance of our algorithm with a straightforward parallel algorithm. For the straightforward algorithm, the vertices in the outer loop of Algorithm 5 are evenly split among the cores.

The performance of the algorithms is dependent on the properties of the input graph. We distinguish two key characteristics that may affect substantially the scalability of the different methods such that the straightforward algorithm is likely to

Table 11: Graphs from the 10th DIMACS Implementation Challenge used in our experiments with the clustering coefficient computation runtimes for the different algorithms. Labels: SF - Straightforward (40 threads), V-B - Vertex-Based (40 threads), and E-B - Edge-Based (40 threads).

Name	Graph Type	$\|V\|$	$\|E\|$	Serial	SF	V-B	E-B
audikw1	Matrix	943k	38.35M	30.89	2.52	0.96	1.05
cage15	Matrix	5.15M	47.02M	18.98	1.10	0.70	0.76
ldoor	Matrix	952k	22.78M	7.94	0.26	0.25	0.28
astro-ph	Clustering	16k	121k	0.09	0.006	0.003	0.003
caidaRouterLevel	Clustering	192k	609k	0.39	0.07	0.01	0.01
cond-mat-2005	Clustering	40k	175k	0.10	0.009	0.003	0.004
in-2004	Clustering	1.38M	13.59M	32.67	5.58	1.19	1.21
coAuthorsCiteseer	Collaboration	227k	814k	0.26	0.02	0.01	0.01
coPapersCiteseer	Collaboration	434k	16M	21.37	3.64	0.86	0.88
coPapersDBLP	Collaboration	540k	15.24M	15.26	1.44	0.68	0.72
luxembourg	Road	114k	119k	0.01	0.0002	0.0005	0.0008
belgium	Road	1.44M	1.55M	0.14	0.005	0.007	0.008
road_central	Road	14.82M	16.93M	3.84	0.19	0.26	0.27
road_usa	Road	23.95M	28.85M	3.47	0.13	0.20	0.22
preferAttachment	Clustering	100k	499k	0.26	0.08	0.009	0.01
smallworld	Clustering	100k	499k	0.14	0.004	0.004	0.004
RMAT-18	Random	262k	10.58M	63.78	2.70	2.01	2.08
RMAT-20	Random	1.05M	44.62M	236.28	8.14	7.1	7.38

outperform our algorithms:

1) *Sparsity* - while most of the graphs are considered to be sparse, the road networks are especially sparse where $E \approx V$. The fact that such networks consist of a single connect component implies that the vertices have few adjacencies, which in turns means that the intersection stage will be considerably short. As such our algorithms may potentially introduce overhead for such networks.

2) *Uniform degree distribution* - for graphs in which the degree distribution is uniform and most vertices have the same number of neighbors the workload is inherently balanced as each vertex will require an equal number of comparisons. .

5.4.1 Scaling

To express the workload imbalance for all these algorithms, we define the ratio of computed comparisons as the fraction of the minimum number of comparisons performed by a thread over the maximum number of comparisons. Fig. 33 depicts the workload imbalance, measured by the ratio of the thread with the least work and the thread with the most work, for the three algorithms given a 40 core partition. In Fig. 34 we show this ratio for a subset of graphs as a function of the number of threads. Our results show that the straightforward algorithm offers a balanced partitioning for the networks with uniform degree distribution and very significant sparsity. Notice that for all the graphs, the straightforward approach has both the upper and lower bounds for work distribution. This is true for all the graphs we tested. For some graphs, including *caidaRouterLevel*, the ratio between the minimal and maximal workload can be as high as 100 times.

In contrast to the straightforward partitioning, our edge-based approach delivers a near equal number of comparisons to each core for all the graphs. This fact can be observed in Fig. 33 (the ratio chart), where the bar for the edge-based method is approximately 1 for all the graphs, which is the ideal scenario. Our observation can be reinforced by the data displayed in Fig. 34, as it is almost impossible to differentiate the two curves for minimal and maximal number of comparisons for the edge-based method. The vertex-based approach delivers mixed results. In some cases the partitioning overlaps with that of the edge-based approach. In some cases it differs by $10\% - 20\%$, as some vertices are computationally more demanding and these vertices are not split among several cores.

Results show that the partitioning of the vertex-based approach is not as accurate as that of the edge-based method. Despite this the vertex-based approach offers better performance, as its load balancing stage is slightly less computationally demanding.

The number of comparisons each processor receives can be also relevant to estimate

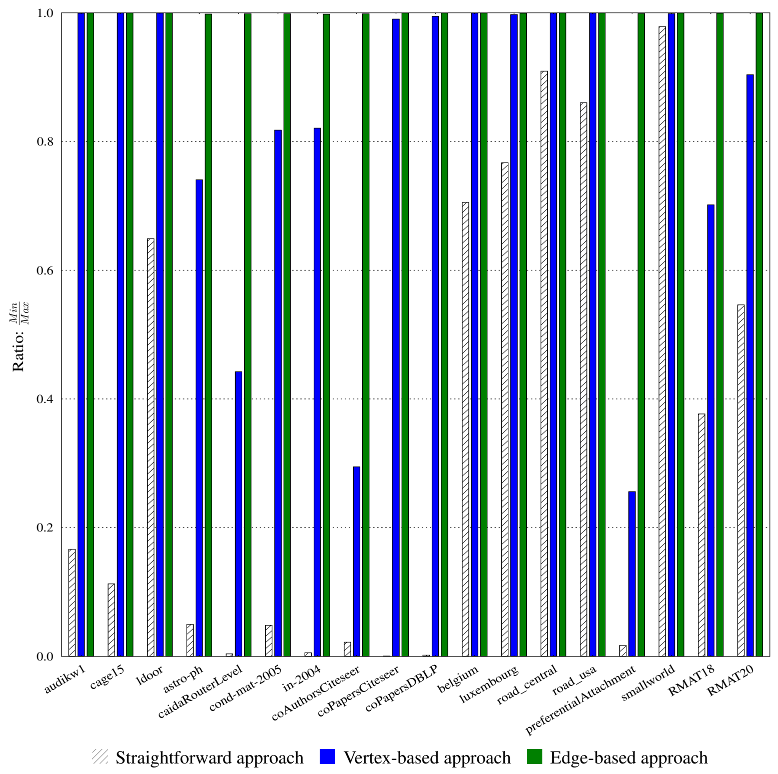


Figure 33: The ordinate is the ratio between the thread with the least amount of work with the thread with the most amount of work based on the number comparisons required by the thread in the adjacency list intersection. The abscissa are the graphs used. Note that for all the graphs the edge-based approach achieves almost perfect partitioning.

an upper bound on the speedup a given parallel clustering coefficients calculation can attain. Given a work distribution, the maximum speedup obtained by parallel computation can be expressed as in expression (9). In the next subsection, we show how such theoretical speedup displays a correlation with the actual speedup attained for the different graphs in the set.

5.4.2 Speedup Analysis

Fig. 35 depicts the speedup obtained for the three algorithms when using 40 cores. Further details of the strong scaling speedups are shown Fig. 36, as a function of the number of threads.

If we consider the "‘Small world"’ graphs, both of our methods show better scalability for 40 cores, which represents an improvement over the straightforward algorithm of $1.5X - 7.5X$. For road network graphs, the straightforward algorithm outperforms both our methods. This is due to the fact that road networks are very sparse, $E \approx V$. As a result, the computation of the intersection represents a smaller fraction of the overall runtime, which includes the load-balancing. In addition, road networks feature a substantially uniform degree distribution. Hence, a straightforward division of the work yields a load-balanced computation. The same behavior can be observed in other graphs that feature uniform degree distributions, such as the *smallworld* network. A clear relationship can be observed between the thread with the most work and the actual speedup. This is not surprising, given the fact that this thread is the execution bottleneck.

We can go a step further and estimate the maximum speedup that can be obtained by a given work distribution. Fig. 37 shows a comparison between the actual speedups obtained for the different graphs when using the straightforward approach and the estimated maximum speedup obtained using Expression (9). Overall, our work estimations provide a fairly precise indicator on the behavior of the algorithm for most of the graphs.

Further insights on the overhead of the work estimation phase are depicted in Fig. 38, which shows the ratio of the time spent computing the clustering coefficients out of the total time spent (including the load balancing) for both our methods on the 40 cores for all the graphs we used. The overhead of our load balancing techniques represents between 1% - 20% of the overall runtime, except for the road network graphs. For the road networks, our overhead is indeed significant. This is mostly due to the fact that very little work is done in the adjacency list intersection stage meaning that the overhead introduced by our techniques plays a more significant role in the overall time.

If the load-balancing stage is not taken into account in the execution time of our new algorithms, the new algorithms would outperform the straightforward algorithm for all thread counts due to load balancing.

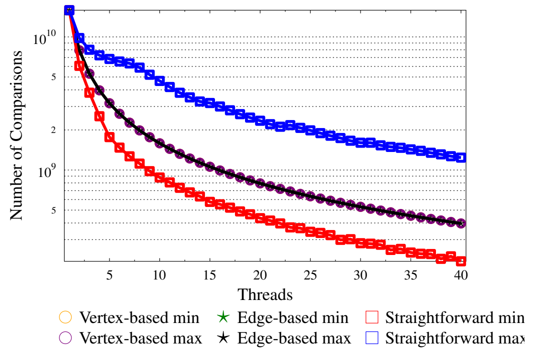
Despite the fact that the edge-based version provides a more balanced work distribution than the vertex-based method, these differences do not translate into a better performance. This is due to the higher computational complexity of the work estimation phase of our edge-based method and synchronization. Notice that for all the graphs, the edge-based approach introduces more overhead than the vertex-based approach. This is caused by the increased memory and computational requirements of this approach. Recall that the edge-based approach uses an $O(E)$ array to store the expected amount of work. This array is then used for the prefix summation process for a total of $O(E)$ operations (whereas the vertex based requires only $O(V)$ operations). Further, this prefix array will be reloaded into the cache for each of the accesses.

5.4.3 Summary

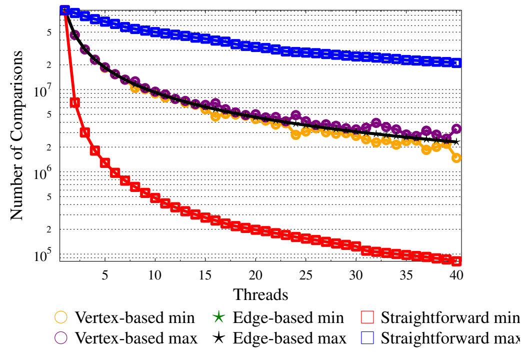
This section presented timing results, scaling results, and workload partitioning for the straightforward parallel implementation, our edge and vertex-based approach using real world graphs from the DIMACS Challenge [19]. We showed that both our load balancing techniques scaled up to 40 cores and can continue to scale to a significantly larger number of cores, whereas the scalability of straightforward algorithm is limited for many types of graphs. In some cases, our approaches show an improvement of a factor of as high as $5.5X - 7.5X$ over the straightforward algorithm. The overhead introduced by our algorithm is discussed.

5.5 Conclusions

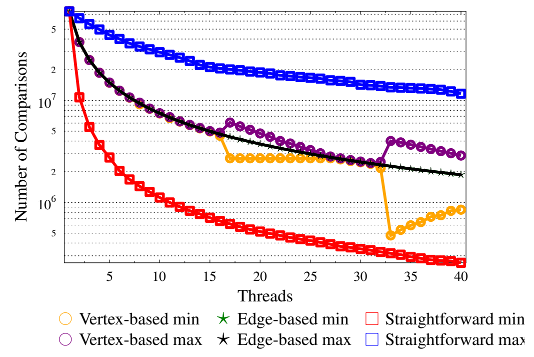
Due to highly skewed vertex degree distributions, computing clustering coefficients on social networks presents big load balancing challenges. In this chapter we presented two parallel methods for computing exact clustering coefficients. By using workload estimation, we achieve effective load-balanced computation for multiple graph topologies. For both of our methods, we present a discussion on the tradeoffs between achieving perfect work distribution and the complexity it requires. In practice, employing an approximate load balancing scheme with a moderate computational cost allows achieving an overall speedup of $25X - 35X$ over the sequential algorithm for most of the graphs. This represents an improvement of $3X - 7.5X$ for real graphs and $1.5X - 4X$ for random graphs over using straightforward parallel approaches. Overall, load balancing is a key element to take into account when leveraging the parallel computing power in graph applications.



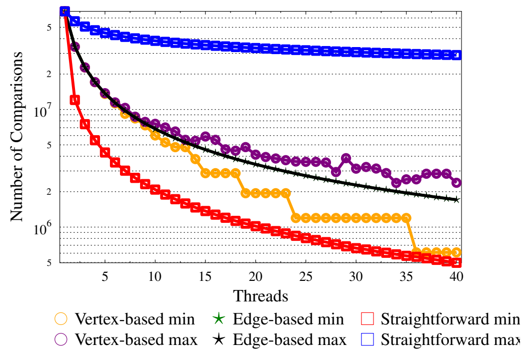
(a) *audikw*1



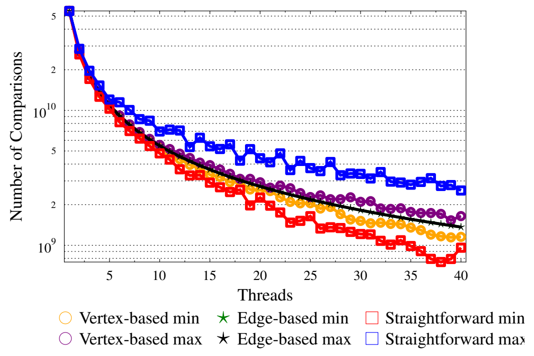
(b) *caidaRouterLevel*



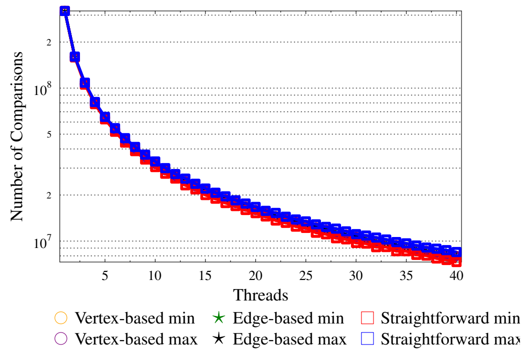
(c) *coAuthorsCiteseer*



(d) *prefAttachment*



(e) $RMAT - 18$



(f) *road_usa*

Figure 34: The abscissa is the number of threads used. The ordinate is the number of comparisons required for a specific thread count. Two curves are shown for each of the algorithms - for threads that receive the most and least number of comparisons. The straightforward partitioning is the lower and upper for all the figures.

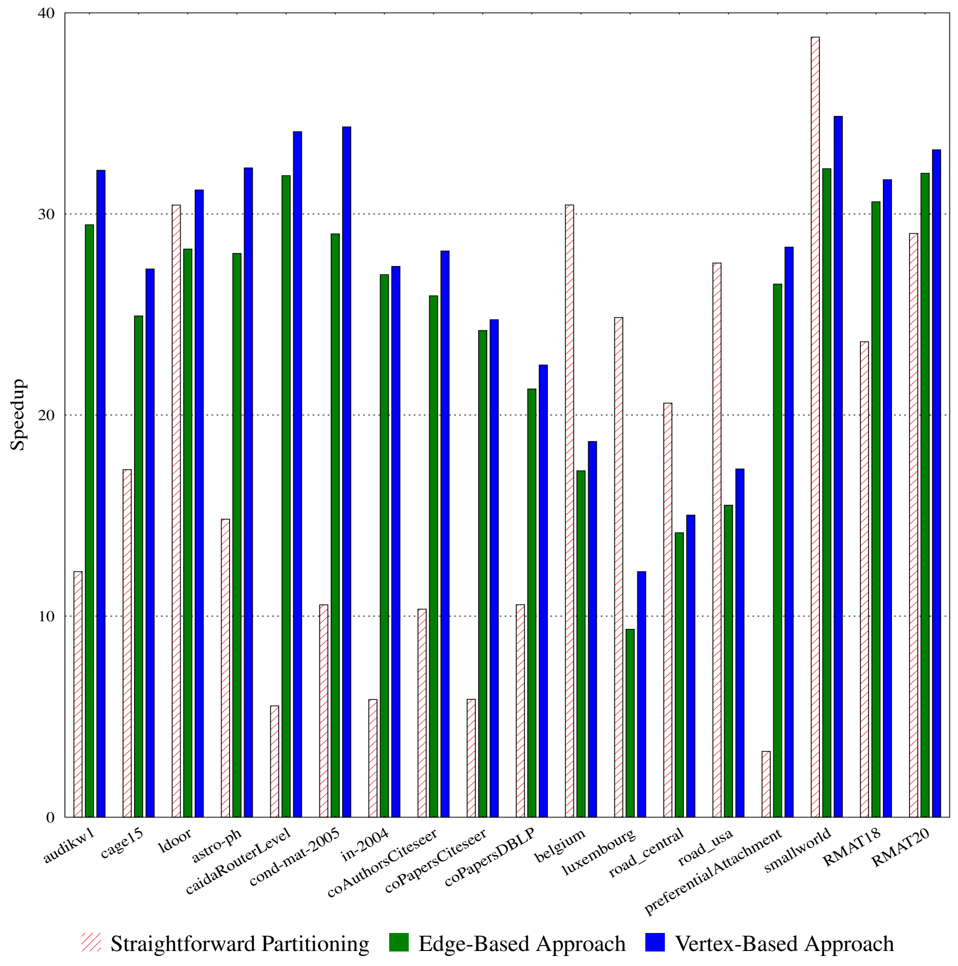
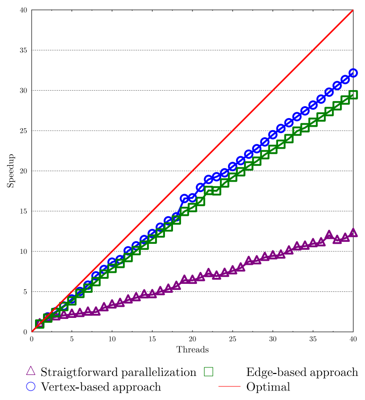
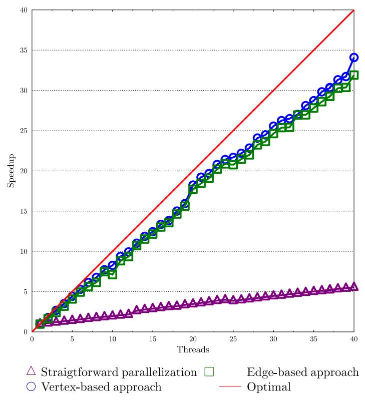


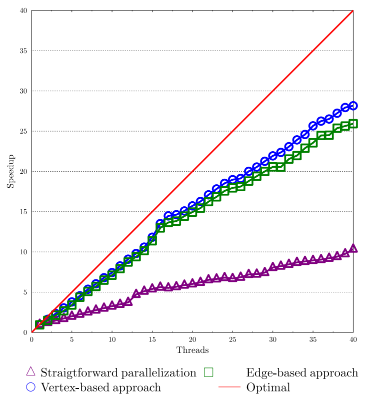
Figure 35: Speedup obtained with 40 cores for the different algorithms



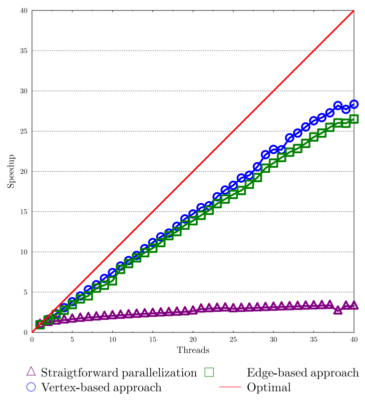
(a) *audikw*1



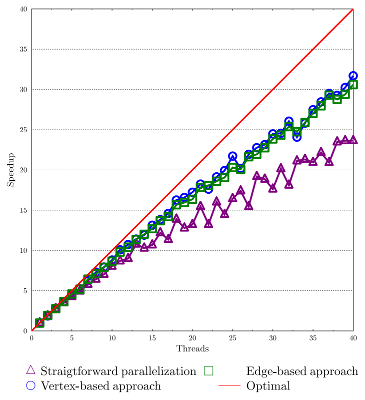
(b) *caidaRouterLevel*



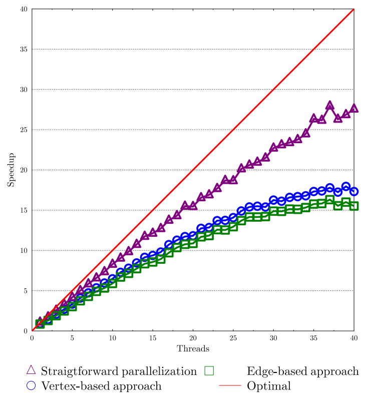
(c) *coAuthorsCiteseer*



(d) *prefAttachment*



(e) $RMAT-18$



(f) *road_usa*

○ Vertex-based approach □ Edge-based approach △ Straigtforward parallelization

Figure 36: The ordinate for all the subfigures is the speedup as a function of the number of threads (the abscissa).

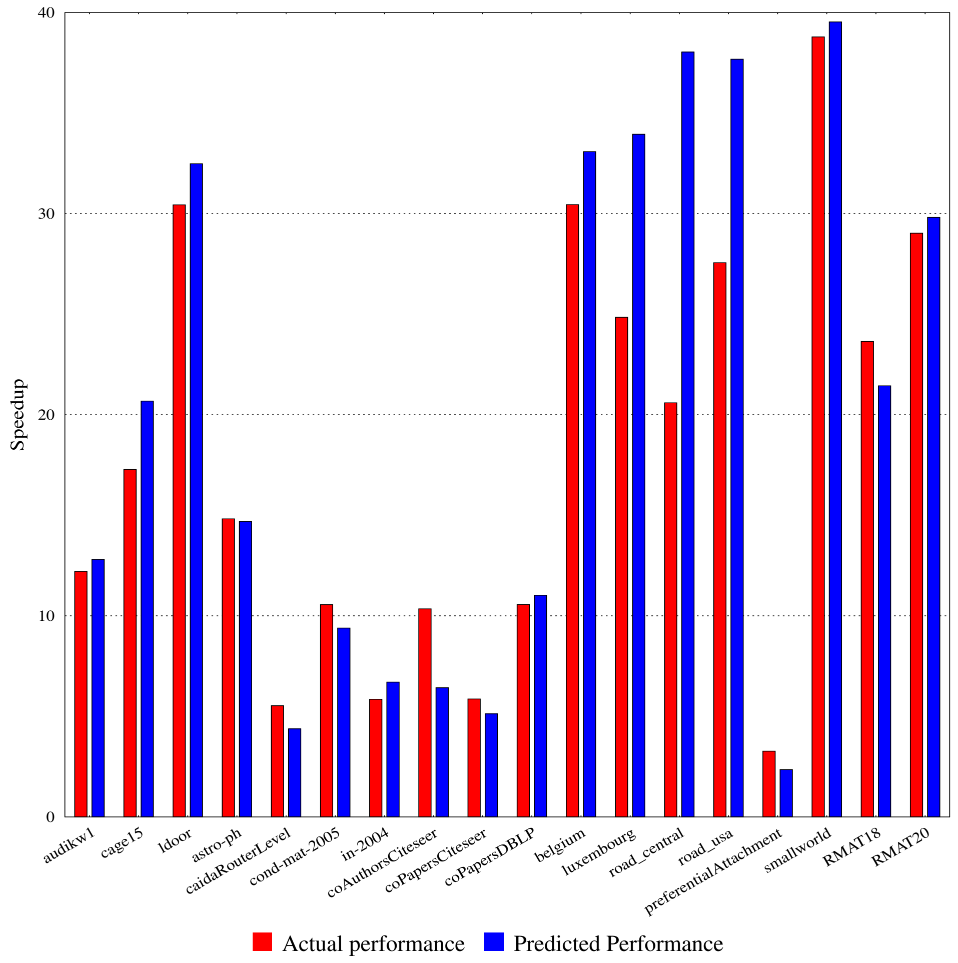


Figure 37: Speedup obtained for 40 cores using straightforward division algorithm in comparison with the estimated speedup.

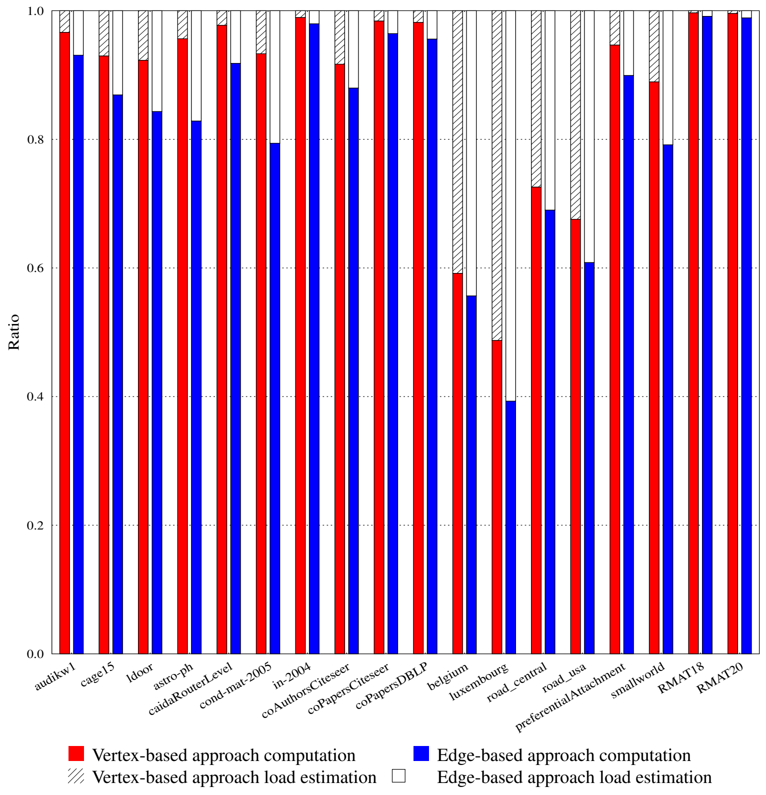


Figure 38: Percentage of time spent in both the load balancing phase vs. the list intersection phase for both approaches.

CHAPTER VI

DYNAMIC PARALLEL CONNECTED COMPONENTS

This chapter is an extension of the paper: R. McColl, O. Green, D. Bader, "A New Parallel Algorithm for Connected Components in Dynamic Graphs", IEEE International Conference on High Performance Computing, Hyderabad, India, 2013.

Social networks, communication networks, business intelligence databases, and large scientific data sources now contain hundreds of millions elements with billions of relationships. The relationships in these massive datasets are changing at ever-faster rates. Through representing these datasets as dynamic and semantic graphs of vertices and edges, it is possible to characterize the structure of the relationships and to quickly respond to queries about how the elements in the set are connected. Statically computing analytics on snapshots of these dynamic graphs is frequently not fast enough to provide current and accurate information as the graph changes. This has led to the development of dynamic graph algorithms that can maintain analytic information without resorting to full static recomputation.

In this work we present a novel parallel algorithm for tracking the connected components of a dynamic graph. Our approach has a low memory requirement of $O(V)$ and is appropriate for all graph densities. On a graph with 512 million edges, we show that our new dynamic algorithm is up to $128X$ faster than well-known static algorithms and that our algorithm achieves a $14X$ parallel speedup on a x86 64-core shared-memory system. To the best of the authors' knowledge, this is the first parallel implementation of dynamic connected components that does not eventually require static recomputation.

6.1 Introduction

In graph theory, given an undirected graph $G = (V, E)$, a connected component $C \subseteq V$ ensures that for each $s, t \in C$ there is a path between s and t. Finding the connected components of a graph is a well-studied problem. The component labeling of a graph can be used as building block within other calculations: betweenness centrality, community detection, image processing, and others [51]. Hopcroft and Tarjan [85] presented one of the first approaches for partitioning a graph into connected components using a series of Depth First Searches (DFS), one for each component. Breadth First Search (BFS) can also be used in place of DFS. Approaches relying only on DFS and BFS are aimed at static graphs which can be thought of as snapshots of a dynamic graph at an exact time.

In examining social networks such as Facebook, where vertices and edges may represent people and friendships or messages, the high rate of change makes computing many analytics on snapshots of these massive graphs impractical as the time between updates is much less than the time needed to compute these analytics. This has led to the development of algorithms for dynamic graphs in which edges can be inserted or deleted. With respect to connected components, edge insertions may join two different components, and edge deletions may split one component into two. Given the graph G and the components labels C, determining if an insertion has joined two components can be done in $O(1)$ time; however, determining if a deletion has broken a component is more expensive. Both of these scenarios must be detected and handled by any dynamic graph algorithm.

In order to keep up with rapid changes, the algorithm and its implementation must attain performance through full system utilization and workload balancing. A strategy used in this and other works is to aggregate updates into a batch over time or until a certain number are collected. This batch can then be applied in parallel. Batches increase the available amount of parallel work and provide opportunities to reduce redundant calculations between updates; however, they also present synchronization challenges and the potential for workload imbalance.

In this work, we show how to maintain an exact labeling of the connected components of a dynamic graph with millions of edges while applying batches of edge insertions and deletions in parallel. To accomplish this task, we employ a "parent-neighbor" sub-graph structure of up to a fixed size $O(V)$. In this sub-graph, parent and neighbor relationships represent paths to the root of a breadth-first traversal of each component. As long as each vertex has a path to the root, the component is unbroken. In practice, we show that the average case for maintaining this approach is much faster than performing the $O(V + E)$ work required to recompute from scratch. The storage complexity is $O(V)$.

The remainder of the chapter will be structured as follows: this section presents related work on serial and parallel connected components algorithms for both static and dynamic graphs. In Section II, we present our new algorithm and the required data structures. In Section III, we will discuss experimental methodology. Section IV gives quantitative and performance results. Finally, in Section V, we will give conclusions and future work.

6.1.1 Related Work

Many authors have published a variety of parallel algorithms that compute solutions to the connected components problem on shared-memory computers. Hirschburg *et al.* [82] presented the CONNECT algorithm, a classic parallel algorithm to find the connected components in an undirected graph. Two variations are presented, the first requires $|V|^2$ processors and the second requires $V\lceil V \log V \rceil$ processors. Both have a time complexity of $O(\log^2 V)$.

Shiloach and Vishkin [129] gave an $O(\log V)$ algorithm that used $|V| + 2|E|$ processors. Because of its simplicity, parallelism, and load balancing it has been implemented on several multi-core and many-core systems. In the average case, it completes in $\sim d/2$ iterations, where d is the diameter of the graph.

Shiloach and Even present an algorithm that tracks connected components in dynamic graphs as edges are removed [127]. They accomplish this by maintaining a structure representing the vertices in the levels of breadth first search tree for each component. For each deleted edge, if the vertices of the edge are on the same level, the data structure does not require updating. However, if they are on different levels and the lower vertex has no other neighbors above it, the lower vertex (and possibly a subtree) could potentially drop a level or fall out of the tree altogether. Both of which require updating the data structure. In more recent theoretical work [118], a sequence of graphs are maintained, one per each edge inserted, and reachability trees. The amortized update time is $O(E + V\log V)$ and a worst-case query time of $O(V)$.

In [60] a technique is shown that allows treating dense graphs as sparse graphs - this is known as sparsification. Sparsification is achieved by dividing the original graph into smaller subgraphs with V vertices and $O(V)$ edges. Certificates (aka graph properties) are computed for each subgraph. This is followed by merging of these certificates. Ferragina [65] uses the sparsification technique with the algorithm of [94] to give an additional algorithm for computing static and parallel connected components for a PRAM-like system.

Henzinger *et al.* [80] use a series of graphs in which the vertices are colored based on their degrees to detect new components in the face of deletions. Deleted edges are removed from all of the graphs that are represented and the colors are updated, then any $O(V + E)$ connected components algorithm is run on all graphs to discover components containing only vertices of a certain color which indicates the creation of a new component.

Henzinger and King [81] created another algorithm that maintains several spanning trees in each component for different levels of sparsity and performs updates on the trees only when deletions occur in both the graph and the tree.

The problem with many of these streaming algorithms is that they are too expensive to compute in practice, require too much storage - even up to the size of the graph $O(V + E)$, ignore real world graph properties, or do not consider practical multi-core and many-core systems.

6.2 Tracking Connected Components in a Dynamic Graph

In the following section we present our new algorithm for maintaining connected components for dynamic graphs. We briefly discuss the concept of tracking connected components in a dynamic graph. We follow this with an introduction to our new algorithm and data structure. Finally, the insertion and deletion approaches are discussed. For simplicity, the pseudo code in this section does not explicitly indicate atomic instructions.

Dynamic graph algorithms present several challenges which include: 1) Correctness: for exact algorithms, the results should be correct and consistent at fixed points in time for the graph at that same point. 2) Parallelism: to achieve the performance necessary to keep up with high-speed data streams, an algorithm must be able to use all of the resources available in the system. Additionally, synchronization and communications need to be minimized. 3) Time complexity: the complexity of the dynamic algorithm should be better than that of the static; however, as long as the real-world performance of the dynamic algorithm is better on average on the data of interest, it may be tolerable for the complexities to be equivalent. 4) Storage complexity: this too should generally be comparable to or better than the static case. Computing the connected components of a static graph requires $O(V + E)$ memory including the component labels $O(V)$ and the graph itself $O(V + E)$. A dynamic graph algorithm should not increase this bound. Given that the size of the graph can be on the same scale as the total memory in the machine, it is preferred that a dynamic graph algorithm limit the amount of extras storage required to $O(V)$.

Updating the connected components following an edge insertion only requires a comparison of the component labels of the vertices belonging to the edge. If the vertices have the same component label, then the insertion operation is complete. If the vertices have different component labels, then the two independent components need to be relabeled as the same component.

When maintaining a component labeling, edge deletion is considerably more challenging to handle than edge insertion. Deletions require determining if the deleted edge was the single path connecting two otherwise independent components. This is easy when the deleted edge is the only edge incident to one (or both) of the vertices; however, if both vertices have additional adjacencies, alternative path(s) between the vertices may exist. Obviously it is possible to use a SPSP (Single Pair Shortest Path) algorithm such as BFS to verify that an alternative path exists. Unfortunately, the worst case complexity for this is $O(V+E)$ which is the same as the complexity bound for computing connected components. For the cases that the edge deletion did split the connected component into two parts, it is necessary to search and relabel the new components.

It becomes apparent that it is desirable to find a mechanism that can state if the deletion is “safe”, meaning that it is possible to state in $O(1)$ time whether or not the deletion could have broken a component. We require that this mechanism have a 100% true positive rate – all deletions marked as safe are truly safe, but allow for some false negatives as the search and relabel process will appropriately handle these cases.

The key challenge is minimizing the false negatives - the cases where the mechanism suggests that the deletion is unsafe when it actually is safe. This goes back to reducing the need to search for an additional path between the vertices to avoid doing the same work as static recomputation.

6.2.1 The Parents-Neighbors Sub-graph

In this subsection we present our algorithm and its respective data structure that has a low memory requirement of $O(V)$. Some approaches have higher memory requirements of $O(V+E)$. This limits the size of graph that can be analyzed. Further, a smaller memory footprint can allow for better usage of the cache and reduced dependence on memory bandwidth.

We call our data structure the parent-neighbor sub-graph. This sub-graph is extracted using breadth-first traversals of the original graph (one for each connected component). The result is a directed sub-graph of the original undirected graph. Each vertex will maintain a list of vertices that are in the level above ("parents") and/or in the same level ("neighbor") of the traversal where a level is a single frontier in the search and parents / neighbors of a vertex must be adjacent to that vertex in the original graph. Note that if all the parents and neighbors were maintained then the memory requirement of this would be $O(V + E)$. Instead we place a threshold ($thresh_{PN}$) on the total number of parent-neighbors for each vertex. Given that each vertex can have at most $thresh_{PN}$ parent-neighbors, the storage requirement of the parent-neighbor subgraph is $O(V \cdot thresh_{PN})$. Since $thresh_{PN}$ is a constant $O(1)$, the storage requirement can be reduced to $O(V)$. In Section 2.4 we discuss the impact of selecting $thresh_{PN}$.

This sub-graph is similar to the parent lists maintained in Brandes's betweenness centrality algorithm [30]. In that algorithm, every vertex has a list of the vertices in the level above that are adjacent to it. Since each vertex stores a list of up to the full size of its adjacency list, the memory required for Brandes's algorithm is $O(V + E)$. The key differences between our parent-neighbor sub-graph and the parent lists of

Table 12: The data structures maintained while tracking dynamic connected components.

Name	**Description**	**Type**	**Size (Elements)**
C	Component labels	array	$O(V)$
$Size$	Component sizes	array	$O(V)$
$Level$	Approximate distance from the root	array	$O(V)$
PN	Parents and neighbors of each vertex	array of arrays	$O(V \cdot thresh_{PN}) = O(V)$
$Count$	Counts of parents and neighbors	array	$O(V)$
$thresh_{PN}$	Maximum count of parents and neighbors for a given vertex	value	$O(1)$
$\tilde{E}_I$	Batch of edges to be inserted into graph	array	$O(batch\ size)$
$\tilde{E}_R$	Batch of edges to be deleted from graph	array	$O(batch\ size)$

[30] are that we have placed a bound on the maximum number of adjacencies in the list and that our list also stores adjacent vertices that are on the same level.

6.2.2 Data Structure and Algorithm Details

Table 12 denotes the variables used by the algorithm and data structure. We give a brief justification for the memory requirements. As each vertex knows the component it belongs to and the size of its component, a total of $O(V)$ memory is required.

To store the partial list of parents and neighbors of a vertex we have created an array called PN. To distinguish between the parents and neighbors, the parents use positive numbers and the neighbors use negative numbers. The benefit of such an implementation is that there is a single array and single counter for each vertex. Additional benefits include spatial locality, reduced memory footprint vs. separate arrays, and ease of programming. For this reason, vertices will be indexed $1, 2, 3, .., |V|$.

In the next subsection we further elaborate on the $Level$ array, yet, we want to note ahead of time that this array does not maintain the actual distance from the root but only an approximate distance as will become apparent. In addition to this, we use “negative” distances to mark vertices that potentially have lost all their parents yet still have neighbors. These vertices should still have a path to the root through their neighbors. The negative simply indicates to other vertices that this vertex should not be depended on when searching for a connection to the root.

As PN is an undirected sub-graph of a directed graph, each inserted or deleted

Algorithm 8: A parallel breadth-first traversal that extracts the parent-neighbor subgraph.

```
for $v \in V$ do
    $Level[v] \leftarrow \infty$, $Count[v] \leftarrow 0$
for $v \in V$ do
    if $Level[v] = \infty$ then
        $Q[0] \leftarrow v$
        $Q_{start} \leftarrow 0$
        $Q_{end} \leftarrow 1$ $Level[v] \leftarrow 0$
        $C_{id}[v] \leftarrow v$
        while $Q_{start} \neq Q_{end}$ do
            $Q_{stop} \leftarrow Q_{end}$
            for $i \leftarrow Q_{start}$ to $Q_{stop}$ in parallel do
                for each neighbor $d$ of $Q[i]$ do
                    if $Level[d] = \infty$ then
                        $Q[Q_{end}] \leftarrow d$
                        $Q_{end} \leftarrow Q_{end} + 1$
                        $Level[d] \leftarrow Level[Q[i]] + 1$
                        $C_{id}[d] \leftarrow C_{id}[Q[i]]$
                    if $Count[d] < thresh_{PN}$ then
                        if $Level[Q[i]] < Level[d]$ then
                            $PN_d[Count[d]] \leftarrow Q[i]$
                            $Count[d] \leftarrow Count[d] + 1$
                    else if $Level[Q[i]] = Level[d]$ then
                        $PN_d[Count[d]] \leftarrow -Q[i]$
                        $Count[d] \leftarrow Count[d] + 1$
            $Q_{start} \leftarrow Q_{stop}$
        $Size[v] \leftarrow Q_{end}$
```

edge, (u, v), is taken care of twice – once from v's perspective and once u's perspective.

6.2.3 Data Structure Initialization

Each vertex is initially unlabeled, its *Level* is set to ∞, and its *Counter* is set to zero. Component sizes are set to zero. In addition to the structures listed in Table 12, a temporary workspace of two $|V|$ queues is used during the initialization.

We use a series of parallel BFS traversals, one for each connected component, to find and label the members of each component and initialize the component sizes, parents, and neighbors. Our BFS can be found in Algorithm 8. The first unlabeled vertex is selected and enqueued. While the queue is not empty, the edges of the vertices in the current frontier are explored concurrently. Newly discovered vertices are marked, enqueued to the next frontier, and a new parent is inserted. For previously discovered vertices, two scenarios of interest can arise: a new parent has been found or a new neighbor has been found. These will be added to the PN array.

Insertion of the current vertex as a parent or neighbor only occurs if the total number of parents and neighbors for the adjacent vertex is less than $thresh_{PN}$. Since the BFS is level-synchronous, all parents of a vertex will be found before even a single neighbor is found. Neighbors will only be added to the parent-neighbor list if the vertex did not have at least $thresh_{PN}$ parents. Synchronization is handled through atomic **compare-and-swap** operations on the *Level* array and atomic **fetch-and-add** to enqueue newly found vertices and to update the PN array and counter. For simplicity these have not been marked in the pseudo code.

As a slight optimization, all edges connecting singleton connected components are skipped in the first pass. Once the larger components have been labeled, a parallel pass over all vertices is used to initialize the singleton components.

6.2.4 Insertions and the Subgraph

For an edge insertion, the vertices on each edge are first checked to see if they belong to the same component. The pseudo-code for edge insertion can be found in Algorithm 9. We differentiate two key scenarios for the insertion of edge $\langle s, d \rangle$: 1) the edge is within a connected component (intra-connecting) and 2) the edge joins two components (inter-connecting).

For the first, the levels of s and d are checked to see if a new parent or neighbor relationship can be created. Assume that $Level_s \leq Level_d$. If $Counter_d < thresh_{PN}$ then s is added to PN_d as a parent if ($Level_s < Level_d$) or as a neighbor otherwise, and $Counter_d$ is incremented. If $Counter_d = thresh_{PN}$ and $Level_s < Level_d$ (i.e. s can be a parent), d's parents and neighbors are searched for neighbors that could be replaced by the parent s.

Algorithm 9: The algorithm for updating the parent-neighbor subgraph for inserted edges.

```
for all$\langle s,d\rangle \in \tilde{E}_I$ in parallel do
    $E \leftarrow E \cup \langle s,d\rangle$ insert$(E, \langle s,d\rangle)$ if $C_{id}[s] = C_{id}[d]$ then
        if $Level[s] > 0$ then
            if $Level[d] < 0$ then
                // d is not "safe"
                if $Level[s] < -Level[d]$ then
                    if $Count[d] < thresh_{PN}$ then
                        $PN_d[Count[d]] \leftarrow s$
                        $Count[d] \leftarrow Count[d] + 1$
                    else
                        $PN_d[0] \leftarrow s$
                    $Level[d] \leftarrow -Level[d]$
        else
            if $Count[d] < thresh_{PN}$ then
                if $Level[s] < Level[d]$ then
                    $PN_d[Count[d]] \leftarrow s$
                    $Count[d] \leftarrow Count[d] + 1$
                else if $Level[s] = Level[d]$ then
                    $PN_d[Count[d]] \leftarrow -s$
                    $Count[d] \leftarrow Count[d] + 1$
            else if $Level[s] < Level[d]$ then
                for $i \leftarrow 0$ to $thresh_{PN}$ do
                    if $PN_d[i] < 0$ then
                        $PNV_d[i] \leftarrow s$,
                        Break for-loop
    $\tilde{E}_I \leftarrow \tilde{E}_I \backslash \langle s,d\rangle$
for all$\langle s,d\rangle \in \tilde{E}_I$ do
    if $C_{id}[s] \neq C_{id}[d]$ then
        if $Size[s] = 1$ then
            $Size[s] \leftarrow 0$
            $Size[d] \leftarrow Size[d] + 1$
            $C_{id}[s] \leftarrow C_{id}[d]$
            $PN_s[0] \leftarrow d$
            $Level[s] \leftarrow$ abs$(Level[d]) + 1, Count[s] \leftarrow 1$
        else
            connectComponent(Input, $s, d$)
```

The intra-connecting edges are handled in parallel. The inter-connecting edges are handed consecutively upon completion of the intra-connecting edges.

When components are connected, a parallel BFS starts at the joining vertex for the smaller of the two components to relabel the smaller component's members and

Algorithm 10: The algorithm for updating the parent-neighbor subgraph for deleted edges.

```
for all$\langle s,d\rangle \in \tilde{E}_R$ in parallel do
    $E \leftarrow E \backslash \langle s,d\rangle$
    $hasParents \leftarrow$ false for $p \leftarrow 0$ to $Count[d]$ do
        if $PN_d[p] = s$ or $PN_d[p] = -s$ then
            $Count[d] \leftarrow Count[d] - 1$
            $PN_d[p] \leftarrow PN_d[Count[d]]$
        if $PN_d[p] > 0$ then
            $hasParents \leftarrow$ true
    if (not $hasParents$) and $Level[d] > 0$ then
        $Level[d] \leftarrow -Level[d]$
for all $\langle s,d\rangle \in \tilde{E}_R$ in parallel do
    for all $p \in PN_d$ do
        if $p \geq 0$ or $Level[\mathbf{abs}(p)] > 0$ then
            $\tilde{E}_R \leftarrow \tilde{E}_R \backslash \langle s,d\rangle$
$PREV \leftarrow C_{id}$
for all $\langle s,d\rangle \in \tilde{E}_R$ do
    $unsafe \leftarrow (C_{id}[s] = C_{id}[d] = PREV_s)$
    for all $p \in PN_d$ do
        if $p \geq 0$ or $Level[\mathbf{abs}(p)] > 0$ then
            $unsafe \leftarrow$ false
    if $unsafe$ then
        if $\{\langle u,v\rangle \in G(E,V) : u = s\} = \emptyset$ then
            $Level[s] \leftarrow 0, C_{id}[s] \leftarrow s$
            $Size[s] \leftarrow 1, Count[s] \leftarrow 0$
        else
            Algorithm 4
            repairComponent(Input, $s, d$)
```

add them to the larger component's tree in PN. For performance purposes, singleton components are set aside during this step. In the following step, all singletons handled in parallel. Also, since two or more components could be connected through multiple edge insertions within a single batch, inter-connecting insertions are checked to see if the components have already been rebuilt and relabeled by another insertion before the parallel BFS rebuild is performed.

6.2.5 Deletions and the Subgraph

Once the data structure has been initialized, edge deletions within the graph can be checked against the parents and neighbors of the involved vertices to determine if the

Algorithm 11: The algorithm for repairing the parent-neighbor subgraph when an unsafe deletion is reported.

```
$Q[0] \leftarrow d$
$Q_{start} \leftarrow 0$
$Q_{end} \leftarrow 1$
$SLQ \leftarrow \emptyset$
$SLQ_{start} \leftarrow 0$
$SLQ_{end} \leftarrow 0$
$Level[d] \leftarrow 0$, $C_{id}[d] \leftarrow d$
$disconnected \leftarrow$ true
while $Q_{start} \neq Q_{end}$ do
    $Q_{stop} \leftarrow Q_{end}$ for $i \leftarrow Q_{start}$ to $Q_{stop}$ in parallel do
        $u \leftarrow Q[i]$ for each neighbor $v$ of $u$ do
            if $C_{id}[v] = C_{id}[s]$ then
                if $Level[v] \leq$ abs($Level[d]$) then
                    $C_{id}[v] \leftarrow C_{id}[d]$
                    $disconnected \leftarrow$ false
                    $SLQ[SLQ_{end}] \leftarrow v$
                    $SLQ_{end} \leftarrow SLQ_{end} + 1$
                else
                    $C_{id}[v] \leftarrow C_{id}[d]$
                    $Count[v] \leftarrow 0$
                    $Level[v] \leftarrow Level[u] + 1$
                    $Q[Q_{end}] \leftarrow v$
                    $Q_{end} \leftarrow Q_{end} + 1$
            if $Count[v] < thresh_{PN}$ then
                if $Level[u] < Level[v]$ then
                    $PN_v[Count[v]] \leftarrow u$
                    $Count[v] \leftarrow Count[v] + 1$
                else if $Level[v] = Level[v]$ then
                    $PN_v[Count[v]] \leftarrow -u$
                    $Count[v] \leftarrow Count[v] + 1$
    $Q_{start} \leftarrow Q_{stop}$
```

deletion is safe. The pseudo-code for edge deletion can be found in Algorithm 10.

Here we will focus on the deleted edge $\langle s, d \rangle$ from d's perspective again assuming $Level_s \leq Level_d$. Since the graph is undirected, the same process is repeated for s. This is crucial, as a deletion marked safe in one direction may still be considered unsafe from the other.

To determine if the deletion was safe, if s is in PN_d it is removed, and PN_d is searched for a remaining parent. If a parent remains whose level is non-negative, a connection to the root of the component must exist, the deletion is safe, and nothing

Algorithm 11: Cont.

```
if $disconnected$ then
    $Size[d] \leftarrow Q_{end}$
else
    for $i \leftarrow SLQ_{start}$ to $SLQ_{end}$ in parallel do
        $C_{id}[i] \leftarrow C_{id}[s]$
    while $SLQ_{start} \neq SLQ_{end}$ do
        $SLQ_{stop} \leftarrow SLQ_{end}$ for $i \leftarrow SLQ_{start}$ to $SLQ_{stop}$ in parallel do
            $u \leftarrow SLQ[i]$ for each neighbor $v$ of $u$ do
                if $C_{id}[v] = C_{id}[d]$ then
                    $C_{id}[v] \leftarrow C_{id}[u]$
                    $Count[v] \leftarrow 0$
                    $Level[v] \leftarrow Level[u] + 1$
                    $SLQ[SLQ_{end}] \leftarrow v$
                    $SLQ_{end} \leftarrow SLQ_{end} + 1$
                if $Count[v] < thresh_{PN}$ then
                    if $Level[u] < Level[v]$ then
                        $PN_v[Count[v]] \leftarrow u$
                        $Count[v] \leftarrow Count[v] + 1$
                    else if $Level[v] = Level[v]$ then
                        $PN_v[Count[v]] \leftarrow -u$
                        $Count[v] \leftarrow Count[v] + 1$
        $Q_{start} \leftarrow Q_{stop}$
```

else needs to be done.

If d no longer has parents, a marker in the form of $Level_d \leftarrow -Level_d$,is placed to indicate this fact to the neighbors of d. This marker will be removed only when an inserted edge creates a new parent for d or PN_d is recreated during a component merger or split. If d still has neighbors, they will be checked to see that they are still valid (i.e. $Level > 0$), meaning that they have a path to the root. If such a path exists, then d has a path to the root. If so, from the perspective of d, the deletion was safe.

The first parallel for loop updates the parents and neighbors of the vertices involved in the edge deletion. Note, that the safety of the deleted edges is not confirmed by the end of this loop. Due to parallel race conditions that may cause two neighboring deleted edges to assume that they have paths to the root through each other, the data structure must be updated in the first parallel loop and safety must be checked in a secondary parallel loop by verifying that each vertex has it least one parent or valid neighbor.

For each unsafe deletion, the parent-neighbor graph needs to be corrected. This is done using a partial parallel breadth first traversal for which the pseudo-code can be found in Algorithm 11. In an early version of the algorithm, the approach was instead to perform a full search and simply rebuild the component, but the performance of this approach was found to be inferior to the presented approach.

The goal of this search is to find connections from the starting vertex d back to the root of the component by searching for other vertices in the same level that still have parents or vertices in the level closer to the root.

Initially, d is marked as the root of a new component and a BFS is begun to update PN data structure and component labels. If a connection to the original component is not found, the component is split into two and a new component rooted at d is created by this search process. If the search finds a connection back to the original component, the first search ends and a second traversal is started to relabel and rebuild part of the original component. The second traversal begins from the set of vertices found in that last frontier of the first and proceeds backward toward d. The resulting sizes of the breadth first searches are used to reconcile the component sizes. No vertices closer to the root than the level of d will ever be added to the search or relabeled. This limits the work of the search in the average case. Since unsafe deletions are processed consecutively, the parents and component labels are quickly checked before processing an unsafe delete to determine if the unsafe condition has already been repaired. As a result, in the worst case, the combined number of edges traversed by all searches in this step is limited to the number of edges in the graph; thus, the worst-case performance is equal to that of a static re-computation. As a slight optimization, vertices are checked to see if they are of degree zero and are directly initialized to being their own components.

Table 13: Graph sizes used in our experiments for testing the algorithm. Multiple graphs of each size were used.

	Totals edge per average degree			
Vertices	8	16	32	64
2M	16M	32M	64M	128M
4M	32M	64M	128M	256M
16M	128M	256M	512M	—

6.3 Experimental Methodology

6.3.1 Synthetic Graphs and Experiments

Due to proprietary constraints, researchers do not always have access to real social networks for investigation and many of the datasets that are available are static. Furthermore, use of a single data set can limit the applicability of experiments. Instead, synthetic networks are used. Generating synthetic networks gives experimenters control over the size and properties of the network. Many works have been written using the Erdös-Rènyi (ER) [61, 62] model which uses a uniform random distribution for generating edges; however, this tends to create one well-connected component in which it is unlikely that an insertion or deletion would ever connect or disconnect any components. As such, we do not use this type of random graph.

In this work, we use an implementation of the Recursive Matrix (R-MAT) [39] synthetic random graph generator. This generator recursively divides the adjacency matrix into quadrants, randomly selects one of these quadrants with probabilities a, b, c, and d, and continues this process recursively until the selected quadrant is of size one. For our experimentation , we have used $a = 0.55, b = c = 0.1$, and $d = 0.25$. R-MAT graphs mimic the structure of real social networks in that they have a skewed degree distribution that follows a power law and tend toward one large component and many smaller components and singletons.

In this chapter, we vary the size of the graph in terms of its scale S and edge factor E, where the number of vertices is 2^S and the number of edges is $E \cdot 2^S$. E thus corresponds to the average degree. We used scales 21, 22, and 24 with edge factors 8, 16, and 32 (qualitative results also include edge factor 64). We refer to these graphs as R-MAT-21, R-MAT-22, and R-MAT-24 with the average adjacencies. The sizes of these graphs are listed in Table 13. We generated three graphs and an update stream for each scale and edge factor combination using different random seeds. For example for an R-MAT-21 graph with $E = 8$, three graphs and three streams were generated.

An update stream consists of a series of edges to be inserted or deleted. A fixed probability p_{delete} is used to determine whether or not an update will be a deletion. Deletions are selected from previous insertions.

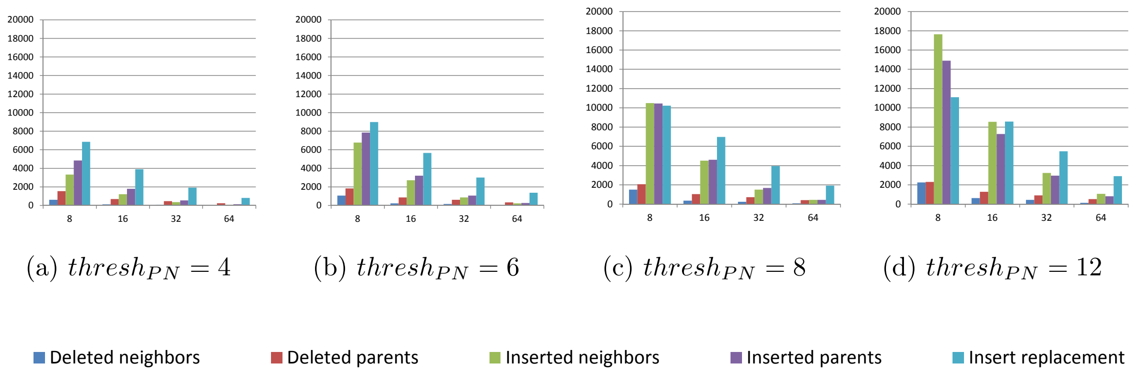
In [55, 56] batches of $100K$ updates are used with $p_{delete} = 6.25\%$. For consistency, we use these parameters as well. For each graph, 10 batches of $100K$ are used. RMAT can potentially duplicate existing edges. We ignore these as they already are in the graph. A single deletion removes an edge regardless of the number of times that it has been inserted.

The system used for our tests is a quad-socket system with four 16-core AMD Opteron 6282 SE processors for a total of 64 cores running at 2.6GHz. Each core has a private 1MB L2 cache, and each processor has a shared 16MB L3 cache. The system has 256GB of DDR3 RAM running at 1600MHz.

6.3.2 First Attempts

During the creation of our new algorithm, we attempted several other approaches that were rejected due to being too computationally demanding, requiring a full static recomputation for each batch of $100K$, or having limited parallel scalability. These are presented here with a focus on how deletions are handled:

1) Adjacency list intersection in the hope of finding two-hop connecting paths. A



(a) $thresh_{PN} = 4$ (b) $thresh_{PN} = 6$ (c) $thresh_{PN} = 8$ (d) $thresh_{PN} = 12$

Figure 39: Average number of inserts and deletes in PN array for batches of $100K$ updates for RMAT-22 graphs. The subfigures are for different values of $thresh_{PN}$. Note that the ordinate is dependent on the specific bar chart. The charts for RMAT-21 graphs had very similar structure and have been removed for the sake of brevity.

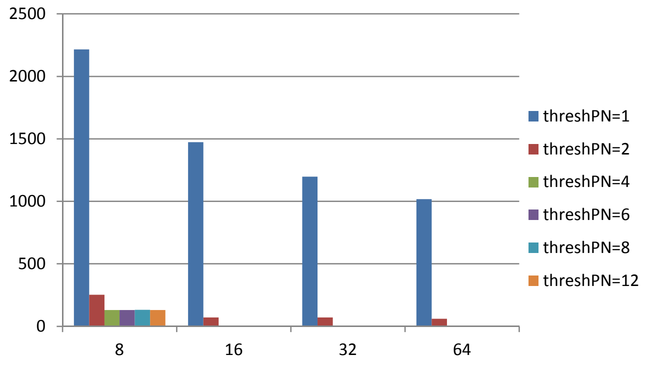


Figure 40: Average number of unsafe deletes in PN data structure for batches of $100K$ updates as a function of the average degree (x-axis) and $thresh_{PN}$ (bars).

similar approach with reasonable performance was shown in [56]; however, at batches of $100k$ it produces 750 unsafe deletes on average, thus requiring a full static recompute.

2) Maintain a spanning tree for each component. If a deleted edge is not in a tree, then it is considered safe. If the edge is in the tree, then the tree and affected components must be recomputed. In our experiments this approach is able to mark 90% of all deletions as safe.

3) Maintain two independent spanning trees for each connect component. Simply, find a spanning tree T, remove T from G to create G', and find a second spanning tree T' in G'. Deletions are safe until a vertex has no parent in either tree. When this occurs, the trees are recomputed from scratch. This approach is able to mark 99.7% of the deletes as safe, but this is not enough. This approach is also computationally demanding relative to others.

4) Attempting a BFS from one or both vertices to find a path between them. Given the low diameter, power law distribution, and large single component tendency, this can quickly encompass the entire component and most of the graph.

6.4 Results

We present both quantitative and performance results. In the quantitative results, we count how many deletions removed relationships from the PN sub-graph, how many insertions resulted in new relationships being added to the PN sub-graph, and how many insertions resulted in a new parent replacing an existing neighbor. We also track the number of deletions reported as unsafe. In the performance results, we show speedups over static re-computation, strong scaling, and overhead given as a fraction of the total update time spent maintaining the metric.

6.4.1 Quantitative

Fig. 39 depicts quantitative results for $thresh_{PN}$ of 4, 6, 8, and 12 at different graph sizes. For a specific $thresh_{PN}$, different edge factors were tested from $E = 8$ to 64; these are the abscissa. Due to the similarity of the results for R-MAT-21 and R-MAT-22, we present charts only for the R-MAT-22 graphs.

We observe a trend that a decreasing number of deletions and insertions affecting the PN sub-graph as the graph becomes denser. This is due to the fact that the fixed-size sub-graph covers a smaller fraction of the total edges. The number of neighbor replacements that occurs steadily becomes greater than the number of inserted neighbors and inserted parents. Based on Leskovec *et al.* [100], graph densification causes a shrinkage in the graph diameter. As such, in the initial data structure creation, the number of parents that a vertex has goes up (on average) leading to fewer replacements as edges are added. It can be further inferred that as the graphs become denser $E > 64$, there will be even fewer updates made to PN.

Looking across all of the subfigures, we see that the number of insertions of parents (purple bars), neighbors (green bars), and parents replacing neighbors (light blue) increases with $thresh_{PN}$ for a fixed average degree.

As the graphs become denser, it is more likely that a deleted edge is in the PN sub-graph. This can reduce performance due to the extra work in checking for disconnections. Moreover, as the $thresh_{PN}$ increases, we see more insertions and deletions into the PN sub-graph as expected. However, the number of unsafe deletes is significantly small, meaning that the data structure does not require significant repairs.

Fig. 40 shows the number of unsafe deletions marked as a function of the average degree for different $thresh_{PN}$ for the R-MAT-22 graph. The figure for the R-MAT-21 graph is similar to R-MAT-22 and has been omitted. For any given density, there are fewer unsafe deletes as the value of $thresh_{PN}$ increases. It is clear that using $thresh_{PN} = 1$ or $thresh_{PN} = 2$ is simply ineffective. For $thresh_{PN} = 1$, each deletion from the data structure becomes an unsafe delete. Increasing $thresh_{PN}$ beyond 4 gives significantly diminished returns in terms of he number of deletes marked unsafe. For this reason, we chose $thresh_{PN} = 4$ for our performance results.

6.4.2 Performance

In this section we present performance analysis of the parent-neighbor sub-graph approach. We demonstrate the scalability of our approach, compare performance versus a parallel static recomputation after each batch, and examine the effect of including updates to the parent-neighbor sub-graph in the update cycle.

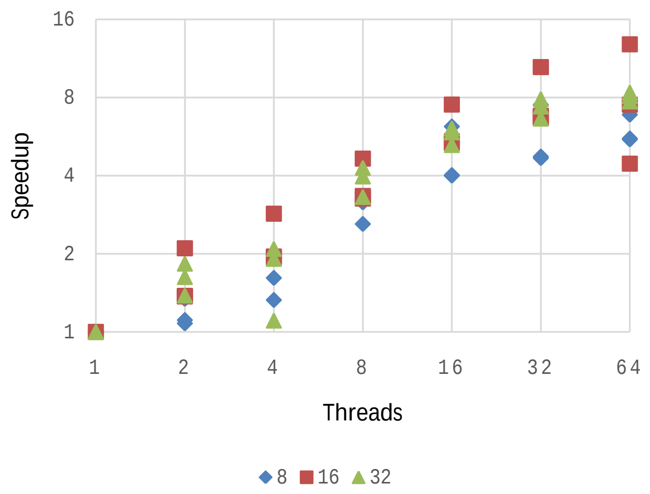


Figure 41: Strong scaling results on RMAT-22 graphs with different average degree as a function of the number of threads. Results include three graphs at each average degree.

We compare our new algorithm to a parallel implementation based on the Shiloach-Vishkin [129] algorithm that has been experimentally determined to perform better than traditional BFS on R-MAT graphs stored the STINGER data structure. Although this implementation is not the most work-efficient, it is scalable and highly parallel with good workload balance, has low synchronization costs, and performs well for graphs with low diameter. As a reminder from the previous subsection, our results use $thresh_{PN} = 4$.

Fig. 41 gives strong scaling results (holding the amount of work constant while increasing the number of threads/cores) for our algorithm on nine different R-MAT-22 graphs, three different graphs for each edge factor (8, 16, and 32). The threads are increased in multiples of 2 from 1 to 64. The plot shows nearly linear but not optimal scaling up to 32 threads in comparison with a single thread. The speedup is $10.5x$ at 32 threads and $12.8x$ at 64 threads.

Looking across the average adjacencies, the trend is similar. At higher thread counts, increasing density results in slight improvements in the average speedup. This is due to a combination of increases in the amount of work that can be performed in

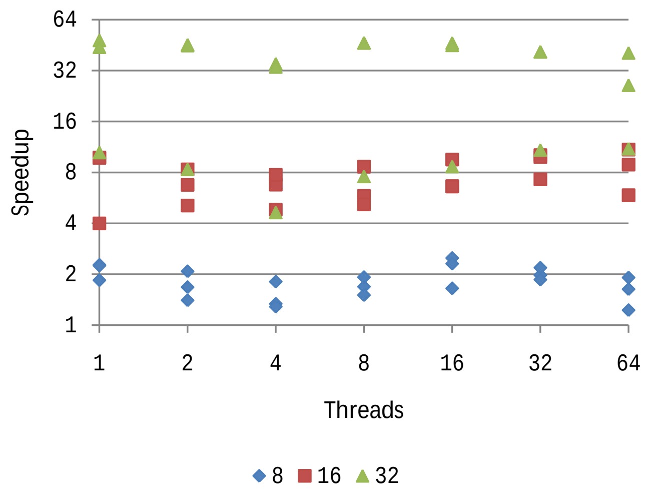


Figure 42: Speed up of the new algorithm over performing parallel static recomputation after each batch on three different RMAT-22 graphs with each average degree as a function of the number of threads.

parallel and fewer joined and broken components at higher densities.

Fig. 42 shows the performance improvement of using the parent-neighbor subgraph approach over recalculating the connected components using the parallel static Shiloach-Vishkin implementation after each batch. This is shown for multiple graphs with different average degrees and is also shown at each thread count. We show that the speedup ranges from an average of 1.8x for an average degree of 8 up to 30.8x for an average degree of 32 with a maximum of 48.3x. A key insight in this graph is that the implementation of our algorithm on STINGER and our implementation of static connected components on STINGER have the same scalability. This can be inferred from that fact that the ratio between the time of the PN update and the static recomputation remains constant as the thread count is increased.

In Fig. 43, a similarly equal scalability is shown. This graph shows the percentage of the time taken to perform the PN updates in each update cycle (where the full time for the cycle also includes updating the STINGER graph structure itself). The fraction of the time taken by the PN updates at a given edge factor remains constant as the thread count is increased. As the density increases, the updates cost less

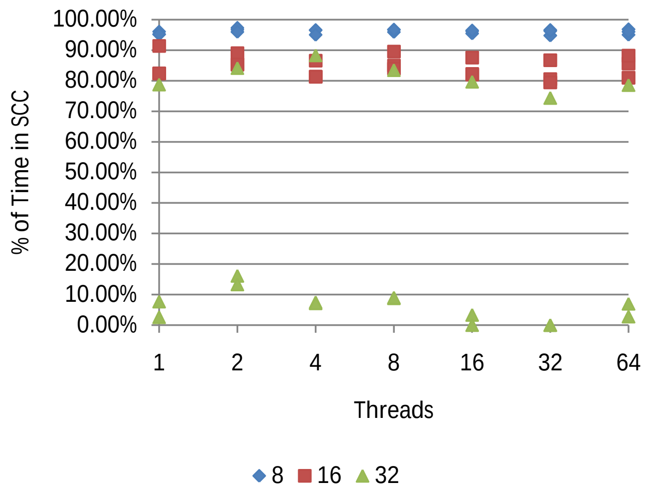


Figure 43: Fraction of the update time spent updating connected components over time spent updating the graph structure and connected components.

time due to components splitting and merging less frequently. This is evident in the increased speedup in Fig. 42. The static cost remains constant regardless of how often components merge or split, but our approach becomes faster. At the same time, updating the data structure has increasing cost with increasing density due to the greater number of edges per vertex that must be traversed to insert or remove and edge. More information on the implementation, performance and scalability of updates in STINGER can be found in [55].

Fig. 44 shows speedup over performing static recomputation after each batch for scale 24 graphs at edge factors up to 32 using 64 threads. We see a similar speedup trend to scale 22. The variance across the graphs of the same size shows that our algorithm is more sensitive to the structure of the graph and which edges are inserted and deleted while the static algorithm is extremely consistent and load balanced. The graph also shows that denser graphs give much better results, with the third scale 24 edge factor 16 graph performing 1.26 million updates per second while tracking connected components - 137x faster than static recomputation.

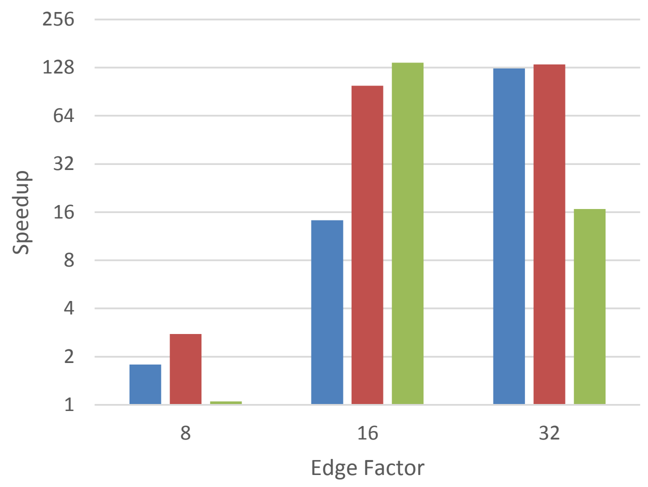


Figure 44: Speed up over performing static recomputation after each batch on scale 24 graphs for three graphs at each edge factor using 64 threads.

6.5 Conclusions

In this work we presented a novel parallel low-memory algorithm and data structure for maintaining a labeling of the connected components in a dynamic graph. We have shown that the algorithm performs well on sparse graphs and that by tracking only a few edges per vertex ($thresh_{PN}$) the number of unsafe deletes is reduced resulting in high performance. We have shown that the new dynamic graph algorithm outperforms a well-known static algorithm and that it has the same parallel scalability. Further, we have shown good strong scaling results despite our algorithm containing some sections with only fine-grain parallelism.

Beamer *et al.* [26] have shown a BFS algorithm that searches from the undiscovered vertices once half of all vertices have been found. This outperforms traditional BFS due to a large number of edge traversals in the traditional BFS that do not find new vertices. Given that our algorithm uses a BFS in both the initial stage and the streaming stage, an efficient implementation of the Beamer algorithm for STINGER data structure should be investigated.

CHAPTER VII

MERGE PATH

This chapter is an extension of the paper: S. Odeh, O. Green, Z. Mwassi, O. Shmueli, Y. Birk, " Merge Path - Parallel Merging Made Simple", Multithreaded Architectures and Applications (MTAAP) Workshop, IEEE 26th International Parallel & Distributed Processing Symposium (IPDPS), 2012.

Merging two sorted arrays is a prominent building block for sorting and other functions. Its efficient parallelization requires balancing the load among compute cores, minimizing the extra work brought about by parallelization, and minimizing inter-thread synchronization requirements. Efficient use of memory is also important.

We present a novel, visually intuitive approach to partitioning two input sorted arrays into pairs of contiguous sequences of elements, one from each array, such that 1) each pair comprises any desired total number of elements, and 2) the elements of each pair form a contiguous sequence in the output merged sorted array. While the resulting partition and the computational complexity are similar to those of certain previous algorithms, our approach is different, extremely intuitive, and offers interesting insights. Based on this, we present a synchronization-free, cache-efficient merging (and sorting) algorithm.

While we use a shared memory architecture as the basis, our algorithm is easily adaptable to additional architectures. In fact, our approach is even relevant to cache-efficient sequential sorting. The algorithms are presented, along with important cache-related insights.

7.1 Introduction

Merging two sorted arrays, A and B, to form a sorted output array S is an important utility, and is the core the of merge-sort algorithm [45] . Additional uses include joining the results of database queries and merging adjacency lists of vertices in graph contractions.

The merging (e.g., in ascending order) is carried out by repeatedly comparing the smallest (lowest-index) as-yet unused elements of the two arrays, and appending the smaller of those to the result array.

Given an (unsorted) N-element array, merge-sort comprises a sequence of $\log_2 N$ merge rounds: in the first round, *N/2* disjoint pairs of adjacent elements are sorted, forming *N/2* sorted arrays of size two. In the next round, each of the *N/4* disjoint pairs of two-element arrays is merged to form a sorted 4-element array. In each subsequent round, array pairs are similarly merged, eventually yielding a single sorted array.

Consider the parallelization of merge-sort using p compute cores (or processors or threads, terms that will be used synonymously). Whenever $N \gg p$, the early rounds are trivially parallelizable, with each core assigned a subset of the array pairs. This, however, is no longer the case in later rounds, as only few such pairs remain. Consequently and since the total amount of computation is the same in all rounds, effective parallelization requires the ability to parallelize the merging of two sorted arrays

An efficient Parallel Merge algorithm must have several salient features, some of which are required due to the very low compute to memory-access ratio: 1) equal amounts of work for all cores; 2) minimal inter-core communication (platform-dependent ramifications); 3) minimum excess work (for parallelizing, as well as replication of effort); and 4) efficient access to memory (high cache hit rate and minimal cache-coherence overhead). Coherence issues may arise due to concurrent access to the same address, but also due to concurrent access to different addresses in the same cache line (false sharing). Memory issues have platform-dependent manifestations.

A naïve approach to parallel merge would entail partitioning each of the two arrays into equal-length contiguous sub-arrays and assigning a pair of same-size sub arrays to each core. Each core would then merge its pair to form a single sorted array, and those would be concatenated to yield the final result. Unfortunately, this is incorrect. (To see this, consider the case wherein all the elements of A are greater than all those of B.) So, correct partitioning is the key to success.

In this chapter, we present a parallel merge algorithm for Parallel Random Access Machines (PRAM), namely shared-memory architectures that permit concurrent (parallel) access to memory. PRAM systems are further categorized as CRCW, CREW, ERCW or EREW, where C, E, R and W denote concurrent, exclusive, read and write, respectively. Our algorithm assumes CREW, but can be adapted to other variants. Also, complexity calculations assume equal access time of any core to any address, but this is not a requirement.

Our algorithm is load-balanced, lock-free, requires a negligible amount of excess work, and is extended to a memory-efficient version. Being lock-free, the algorithm does not rely on a set of atomic instructions of any particular platform and therefore can be easily applied. The efficiency of memory access is also not confined to one kind of architecture; in fact, the memory access is efficient for both private- and shared-cache architectures.

We show a correspondence between the merge operation and the traversal of a path on a grid, from the upper left corner to the bottom right corner and going only rightward or downward. This greatly facilitates the comprehension of parallel merge algorithms. By using this path, dubbed *Merge Path*, one can divide the work equally among the cores. Most important, we parallelize the partitioning of the merge path.

Our actual basic algorithm is similar to that of [50] , but is more intuitive and

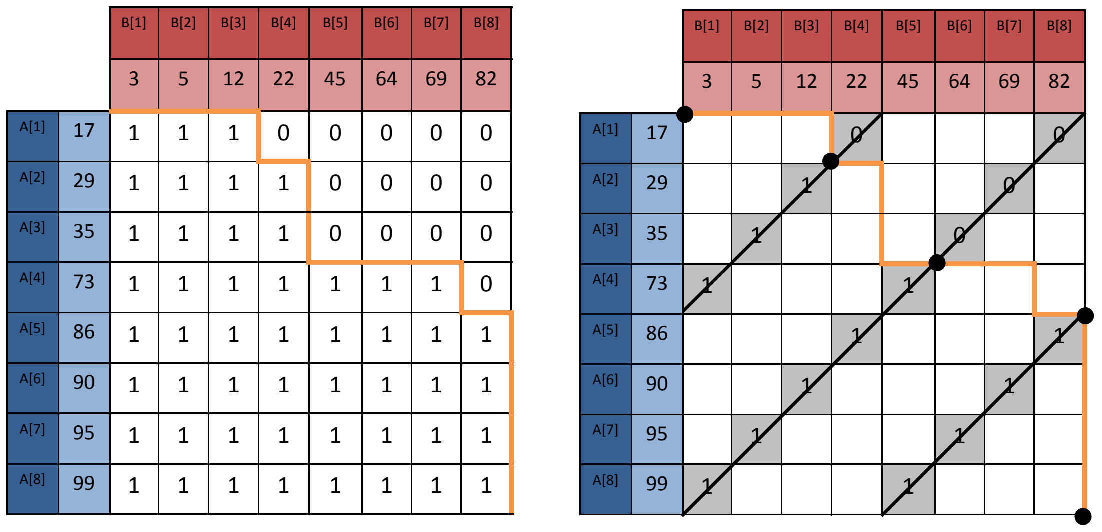


Figure 45: Merge Matrix and Merge Path. (a) The Merge Matrix is shown with all the values explicitly computed. The Merge Path is on the boundary between the zeros and the ones. (b) The cross diagonals in a Merge Matrix are used to find the points of change between the ones and the zeros, i.e., the intersections with the Merge Path.

conceptually simpler. Furthermore, using insights from the aforementioned geometric correspondence, we develop a new cache-efficient merge algorithm, and use it for a memory-efficient parallel sort algorithm.

The remainder of the chapter is organized as follows. In Section II, we present the Merge Path, the Merge Matrix and the relationship between them. These are used in Section III to develop parallel merge and sort algorithms. Section IV introduces cache-related issues and presents a cache-efficient merge algorithm. Section V discusses related work and Section VI presents experimental results of our two new parallel algorithms on two systems.Section VII offers concluding remarks.

7.2 Merge Path

7.2.1 Construction and basic properties

Consider two sorted arrays, A and B, with */A/* and */B/* elements, respectively. (The lengths of the arrays may differ: $/A/ \neq /B/$) Without loss of generality, assume that they are sorted in ascending order. As depicted in Fig. 1 (a) (ignore the contents of the matrix), create a column comprising A's elements and a Row comprising B's elements, and an $|A|x|B|$ matrix M, each of whose rows (columns) corresponds to an element of A (B). We refer to this matrix as the *Merge Matrix.* A formal definition and additional details pertaining to M will be provided later.

Next, let us merge the two arrays: in each step, pick the smallest (regardless of array) yet-unused array element. Alongside this process, we construct the *Merge Path.* Referring again to Fig. 1 (a), start in the upper left corner of the grid, i.e., at the upper left corner of *M[1,1].* If *A[1]>B[1]*, move one position to the right*;* else move one position downward. Continue similarly as follows: consider matrix position *(i,j)* whose upper left corner is the current end of the merge path: if *A[i]>B[j]*, move one position to the right and increase j*;* else move one position downward and increase i; having reached the right or bottom edge of the grid, proceed in the only possible direction. Repeat until reaching the bottom right corner.

The following four lemmas follow directly from the construction of the Merge Path:

Lemma 1: Traversing a Merge Path from beginning to end, picking in each rightward step the smallest yet-unused element of *B,* and in each downward step the smallest yet-unused element of A, yields the desired merger.

■

Lemma 2: Any contiguous segment of a Merge Path is composed of a contiguous sequence of elements of A and of a contiguous sequence of elements of B. ■

Lemma 3: Non-overlapping segments of a merge path are composed of disjoint sets of elements, and vice versa. ■

Lemma 4: Given two non-overlapping segments of a merge path, all array elements composing the later segment are greater than or equal to all those in the earlier segment. ■

Theorem 5: Consider a set of element-wise disjoint sub-array pairs (one, possibly empty sub-array of A and one, possibly empty sub-array of B), such that each such pair comprises all elements that, once sorted, jointly form a contiguous segment of a merge path. It is claimed that these array pairs may be merged in parallel and the resulting merged sub-arrays may be concatenated according to their order in the merge path to form a single sorted array.

Proof: By Lemma 1, the merger of each sub-array pair forms a sorted sub-array comprising all the elements in the pair. From Lemma 2 it follows that each such sub-array is composed of elements that form a contiguous sub-array of their respective original arrays, and by Lemma 3 the given array pairs correspond to non-overlapping segments of a merge path. Finally, by Lemma 4 and the construction order, all elements of a higher-indexed array pair are greater than or equal to any element of a lower-indexed one, so concatenating the merger results yields a sorted array. ■

Corollary 6: Any partitioning of a given Merge Path of input arrays A and B into non-overlapping segments that jointly comprise the entire path, followed by the independent merger of each corresponding sub-array pair and the concatenation of the results in the order of the corresponding Merge-Path segment produces a single sorted array comprising all the elements of A and B. ■

Corollary 7: Partitioning a Merge Path into equisized segments and merging the corresponding array pairs in parallel balances the load among the merging processors.

Proof: each step of a Merge Path requires the same operations (read, compare and write), regardless of the outcome. ■

Equipped with the above insights, we next set out to find an efficient method for partitioning the Merge Path into equal segments. The challenge, of course, is to do so without actually constructing the Merge Path, as its construction is equivalent to carrying out the entire merger. Once again using the geometric insights provided by Fig. 1 (b), we begin by exposing an interesting relationship between positions on any Merge Path and cross diagonals (ones slanted upward and to the right) of the Merge Matrix *M*. Next, we define the contents of a Merge Matrix and expose an interesting property involving those. With these two building blocks at hand, we construct a simple method for parallel partitioning of any given Merge Path into equisized segments. This, in turn, enables parallel merger.

7.2.2 The Merge Path and cross diagonals

Lemma 8: Regardless of the route followed by a Merge Path, and thus regardless of the contents of A and B, the i'th point along a Merge Path lies on the i'th cross diagonal of the grid and thus of the Merge Matrix *M*.

Proof: each step along the Merge Path is either to the right or downward. In either case, this results in moving to the next cross diagonal. ■

Theorem 9: Partitioning a given merge path into p equisized contiguous segments is equivalent to finding its intersection points with *p-1* equispaced cross diagonals of *M*,

Proof: follows directly from Lemma 8. ■

7.2.3 The Merge Matrix – content & properties

Definition 1: A binary merge matrix M of A, B is a Boolean two dimensional matrix of size $|A| \times |B|$ such that:

$$M[i,j] = \begin{cases} 1 & A[i] > B[j] \\ 0 & otherwise \end{cases} .$$

Proposition 10: Let M be a binary merge matrix. Then, $M[i,j] = 1 \Rightarrow \forall k, m : i \le k \le |A| \wedge 1 \le m \le j,\ M[k,m] = 1$

Proof: If $M[i,j] = 1$ then according to definition 1, $A[i] > B[j]$. $k \geq i \Rightarrow A[k] \geq A[i]$ (A is sorted). $j \geq m \Rightarrow B[j] \geq B[m]$ (B is sorted). $A[k] \geq A[i] > B[j] \geq B[m]$ and according to definition 1, $M[k,m] = 1$. ■

Proposition 11: Let M be a binary merge matrix. If $M[i,j] = 0$, then $\forall k, m \ s.t. (1 \leq k < i) \wedge (j \leq m \leq |B|)$, $M[k,m] = 0$.

Proof: Similar to the proof of proposition 10. ■

Corollary 12. The entries along any cross diagonal of M form a monotonically non-increasing sequence. ■

7.2.4 The Merge Path and the Merge Matrix

Having established interesting properties of both the Merge Path and the Merge Matrix, we now relate the two, and use *P(M)* to denote the Merge Path corresponding to Merge Matrix *M*.

Proposition 13: Let (i,j) be the highest point on a given cross diagonal M such that $M[i,j-1] = 1$ if exists, otherwise let (i,j) be the lowest point on that cross diagonal. Then, $P(M)$ passes through (i,j). This is depicted in Fig 2.

Proof: by induction on the points on the path.

Base: The path starts at $(1,1)$. The cross diagonal that passes through $(1,1)$ consists only of this point; therefore, it is also the lowest point on the cross diagonal.

Induction step: assume the correctness of the claim for all the points on the path up to point (i,j). Consider the next point on $P(M)$. Since the only permissible moves are *Right* and *Down*, the next point can be either $(i,j+1)$ or $(i+1,j)$, respectively.

Case 1: *Right* move. The next point is $(i,j+1)$. According to Definition 1, $M[i,j] = 1$. According to the induction assumption, either $i = 1$ or $M[i-1,j] = 0$. If $i = 1$ then the new point is the highest point on the new cross diagonal such that $M[i,j] = 1$. Otherwise, $M[i-1,j] = 0$. According to Proposition 11, $M[i-1,j+1] = 0$. Therefore, $(i,j+1)$ is the highest point on its cross diagonal at

Algorithm 12: $ParallelMerge(A, B, S, p)$

```
for $i = 1$ to $p$ parallel do
    $DiagonalNum \leftarrow (i-1) \cdot (|A| + |B|)/p + 1$
    $length \leftarrow (|A| + |B|)/p$
    $a_{i,start}, b_{i,start} \leftarrow$ Diagonal Intersection (A,B,i,p) // Algorithm 13
    $s_{i,start} \leftarrow (i-1) \cdot (|A| + |B|)/p + 1$
    $Merge(A, a_{i,start}, B, b_{i,start}, S, s_{i,start}, length)$
Barrier
```

Algorithm 13: $DiagonalIntersection(A, B, thread_{id}, p)$ - Algorithm for finding intersection of the Merge Path and the cross diagonals.

```
$diag \Leftarrow i \cdot (|A| + |B|)$ / $p$
$a_{top} \Leftarrow diag > |A|$ ? $|A| : diag$
$b_{top} \Leftarrow diag > |A|$ ? $diag - |A| : 0$
$a_{bottom} \Leftarrow b_{top}$
while true do
    $offset \Leftarrow (a_{top} - a_{bottom})/2$ $a_i \Leftarrow a_{top} - offset$ $b_i \Leftarrow b_{top} + offset$ if
    $A[a_i] > B[b_i - 1]$ then
        if $A[a_i - 1] \leq B[b_i]$ then
            $a_{start} \Leftarrow a_i$
            $b_{start} \Leftarrow b_i$
            break
        else
            $a_{top} \Leftarrow a_i - 1$
            $b_{top} \Leftarrow b_i + 1$
    else
        $a_{bottom} \Leftarrow a_i + 1$
return $\{a_{start}, b_{start}\}$
```

which $M[i,j] = 1$.

Case 2: the move was *Down*, so the next point is $(i+1, j)$. According to Definition 1, $M[i,j] = 0$. According to the induction assumption, either $j = 1$ or $M[i, j-1]$ =1. If $j = 1$ then the new point is the lowest point in the new cross diagonal. Since $M[i,j] = 0$ and according to Proposition 11, the entire cross diagonal is 0. Otherwise, $M[i, j-1] = 1$. According to Proposition 10, $M[i+1, j-1] = 1$. Therefore, $(i, j+1)$ is the highest point on its cross diagonal at which $M[i+1, j-1] = 1$. ■

Theorem 14: Given sorted input arrays A and B, they can be partitioned into p pairs of sub-arrays corresponding to p equisized segments of the corresponding merge path. The *p-1* required partition points can be computed independently of one another (optionally in parallel), in at most $\log_2(\min(|A/,/B/))$ steps per partition point, with neither the matrix nor the path having to actually be constructed.

Proof: According to Theorem 9, the required partition points are the intersection points of the Merge Path with *p-1* equispaced (and thus content-independent) cross diagonals of *M*. According to Corollary 12 and Proposition 13, each such intersection point is the (only) transition point between '1's and '0's along the corresponding cross diagonal. (If the cross diagonal has only '0's or only '1's, this is the uppermost and the lowermost point on it, respectively.) Finding this partition point, which is the intersection of the path and the cross diagonal, can be done by way of a binary search on the cross diagonal, whereby in each step a single element of A is compared with a single element of *B*. Since the length of a cross diagonal is at most $\min(|A/,/B/)$, at most $\log_2(\min(|A/,/B/))$ steps are required. Finally, it is obvious from the above description that neither the Merge Path nor the Merge Matrix needs to be constructed and that the *p-1* intersection points can be computed independently and thus in parallel. ∎

7.3 Parallel Merge and Sort

Given two input arrays A and B parallel merger is carried out by p processors according to Algorithm 1. For the sake of brevity we do not present the pseudo code for a sequential merge.

Remark. Note that no communication is required among the cores: they write to disjoint sets of addresses and, with the exception of reading in the process of finding the intersections between the Merge Path and the diagonals, read from disjoint addresses. Whenever $/A/+/B/>>p$, which is the common case, this means that concurrent reads from the same address are rare.

Summarizing the above, the time complexity of the algorithm for $|A| + |B| = N$ and p processors is $O\left(N/p + \log\left(N\right)\ \right)$, and the work complexity is $O(N + p \cdot \log N\ )$. For $p < N/log(N)$, this algorithm is considered to be optimal. In the next section, we address the issue of efficient memory (cache) utilization.

Finally, merge-sort can be employ Parallel Merge to carry out each of $\log_2 N$ rounds. The rounds are carried out one after the other.

The time complexity of this Parallel Merge-Sort is: $O(N/p \cdot log(N/p) + N/p \cdot log(p) + log(p) \cdot log(N) = O(N//p \cdot log(N) + log(p) \cdot log(n)$

In the first expression, the first component corresponds to the sequential sort carried out concurrently by each core on N/p input elements, and the two remaining ones correspond to the subsequent rounds of parallel merges.

7.4 Cache Efficient Merge Path

In the remainder of this section, we examine the cache efficiency issue in conjunction with our algorithm, offering important insights, exploring trade-offs and presenting our approaches. Before continuing along this path, however, let us digress briefly to discuss relevant salient properties of hierarchical memory in general, and particularly in shared-memory environments. We have kept this discussion short and only present relevant properties of the cache. For additional information on caches we refer the reader to Stenstrom [132] for a brief survey on cache coherence, Conway *et al.* [43] for cache design considerations for modern caches, and Lam *et al.* [96] for performance of blocked algorithms that take the cache into consideration.

7.4.1 Overview

The rate at which merging and sorting can be performed even in memory (as opposed to disk), is often dictated by the performance of the memory system rather than by processing power. This is due to the fact that these operations require a very small amount of computing per unit of data, and the fact that only a small amount of memory, the cache, is reasonably fast. (The next level in the memory hierarchy typically features a ten-fold higher access latency as well as coarser memory-management granularity.) Parallel implementation on a shared memory system further aggravates the situation for several reasons: 1) the increased compute power is seldom matched by a commensurate increase in memory bandwidth, at least beyond the 1^{st}-level or 2^{nd}-level cache, 2) the cores potentially share a 2^{nd} or 3^{rd} level cache, and 3) cache coherence mechanisms can present an extremely high overhead. In this section, we address the memory issues.

Assuming large arrays (relative to cache size) and merge-sort, it is clear that data will have to be brought in multiple times ($\log_2 N$ times, one for each level of the merge tree), so we again focus on merging a pair of sorted arrays.

7.4.2 Memory-hierarchy highlights

Cache Organization and management

Unlike software-managed buffers, caches do not offer the programmer direct control over their content and, specifically, over the choice of item for eviction. Furthermore, in order to simplify their operation and management, caches often restrict the locations in which an item may reside based on certain bits of its original address (the index bits). The number of cache locations at which an item with a given address may reside is referred to as the level of associativity: in a fully associative cache there are no restrictions; at the other extreme, a direct-mapped cache permits any given address to be mapped only to a single specific location in the cache. The collection of cache locations to which a given address may be mapped is called a set, and the size of the set equals the degree of associativity.

Whenever an item must be evicted from the cache in order to make room for a new one, the cache management system must select an item from among the members of the relevant set. One prominent replacement policy is least recently used (LRU), whereby the evicted item is the set member that was last accessed in the most distant pass. Another is first in – first out (FIFO), whereby the evicted item is the one that was last brought into the cache in the most distant past. Additional considerations may include eviction of pages that have not been modified while in the cache, because they often don't have to be copied to the lower level in the hierarchy, as it maintains a copy (an inclusive cache hierarchy).

Cache content is managed in units of cache line. We will initially assume that the size of an array item is exactly one cache line, but will later relax this assumption.

Cache performance

The main cache-performance measure is the hit rate, namely the fraction of accesses that find the desired data in the cache. (Similarly, miss rate = 1- hit rate.)

There are three types of cache misses: Compulsory, Capacity, and Contention [79].

1) Compulsory – a miss that occurs upon the first request for a given data item. (Whenever multiple items fit in a cache line, as well as when automatic prefetching is used, the compulsory miss rate may be lower than expected. Specifically, access to contiguous data would result in one miss per cache line or none at all, respectively.)

2) Capacity – this refers to cache misses that would have been prevented with a larger cache.

3) Conflicts – these misses occur despite sufficient cache capacity, due to limited flexibility in data placement (limited associativity and non-uniform use of different sets).

Remark: even if cache misses can be reduced by appropriate policies, one must also consider the total communication bandwidth between the cache and the lower level. Specifically, data prefetching can mask latency but does not reduce the mean bandwidth requirement, and speculative prefetching may actually increase it.

Cache coherence

In multi-core shared memory systems with private caches, yet another complication arises from the fact that the same data may reside in multiple private caches (for reading purposes), yet coherence must be ensured when writing. There are hardware cache-coherence mechanisms that obviate the programmer's need to be concerned with correctness; however, the frequent invocation of these mechanisms can easily become the performance bottleneck. The most expensive coherence-related operations occur when multiple processors attempt to write to the same place. The fact that management and coherence mechanisms operate at cache-line granularity complicates matters, as coherence-related operations may take place even when cores access different addresses, simply because they are in the same cache line. This is known as false sharing.

Cache replacement policy

A problem may arise at replenishment time. Consider, for example, LRU and a situation wherein a given merge segment only comprises elements of A. As replenishment elements are brought in to replace the used elements of A, the least recently used elements are actually those of B, as both the A element positions and the result element positions were accessed in the previous iteration whereas only one element of B was accessed (it repeatedly "lost" in the comparison). A similar problem occurs with a FIFO policy.

A possible solution for LRU is, prior to fetching replenishment elements, touching all cache lines containing unused input elements. If each cache line only contains a single item, this would represent approximately a 50% overhead in cache access (the usual comparison is between the loser of the previous comparison, which is in a register, and the next element of the winning array, which must be read from cache; also, the result must be written. So the number of cache accesses per step grows from 2 to 3.) If there are multiple elements per cache line, the overhead quickly becomes negligible.

Limited associativity

Proposition 15. With 3-way associativity or higher, conflict misses can be avoided.

Proof: Consider a cache of size C. With k-way associativity, any C/k consecutive addresses are mapped to C/k different sets. We partition the merge path into segments of size $C/3$, thus constructing the merged array in segments of $C/3$ elements. The corresponding result array comprises at most $C/3$ elements of A and at most $C/3$ elements of B. (The actual numbers are data dependent.) The $C/3$ items of each of A, B and the merged array will take up exactly one position in each of the three sets, regardless of the start address of each of these element sequences. Similarly, each will take up to two positions in a 6-way set associative cache, three in a 9-way, etc. For associativity levels that are greater than 3 but not integer multiples thereof, one can reduce the segment length such that each array's elements occupy at most a safe number of positions in each set. (A safe number is one such that even if all three arrays occupy the maximum number of positions in a given set, this will not exceed the set size, i.e., the degree of associativity.) ■

7.4.3 Cache-Efficient Parallel Merge

In this sub-section we present an extension to our algorithm for parallel merging that is also cache-efficient and considers a shared-memory hierarchy (including a shared cache).

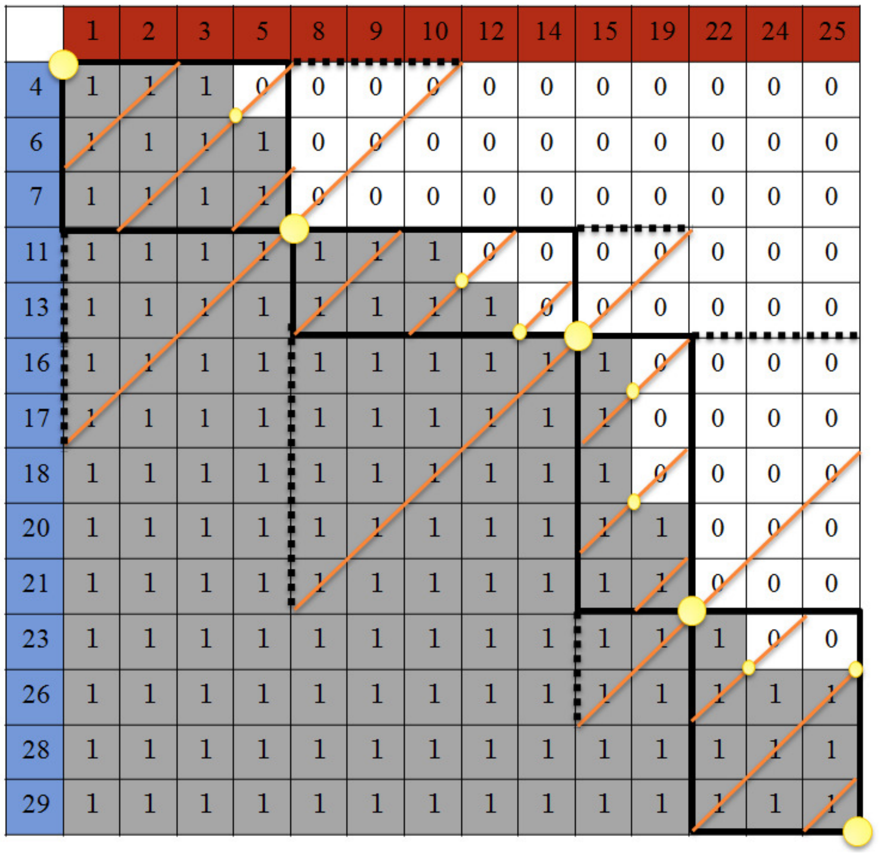


Figure 46: Merge Matrix for the cache efficient algorithm. The yellow circles depict the initial and final points of the path for a specific block in the cache algorithm.

Collisions in the cache between any two items are avoided when they are guaranteed to be able to reside in different cache locations, as well as when they are guaranteed to be in the cache at different times. In a Merge operation, a cache-resident item is usually required for a very short time, and is used only once. However, many items are brought into the cache. Also, the relative addresses of "active" items are data dependent. This is true among elements of different arrays (A, B, S) and, surprisingly, also among same-array elements accessed by different cores. This is because the segment-partition points in any given array are data dependent, as is the rate at which an array's elements are consumed.

Given our efficient parallelization, we are able to efficiently carry out parallel merger of even cache-size arrays. In view of this, we explore approaches that ensure that all elements that may be active at any given time can co-reside in cache.

Let C denote cache size (in elements). Our general approach is to break the overall merge path into cache-size (actually a fraction of that) segments, merging those segments one after the other, with the merging within each segment being parallelized. We refer to this as Segmented Parallel Merge (SPM). See Fig. 2. The pseudo code for SPM is given in Algorithm 14.

Lemma 16. A merge-path segment of length L comprises at most L consecutive elements of A and at most L consecutive elements of B. ■

Theorem 17. Given L consecutive elements of A and L consecutive elements of B, starting with the first element of each of them in the segment being constructed, one can compute in parallel the p segment starting points so as to enable p consecutive segments of length L/p to be constructed in parallel.

Proof: Consider the $p-1$ cross diagonals of the merge matrix comprising the aforementioned elements of the two arrays, such that the first one is L/p away from the upper left corner and the others are spaced with the same stride. The farthest cross diagonal will require the $L'th$ provided element from each of the two arrays, and no other point along any of the diagonals will require "later" elements. Also, since the farthest diagonal is at distance L from the upper left corner (Manhattan distance), the constructed segment will be of length L. ■

Remark. Unlike the case of a full merger of two sorted arrays of size L, not all elements will be used. While L elements will be consumed in the construction of the segment, the mix of elements from A and from B is data dependent.

In order to avoid the extra complexity of using the same space for input elements and for merged data, let $L = C/3$, where C is the cache size.

Algorithm 14: $SegmentedParallelMerge(A, B, S, p, C)$

```
$L \leftarrow C/3$
$length \leftarrow L/p$
$MAX_{iterations} \leftarrow 3 \cdot (|A| + |B|)/C$
$startingPoint \leftarrow$ top left corner
for $k = 1$ to $MAX_{iterations}$ do
    for $i = 1$ to $p$ parallel do
        $DiagonalNum \leftarrow (i-1) \cdot (L)/p + 1$ $length \leftarrow (L)/p$ $a_{i,start}, b_{i,start} \leftarrow$ Compute intersection of the Merge path with DiagonalNum using a binary search $s_{i,start} \leftarrow startingPoint + (i-1) \cdot (L)/p + 1$
        $Merge(A, a_{i,start}, B, b_{i,start}, S, s_{i,start}, length)$
    if $k = p$ then
        update $startingPoint$
    Barrier
```

Remark. Sufficient total cache size does not guarantee collision freedom (conflict misses can occur). However, we have shown that 3-way associativity suffices to guarantee collision freedom.

Computational complexity Assuming a total merged-array segment size of $L = C/3$ per sequential iteration of the algorithm, there are $3N/C$ such iterations. In each of those, at most $2L = 2C/3$ elements of the input arrays (L of each) need to be considered in order to determine the end of the segment and, accordingly, the numbers of elements that should be copied into the cache. Because the sub-segments of this segment are to be created in parallel, each of the p cores must compute its starting points (in A and in B) independently. (We must consider $2L$ elements because the end point of the segment, determined by the numbers of elements contributed to it by A and B, is unknown.)

The computational complexity of the cache-efficient merge of N elements given a cache of size C and p cores is:$\mathrm{O}\left(\mathrm{N/C} \cdot \mathrm{p} \cdot \mathrm{logC} + \mathrm{N}\right)$.

Normally, $p << C << N$, in which case this becomes $O(N)$. In other words, the parallelization overhead is negligible.

The time complexity is $\mathrm{O}\left(\mathrm{N/C} \cdot (\mathrm{logC} + \mathrm{C/p})\right)$.

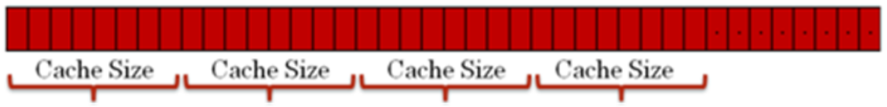


Figure 47: Cache-efficient parallel sort first stage. Each cache sized block is sorted followed by parallel merging

Neglecting $logC$ (the parallelization overhead) relative to C/p (the merge itself), this becomes $O(N/p)$, which is optimal. Finally, looking at typical numbers and at the actual algorithms, it is evident that the various constant coefficients are very small, so this is truly an extremely efficient parallel algorithm and the overhead of partitioning into smaller segments is insignificant.

7.4.4 Cache-Efficient Parallel Sort

Initially, partition the unsorted input array into equisized sub-arrays whose size is some fraction of the cache size C.

Next, sort them one by one using the parallel sort algorithm on all p processors as explained in an earlier section. (Of course, one may sort them in parallel, but this would increase the cache footprint.)

Finally, proceed with merge rounds; in each of those, the cache-efficient parallel merge algorithm is applied to every pair of sorted sub-arrays. This is repeated until a single array is produced.

We now derive the time complexity of the cache efficient parallel sort algorithm. We divide the complexity into two stages: 1) the complexity of the parallel sorting of the sub-arrays of at most Celements, and 2) the complexity of the cache-efficient merge stages.

In the first stage, depicted in Fig. 3, the parallel sort algorithm is invoked on the cache sized sub-arrays. The number of those sub-arrays is O (N/C). Hence, the time complexity of this stage is $O(N/C \cdot (C/p \cdot log(C) + log(p) \cdot \log(C)))$.

The second stage may be viewed as a binary tree of merge operations. The tree leaves are the sorted cache sized sub-arrays. Each two merged sub-arrays are connected to the merged sub-array, and so on. The complexity of each level in the tree is $O(log(N/C) \cdot (N/p + N/C \cdot log(p))$.

The total complexity of the cache-efficient parallel sort algorithm is the sum of the complexities of the two stages: $O(N/p \cdot log(N) + N/C \cdot log(p) \cdot log(C))$.

One may observe again that the new algorithm has a slightly higher complexity, $\mathrm{N/C} \cdot \log(\mathrm{C}) \cdot \log(\mathrm{p}) > \log \mathrm{N} \cdot \log(\mathrm{p})$, due to the numerous partitioning stages. However, this is beneficial for systems where a cache miss is relatively expensive.

7.5 Related work

In this section, we review previous works on the subjects of parallel sorting and parallel merging, and relate our work to them.

Prior works fall into two categories: 1) algorithms that use a problem-size dependent number of processors, and 2) algorithms that use a fixed number of processors.

Several algorithms have been suggested for parallel sorting. While parallel merge can be a building block for parallel sorting, some parallel sorting algorithms do not require merging. An example is Bitonic Sort [25] in which $\mathrm{O}\left(\mathrm{N} \cdot (\log \mathrm{N})^2\right)$ comparators are used (N/2 comparators are used in each stage) to sort N elements in $\mathrm{O}\left((\log \mathrm{N})^2\right)$ cycles. Bitonic sort falls into the aforementioned first category. Our work is in the latter.

We consider two complexity measures: 1) time complexity (the time required to complete the task), and 2) overall work complexity, i.e, the total number of basic operations carried out. In a load balanced algorithm like ours, the work complexity is the product of time complexity and the number of cores. Even with perfect load balancing, however, one must be careful not to increase the total amount of work (overhead, redundancy, etc.), as this would increase the latency. Similarly, one must be careful not to introduce stalls (e.g., for inter-processor synchronization), as these would also increase the elapsed time even if the “net” work complexity is not increased.

Merging two sorted arrays requires Ω(N) operations. Some of the parallel merging algorithms, including ours, have a work complexity of $O(N + p \cdot log(N))$, the latter component is negligible and the complexity is O(N), as observed in [12]. Also, there are no synchronization stalls in our algorithm.

In Shiloach and Vishkin [126], as in our work, a CREW PRAM memory model is used. There, a mechanism for partitioning the workload is presented. This mechanism is less efficient than ours and does not feature perfect load balancing; although each processor is responsible for merging O (N/p) elements on average, a processor may be assigned as many as 2N/p elements. This can introduce a stall to some of the cores since all the cores have to wait for the heaviest job. For truly efficient algorithms, namely ones in which the constants are also tight, as is the case with our algorithm, such a load imbalance can cause a 2X increase in latency! The time complexity of this algorithm is O (1+log p +log N +N/p). For N $\gg$ p, which is the case of interest, it is O (N/p+log N) .

In [12], Akl and Santoro present a merging algorithm that is memory-conflict free using the EREW model. It begins by finding one element in each of the given sorted arrays such that one of those two elements is the median (mid-point) in the output array. The elements found (A [i] , B [j]) are such that if A [i] is the aforementioned median then B [j] is the largest element of B that is smaller than A [i] or the smallest element of B that is greater than A [i]. Once this median point has been found, it is possible to repeat this on both sets of the sub-arrays. Their way of finding the median is similar to the process that we use yet the way the explain their approach is different. The complexity of finding the median is O(log (N)). As these arrays are non-overlapping, there will not be any more conflict on accessed data. This stage is repeated until there are p partitions. This requires O(log (p)) iterations. Once all the partitions have been found, it is possible to merge each pair of sub-arrays

Table 14: Cache misses for parallel merging algorithms assuming a 3-way associativity.

	Cache misses		
Algorithm	***Partitioning stage***	***Merge stage***	***Total***
[126]	$O(p \cdot log(N) + p \cdot log(p))$	$\Omega(N)$	$O(N+p \cdot log(N)+p \cdot log(p))$
[12]	$O(p \cdot log(N))$	$\Omega(N)$	$O(N + p \cdot log(N))$
[50] & Merge Path	$O(p \cdot log(N))$	$\Omega(N)$	$O(N + p \cdot log(N))$
Segmented Merge Path	$O(p \cdot N/C \cdot log(C))$	$\Theta(N)$	$\Theta(N)$

sequentially, concurrently for all pairs, and to simply concatenate the results to form the merged array. The overall complexity of this algorithm is $\mathrm{O}\left(\mathrm{N/p{+}log\,N\,log\,p}\right)$. The somewhat higher complexity is the price for the total elimination of memory conflicts.

In [50], Deo and Sarkar present an algorithm that is conceptually similar to that of [12] is presented. They initially present an algorithm that finds one element in each of two given sorted arrays such that one of these elements is $\mathrm{k-th}$ smallest element in the output (merged) array. In [12] they start off by finding $\mathrm{k = N/2}$. In [50], the elements sought after are those that are equispaced (N/p positions apart) in the output array. Finding each of these elements has the complexity of $\mathrm{O}\left(\mathrm{log}\left(\mathrm{N}\right)\right)$. This algorithm is aimed for CREW systems. The complexity of this algorithm is $\mathrm{O}\left(\mathrm{N/p{+}log}\left(\mathrm{N}\right)\right)$.

Our algorithm is very similar to the one presented in [50]. However, our approach is different in that we show a correspondence between finding the desired elements and finding special points on a grid. Finally, using this correspondence along with additional insights and ideas, we also provide cache efficient algorithms for parallel merging and sorting that did not appear in any of the related works.

The work done in [49] is an extension of [50], in which the algorithm is adapted to an EREW machine with a slightly larger complexity of $O(N/P + log(N) + log(P))$ due to the additional constraints of the EREW.

In Table 14, a summary of the number of cache misses for the aforementioned parallel merge algorithms is given (assuming 3-way associativity as discussed in Proposition 15). Note that the Segmented Merge Path algorithm has a different asymptotic boundary than the remaining algorithms. This is attributed to the possible sharing of cache lines by different cores that the segmented algorithm does not have, thus Segmented Merge Path has the lowest bound on the number of cache misses in the merging stage. Also note that for Segmented Merge Path there is an overlap of cache misses between the partitioning stage and the merge stage: elements fetched in the partitioning stage will not be fetched again in the merging stage. We remind the reader that C denotes the cache size and the assumption that $p << C << N$.

Merging and sorting using GPUs is a topic of great interest as well, and raises additional challenges that need to be addressed. NVIDIA's CUDA (Compute Unified Device Architecture) [113] has made the GPU a popular platform for parallel application development.

In Green et al. [73], the first GPU parallel merge algorithm is presented based on the Merge Path properties. The GPU algorithm uses the Merge Path properties at two different level of granularity. The reader is referred to [112] for an extended discussion on the GPU; for the purpose of this discussion we simplify the GPU's programming model to two levels, the stream multiprocessors(SM) and stream processors (SP). The SMs resemble the multiprocessors of the x86. The SPs are light-weight thread computing units grouped together with a single instruction decoder.

Green *et al.* use the cross diagonal search for partitioning the work to the GPU's multi processors similar to the process that is done by our algorithm. In the second phase, the SP's of each SM repeat the cross diagonal intersection with a smaller subset of the path using an algorithm that is similar to the cache-efficient algorithm that is presented in this paper such that in each iteration a subset of A and B are fetched by the SPs in a way that the best utilizes the GPU's memory control unit. Once

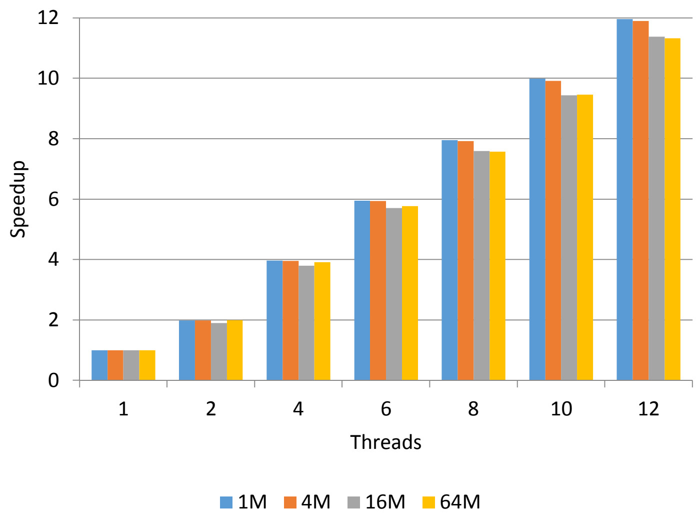


Figure 48: Merge Path speedup on a 12-core system. These results can be found in [115]. The different color bars are for different sized input arrays. 1M elements refers to 2^{20} elements. For each array size, the arrays are equisized.

the segments have been fetched into the memory the cross diagonal intersection is repeated for all the SPs on the SM. At time of publication, this is the fastest known algorithm for merging on the GPU.

In [122] a radix sort for the GPU is presented. In addition to the radix sort, the authors suggest a merge-sort algorithm for the GPU, in which the a pair-wise merge tree is required in the final stages. In [131], a hybrid sorting algorithm is presented for the GPU. Initially the data is sorted using bucket sort and this is followed by a merge sort. The bucket approach suffers from workload imbalance and requires atomic instructions (i.e., synchronization).

Another focus of sorting algorithms is finding a way to implement them in a cache oblivious [11] way. As the algorithm in this paper focused on the merging stage and not the entire sort and presented a cache aware merging algorithm, we will not elaborate on cache oblivious algorithms. The interested reader is referred to [41, 42, 68].

Table 15: Intel X86 systems used

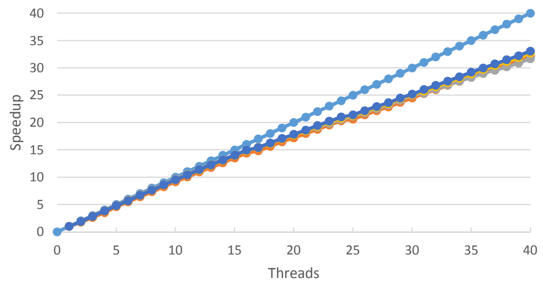
Proc.	#Proc	# Cores Per Proc.	Total Cores	L1	L2	L3	Memory
X5670 Intel	2	6	12	32KB	256KB	12MB	12GB
E7-8870 Intel	4	10	40	32KB	256KB	30MB	256GB

7.6 Results

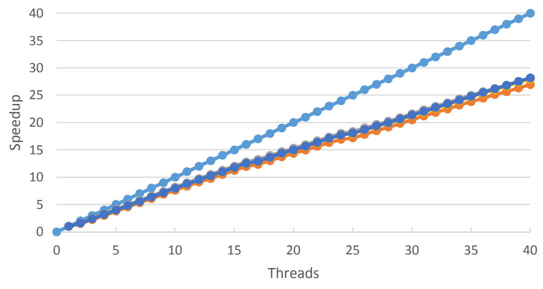
According to Amdahl's Law [14], a fraction of an algorithm that is not parallelized limits the possible speedup, It is quite evident from the previous sections that we have succeeded in truly parallelizing the entire merging and sorting process, with negligible overhead for any numbers of interest. Nonetheless, we wanted to obtain actual performance results on real systems, mostly in order to find out whether there are additional issues that limit performance. Also, so doing increases the confidence in the theoretical claims.

We implemented our basic Merge Path algorithm and the cache-efficient version. In the latter, the two arrays are segmented as described such that each segment-pair fits into a 3-way associative cache with no collisions, and the merging of one such segment pair begins only the merging of the previous pair has been completed. However, the actual fetching of array elements into the cache is done only once demanded by the processor, though any prefetch mechanisms of the system may kick in.

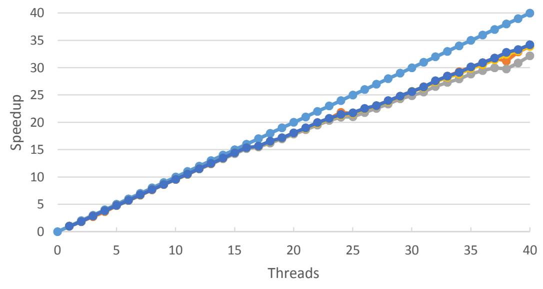
The algorithms were implemented on two very different platforms: the x86 platform and the HyperCore, a novel shared-cache many-core architecture by Plurality. We begin with a brief overview of the two systems systems, including system specifications, and then present some of the practical challenges of implementing the algorithms on each of the platforms. Following this, we present the speedup of both the new algorithms, regular Merge Path and the cache efficient version, on each of the systems. The runtime of Merge-Path with a single thread is used as the baseline.



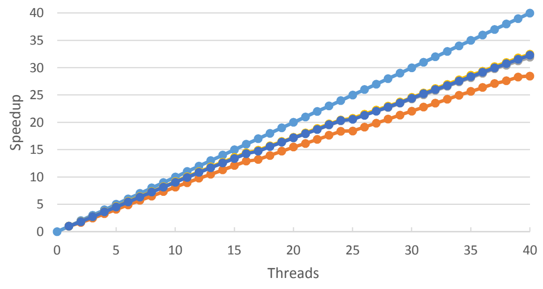
(a) $|A| = |B| = 10M$ with full writes to the output array using NUMA Control.



(b) $|A| = |B| = 50M$ with full writes to the output array using NUMA Control.



(c) $|A| = |B| = 10M$ with writes to local register.



(d) $|A| = |B| = 50M$ with writes to local register.

Figure 49: Merge Path and Segmented Merge Path speedup on a 40-core system. 1M elements refers to 2^{20} elements. For each array size, the arrays are equisized. For the segmented algorithm, the Merge Matrix is divided into 10, 5, and 2 segments.

7.6.1 x86 System

We used two different Intel X86 with Hyperthreading support: a 12 core system and a 40 core system. For both systems, each core has a L1 and L2 private cache. The cores on each processor share a L3 cache. The specifics of these systems can be found in Table 15. Because the cores have private caches, a cache coherency mechanism is required to ensure correctness. Furthermore, as we had multiple processors, each with its own L3 cache, the cache coherence mechanism had to communicate across processors; this is even more expensive from a latency point of view. We do not use the Hyperthreading. For each array size and each thread count, multiple executions were completed and the average was taken.

Our implementation of Merge Path is OpenMP [46] based. On the 12-core system we tested the regular algorithm. On the 40-core system we tested both algorithms (regular and segmented). For both systems we use multiple sizes of arrays and thread count.

Fig. 48 depicts the speedup of the algorithm as a function of threads for the 12-core system. Each of the different colored bars is for a different input size. This figure can also be found in our previous paper, [115]. In all cases $|A| = |B|$. The output array S is twice this size, meaning that the total memory required for the 3 arrays is $4 \cdot |A| \cdot |type|$, where $|type|$ denotes the number of bytes need to stored the data type (for 32 bit integers this will be 4). One mega element refers to 2^{20} elements. As can be seen, the speedups are near linear, with a slight reduction in performance for the bigger input arrays: approximately $11.7X$ for 12 threads.

Fig. 49 depicts the speedup on a larger 40-core system for both the regular and segmented algorithm. For the segmented algorithm we used multiple segment sizes: two, five, and ten segments. As such each segment size is $|S|/\#segments$. Effective parallelization on the 40 core system is more challenging than it is for the 12 core system as the 40 core system consists of 4 processors (each with 10 cores). The 4 processor design can potentially add overhead related to synchronization and cache coherency. Additional considerations include memory bandwidth saturation due to the algorithm being communication bound. We discuss these.

For each input size tested, we present two sets of results - execution time with the write backs and execution time without write backs. Both use the NUMA Contral package [92]. The speedups for the writeback of the results to the main memory are depicted in Fig. 49(a) and Fig. 49(b), for array sizes 10M and 50M respectively. The speedups for the algorithms when the results are written to a register are depicted in Fig. 49(c) and Fig. 49(d), for array sizes 10M and 50M respectively.

Note that for both array sizes, the speedup attained at 10 threads is not doubled at 20 threads and the speedup at 20 threads is not doubled for 40 threads. There are two causes for this: writing back of the elements and synchronization.

As expected, writing the results back to the main memory adds latency and thus reduces the speedup from about 32X to 28X for the 40-thread execution for the $50M$ array size. This side effect is not felt as much for the smaller array sizes, $10M$, a significant part of the array fits in the caches. Note that each processor has a $30MB$ L3 cache, with a total of $120MB$ for all 4 processors. The amount of memory required by the A, B, and S is $160MB$. As such, at the end of the merge, part of the array is still in the cache and is not written back to the main memory as occurs for the larger arrays. This behavior is not a side affect of our algorithm, rather it is a side affect of the architecture.

To verify the impact of the synchronization, we timed only the path intersection and its immediate synchronization (meaning that merging process itself was not timed). As the path intersection takes at most $O(log_2(min(|A|,$
$|B|)))$ steps, if the main cross diagonal is used for intersection in the case of equisized inputs, this intersection should take the longest amount of time. For example, when 2 threads are used, the 2^{nd} thread uses the main cross diagonal. Meaning it should be the bottleneck in the computation. As we increased the number of threads, the amount of time to find the intersections grew - regardless of the amount of computation needed. For the segmented algorithm the synchronization time also grew and became more substantial. Once again, this is not a side affect of our algorithm.

We now compare the regular algorithm with the segmented algorithm. For the smaller array, the segmented algorithm is slightly outperformed by the regular algorithm. This is due to the synchronization overhead where the synchronization costs plays a more substantial part of the total execution time. Also, likelihood of cache contention is smaller for these arrays. As such the regular algorithm outperforms

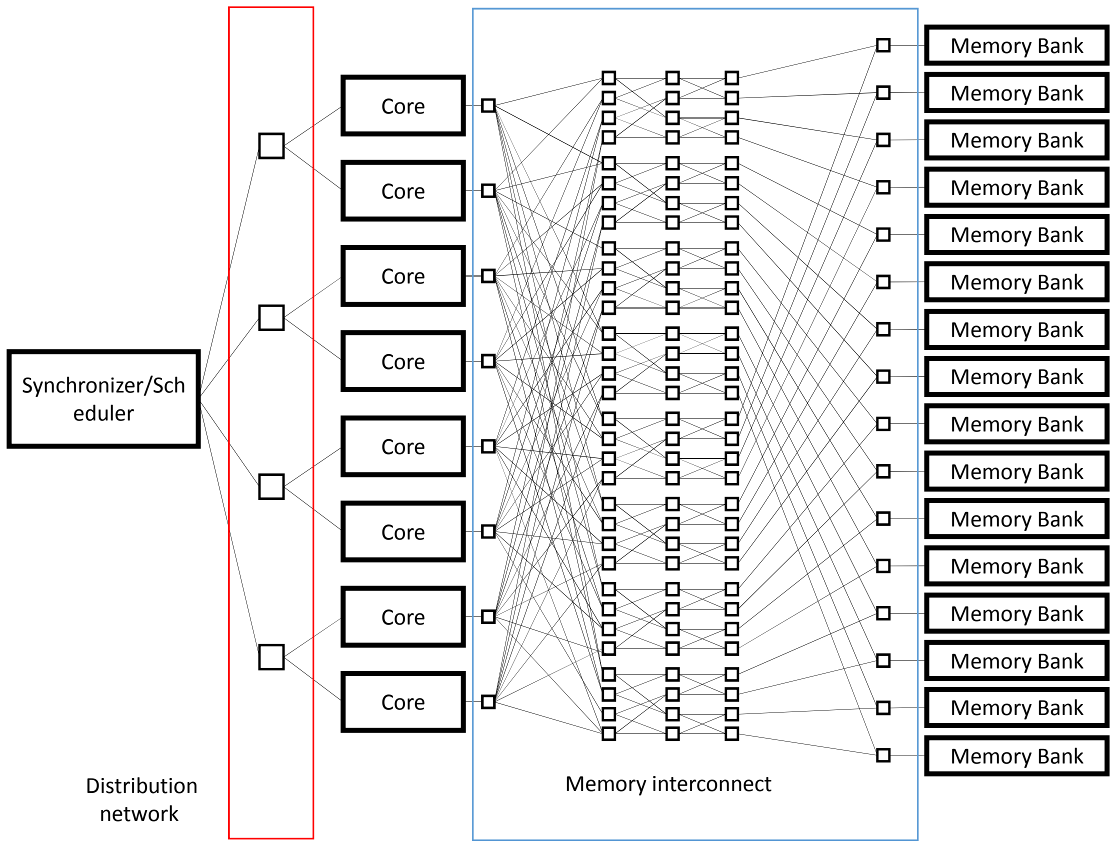


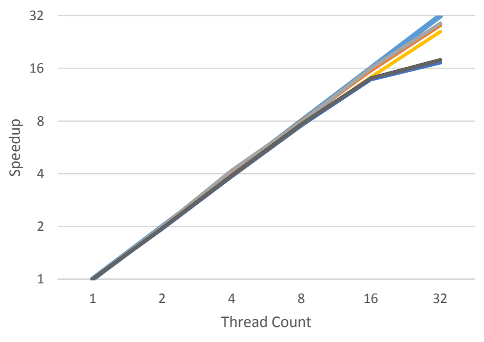
Figure 50: Schematic view of the Hypercore for an 8-core system. The memory banks refer to the shared memory. The Hypercore also has DRAM memory which is not shown in this schematic.

the segment algorithm. As the sizes of the array increases so does the likelihood of contention (for both the partitioning and merging stages). It is not surprising that the new cache efficient algorithm outperforms the regular algorithm.

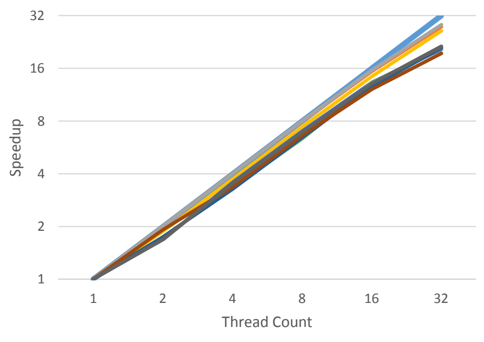
In summary, our x86 OpenMP implementations perform rather well and achieves 75% – 90% of the full system utilization (problem size dependent).

7.6.2 HyperCore Architecture

Plurality's HyperCore architecture [5, 3] features tens to hundreds of compute cores, interconnected to an even larger number of memory banks that jointly comprise the shared cache. The connection is via a high speed, low latency combinational interconnect. As there are no private caches for the cores, memory coherence is not an issue



(a) Regular Algorithm



(b) Segmented Algorithm

Figure 51: Speedup of our algorithm on Plurality's Hypercore [3] system.

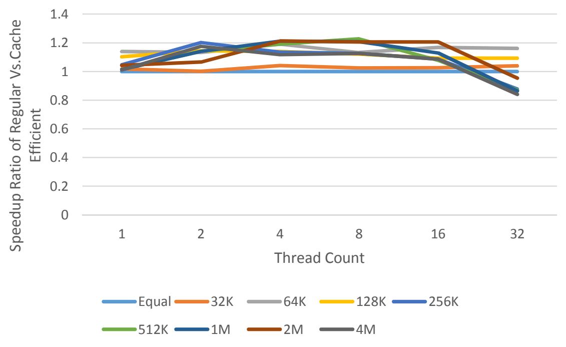


Figure 52: Speedup comparison of the regular and segmented algorithms on the Hypercore. A blue curve, *Equal*, has been added to this chart to denote when the speedups of the algorithm are equal. All curves above *Equal* mean that the regular algorithm outperforms that segmented algorithm. All curves below *Equal* mean that the segmented algorithm is more efficient.

for CREW like algorithms. Same-address writes are serialized by the communication interconnect; however, for our algorithm this was not needed. The memory banks are equidistant from all the cores, so this is a UMA system. The shared cache has a number of memory banks that is larger than the number of cores in the system, reducing the number of conflicts on a single bank. Moreover, addresses are interleaved, so there are no persistent hot spots due to regular access patterns. Fig. 50 depicts an 8-core Hypercore system.

The benefit of such an architecture is that there is no processor-cache communication bottleneck. Finally, the absence of private caches (and a large amount of state in them) and the UMA architecture permit any core to execute any compute task with equal efficiency. The memory hierarchy also includes off-chip (shared) memory. Finally, the programming model is a set of sequential "tasks" along with a set of precedence relations among them, and these are enforced by a very high throughput, low latency synchronizer/scheduler that dispatches work to the cores.

At the time of submission, Plurality has not manufactured the actual chip. We had access to an advanced experimental version of the HyperCore on an FPGA card. The FPGA version we used has a 1MB direct mapped cache and 32 cores. Furthermore, the FPGA has a latency issue on memory write back. Therefore, results are shown for an algorithm that does not write to memory. Instead, we saved the value in a private register.

We ran both the non-segmented and segmented versions of Parallel Merge Path with varying numbers of threads (cores). The input arrays (of type integer) tested on Plurality are substantially smaller than those that we tested on the x86-system due to the FPGA limitations. One might expect that merging smaller arrays would not offer significant speedups due to the overhead required in dispatching threads and to the fact that the search for partition points (binary search on a cross diagonal) become a more significant part of the computation. However, due to HyperCore's ability to dispatch a thread within a handful of cycles, the overhead is not a problem and makes the HyperCore an idle target platform. The sizes of the input arrays are denoted by the number of elements in each of the arrays A and B. Again, the arrays A and B are equisized.

Fig. 51(a) presents the speedup of our basic Parallel Merge algorithm as a function of the number of cores. Multiple input sizes were tested. It is evident that they speedup is quite close to linear up to 16 cores, regardless of the array sizes. For the larger input arrays, the speedup does decrease for the 32 core count. This is most likely due to shared-memory contention and does not occur for the segmented algorithm.

Fig. 51(b) similarly depicts the speedup for the segmented algorithm. Note, however that the cache is direct mapped, so collision freedom cannot be guaranteed. Nonetheless, the partition into sequential iteration, each carrying out parallel merge on a segment, does improve performance. The percentage speedup of the segmented version relative to same-parameter execution without segmentation is depicted in Fig. 52.

7.7 Conclusions

In this chapter, we explored the issue of parallel sorting through the cornerstone of many sorting algorithms – the merging of two sorted arrays.

One important contribution of this chapter is a very intuitive, simple and efficient approach to correctly partitioning each of two input sorted arrays into segments that, once pairs of segments, one from each, are merged, the concatenation of the merged pairs yields a single sorted array. This partitioning is also done in parallel.

Another important contribution is an insightful consideration of cache related issues. These are extremely important because, especially when parallelized, sorting and merging are carried out at a speed that is very often determined by the memory subsystem rather than by the compute power. This culminated in a cache-efficient parallel merge algorithm. To this end, the efficient segmented version of our algorithm is very promising, as it can operate efficiently with simple caches.

CHAPTER VIII

GPU MERGE PATH

This chapter is an extension of the paper: O. Green, R. McColl, D. Bader, "GPU Merge Path - A GPU Merging Algorithm", ACM 26th International Conference on Supercomputing, Venice, Italy, 2012.

Graphics Processing Units (GPUs) have become ideal candidates for the development of fine-grain parallel algorithms as the number of processing elements per GPU increases. In addition to the increase in cores per system, new memory hierarchies and increased bandwidth have been developed that allow for significant performance improvement when computation is performed using certain types of memory access patterns.

Merging two sorted arrays is a useful primitive and is a basic building block for numerous applications such as joining database queries, merging adjacency lists in graphs, and set intersection. An efficient parallel merging algorithm partitions the sorted input arrays into sets of non-overlapping sub-arrays that can be independently merged on multiple cores. For optimal performance, the partitioning should be done in parallel and should divide the input arrays such that each core receives an equal size of data to merge.

In this chapter, we present an algorithm that partitions the workload equally amongst the GPU Streaming Multi-processors (SM). Following this, we show how each SM performs a parallel merge and how to divide the work so that all the GPU's Streaming Processors (SP) are utilized. All stages in this algorithm are parallel. The new algorithm demonstrates good utilization of the GPU memory hierarchy. This approach demonstrates an average of 20X and 50X speedup over a sequential merge on the x86 platform for integer and floating point, respectively. Our implementation is 10X faster than the fast parallel merge supplied in the CUDA Thrust library.

8.1 Introduction

The merging of two sorted arrays into a single sorted array is straightforward in sequential computing, but presents challenges when performed in parallel. Since merging is a common primitive in larger applications, improving performance through parallel computing approaches can provide benefit to existing codes used in a variety of disciplines. Given two sorted arrays A,B of length $|A|$,$|B|$ respectively, the output of the merge is a third array C such that C contains the union of elements of A and B, is sorted, and $|C| = |A| + |B|$. The computational time of this algorithm on a single core is $O(|C|)$ [45]. As of now, it will be assume that $|C| = n$.

The increase in the number of cores in modern computing systems presents an opportunity to improve performance through clever parallel algorithm design; however, there are numerous challenges that need to be addressed for a parallel algorithm to achieve optimal performance. These challenges include evenly partitioning the workload for effective load balancing, reducing the need for synchronization mechanisms, and minimizing the number of redundant operations caused by the parallelization. We present an algorithm that meets all these challenges and more for GPU systems.

The remainder of the chapter is organized as follows: In this section we present a brief introduction to GPU systems, merging, and sorting. In particular, we present Merge Path [115, 114]. Section 2 introduces our new GPU merging algorithm, GPU Merge Path, and explains the different granularities of parallelism present in the algorithm. In section 3, we show empirical results of the new algorithm on two different GPU architectures and improved performance over existing algorithms on GPU and x86. Section 4 offers concluding remarks.

8.1.1 Introduction on GPU

Graphics Processing Units (GPUs) have become a popular platform for parallel computation in recent years following the introduction of programmable graphics architectures like NVIDIA's Compute Unified Device Architecture (CUDA) [113] that allow for easy utilization of the cards for purposes other than graphics rendering. For the sake brevity, we present only a short introduction to the CUDA architecture.

To the authors' knowledge, the parallel merge in the Thrust library [83] is the only other parallel merge algorithm implemented on a CUDA device.

The original purpose of the GPU is to accelerate graphics applications which are highly parallel and computationally intensive. Thus, having a large number of simple cores can allow the GPU to achieve high throughput. These simple cores are also known as stream processors (SP), and they are arranged into groups of 8/16/32 (depending on the CUDA compute capability) cores known as stream multiprocessors (SM). The exact number of SMs on the card is dependent on the particular model. Each SM has a single control unit responsible for fetching and decoding instructions. All the SPs for a single SM execute the same instruction at a given time but on different data or perform no operation in that cycle. Thus, the true concurrency is limited by the number of physical SPs. The SMs are responsible for scheduling the threads to the SPs. The threads are executed in groups called warps. The current size of a warp is 32 threads. Each SM has a local shared memory / private cache. Older generation GPU systems have 8KB local shared memory, whereas the new generation has 64KB of local shared memory which can be used in two separate modes.

In CUDA, users must group threads into blocks and construct a grid of some number of thread blocks. The user specifies the number of threads in a block and the number of blocks in a grid that will all run the same kernel code. Kernels are functions defined by the user to be run on the device. These kernels may refer to a thread's index within a block and the current block's index within the grid. A block will be executed on a single SM. For full utilization of the SM it is good practice to set the block size to be a multiple of the warp. As each thread block is executed by a single SM, the threads in a block can share data using the local shared memory of the SM. A thread block is considered complete when the execution of all threads in that specific block have completed. Only when all the threads blocks have completed execution is the kernel considered complete.

8.1.2 Parallel Sorting

The focus of this chapter is on parallel merging; however, there has not been significant study solely on parallel merging on the GPU. Therefore we give a brief description of prior work in the area of sorting: sequential, multicore parallel sorting, and GPU parallel sorting [122, 131]. In further sections there is a more thorough background on parallel merging algorithms.

Sorting is a key building block of many algorithms. It has received a large amount of attention in both sequential algorithms (bubble, quick, merge, radix) [45] and their respective parallel versions. Prior to GPU algorithms, several merging and sorting algorithms for PRAM were presented in [49, 115, 126] . Following the GPGPU trend, several algorithms have been suggested that implement sorting using a GPU for increased performance. For additional reading on parallel sorting algorithms and intricacies of the GPU architecture (specifcally NVIDIA's CUDA), we refer the reader to [122, 72, 40] on GPU sorting.

Some of the new algorithms are based on a single sorting method such as the radix sort in [122]. In [122], Satish et al. suggest using a parallel merge sort algorithm that is based on a division of the input array into sub arrays of equal size followed by a sequential merge sort of each sub array. Finally, there is a merge stage where all the arrays are merged together in a pair-wise merge tree of depth $log(p)$. A good parallelization of the merge stage is crucial for good performance. Other algorithms use hybrid approaches such as the one presented by Sintorn and Assarsson [131], which uses bucket sort followed by a merge sort. One consideration for this is that each of the sorts for a particular stage in the algorithm is highly parallel, which allows for high system utilization. Additionally, using the bucket sort allows for good load balancing in the merge sort stage; however, the bucket sort does not guarantee an equal work load for each of the available processors and – in their specific implementation – requires atomic instructions.

8.1.3 GPU Challanges

To achieve maximal speedup on the GPU platform, it is necessary to implement a platform-specific (and in some cases, card-specific) algorithm. These implementations are architecture-dependent and in many cases require a deep understanding of the memory system and the execution system. Ignoring the architecture limits the achievable performance. For example, a well known performance hinderer is bad memory access (read/write) patterns to the global memory. Further, GPU-based applications greatly benefit from implementation of algorithms that are cache aware.

For good performance, all the SPs on a single SM should read/write sequentially. If the data is not sequential (meaning that it strides across memory lines in the global DRAM), this could lead to multiple global memory requests which cause all SPs to wait for all memory requests to complete. One way to achieve efficient global memory use when non-sequential access is required is to do a sequential data read into the local shared memory incurring one memory request to the global memory and followed by 'random' (non-sequential) memory accesses to the local shared memory.

An additional challenge is to find a way to divide the workload evenly among the SMs and further partition the workload evenly among the SPs. Improper load-balancing can result in only one of the SPs out of the eight or more doing useful work while others are idle due to bad global memory access patterns or divergent

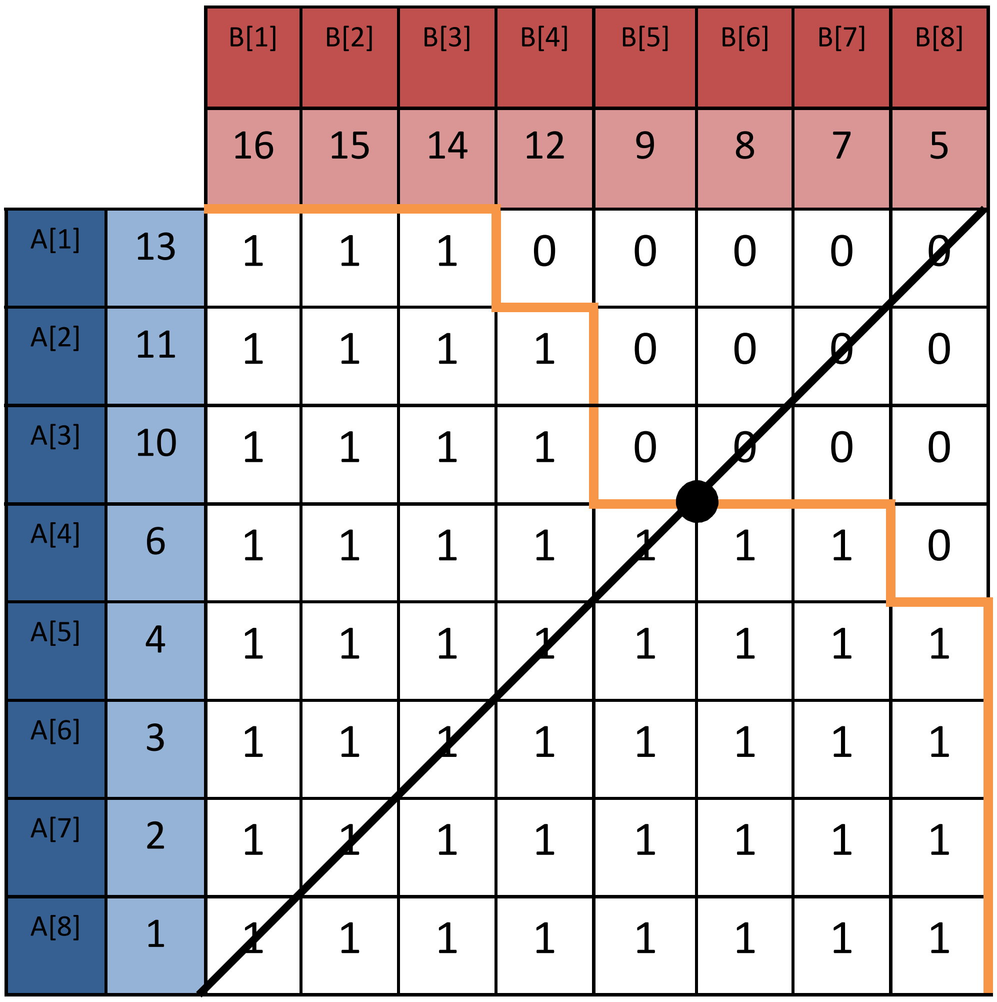


Figure 53: Illustration of the MergePath concept for a non-increasing merge. The first column (in blue) is the sorted array A and the first row (in red) is the sorted array B. The orange path (a.k.a. Merge Path) represents comparison decisions that are made to form the merged output array. The black cross diagonal intersects with the path at the midpoint of the path which corresponds to the median of the output array.

execution paths (if-statements) that are partially taken by the different SPs. For the cases mentioned, where the code is parallel, the actual execution is sequential.

It is very difficult to find a merging algorithm that can achieve a high level of parallelism and maximize utilization on the GPU due to the multi-level parallelism requirements of the architecture. In a sense, parallelizing merging algorithms is even

Algorithm 15: Pseudo code for parallel Merge Path algorithm with an emphasis on the partitioning stage.

```
$A_{diag}[threads] \Leftarrow A_{length}$
$B_{diag}[threads] \Leftarrow B_{length}$
for each $i$ in $threads$ in parallel do
    $index \Leftarrow i * (A_{length} + B_{length})\ /\ threads$
    $a_{top} \Leftarrow index > A_{length}\ ?\ A_{length} : index$
    $b_{top} \Leftarrow index > A_{length}\ ?\ index - A_{length} : 0$
    $a_{bottom} \Leftarrow b_{top}$
    // binary search for diagonal intersections
    while $true$ do
        $offset \Leftarrow (a_{top} - a_{bottom})/2$
        $a_i \Leftarrow a_{top} - offset$
        $b_i \Leftarrow b_{top} + offset$
        if $A[a_i] > B[b_i - 1]$ then
            if $A[a_i - 1] \leq B[b_i]$ then
                $A_{diag}[i] \Leftarrow a_i$
                $B_{diag}[i] \Leftarrow b_i$
                break
            else
                $a_{top} \Leftarrow a_i - 1$
                $b_{top} \Leftarrow b_i + 1$
        else
            $a_{bottom} \Leftarrow a_i + 1$
for each $i$ in $threads$ in parallel do
    merge($A, A_{diag}[i], B, B_{diag}[i], C, i * length/threads$)
```

more difficult due to the small amount of work done per each element in the input and output. The algorithm that is presented in this chapter uses the many cores of the GPU while reducing the number of requests to the global memory by using the local shared memory in an efficient manner. We further show that the algorithm is portable for different CUDA compute capabilities by showing the results on both TESLA and FERMI architectures. These results are compared with the parallel merge from the Thrust library on the same architectures and an OpenMP (OMP) implementation on an x86 system.

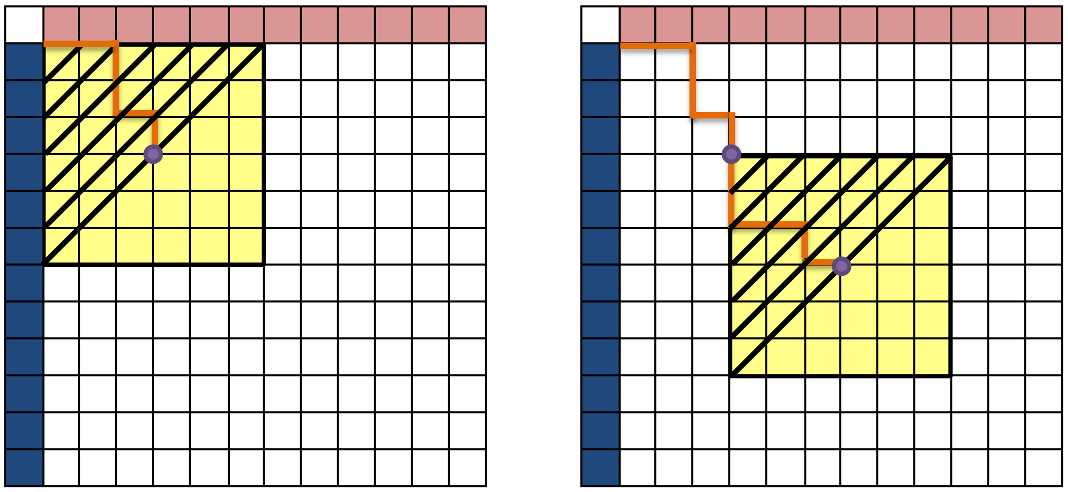

(a) Initial position of the window which. (b) New position of window(after completion of previous block).

Figure 54: Diagonal searches for a single window of one SM. (a) The window is in its initial position. Each SP does a search for the path. (b) The window moves to the farthest position of the path and the new diagonals are searched.

8.2 GPU MergePath

In this section we present our new algorithm for the merging of two sorted arrays into a single sorted array on a GPU. In the previous section we explained Merge Path [115] and its key properties. In this section, we give an introduction to parallel Merge Path for parallel systems. We also explain why the original Merge Path algorithm cannot be directly implemented on a GPU and our contribution development of the GPU Merge Path algorithm.

8.2.1 Parallel Merge

In [115] the authors suggest a new approach for the parallel merging of two sorted arrays on parallel shared-memory systems. Assuming that the system has p cores, each core is responsible for merging an equal n/p part of the final output array. As each core receives an equal amount of work, this ensures that all the p cores finish at the same time. Creating the balanced workload is one of the aspects that makes Merge Path a suitable candidate for the GPU. While the results in [115] intersect with those in [49], the approach that is presented is different and easier to understand. We use this approach to develop a Merge Path algorithm for GPU. Merge Path, like other parallel merging algorithms, is divided into 2 stages:

1. Partitioning stage - Each core is responsible for dividing the input arrays into partitions. Each core finds a single non-overlapping partition in each of the input arrays. While the sub-arrays of each partition are not equal length, the sum of the lengths of the two sub-arrays of a specific partition is equal (up to a constant of 1) among all the cores. In Algorithm 15 and in the next section, we present the pseudo code for the partitioning and give a brief explanation. For more information we suggest reading [49, 114, 115].

2. Merging stage - Each core merges the two sub arrays that it has been given using the same algorithm as a simple sequential merge. The cores operate on non-overlapping segments of the output array, thus the merging can be done concurrently and lock-free. Using the simple sequential mere algorithm for this stage is not well-suited to the GPU.

Both of these stages are parallel.

As previously stated, Merge Path suggests treating the sequential merge as if it was a path that moves from the top-left corner of a rectangle to the bottom-right corner of the rectangle $(|B|, |A|)$. The path can move only to the right and down. We denote $n = |A + |B|$. When the entire path is known, so is the order in which the merge takes place. Thus, when an entire path is given, it is possible to complete the merge concurrently; however, at the beginning of the merge the path is unknown and computation of the entire path through a diagonal binary search for all the cross diagonals is considerably expensive $O(n \cdot log(n))$ compared with the complexity of sequential merge which is $O(n)$. Therefore, we compute only p points on this path. These points are the partitioning points.

To find the partitioning points, use cross diagonals that start at the top and right borders of the output matrix. The cross diagonals are bound to meet the path at some point as the path moves from the top-left to the bottom-right and the cross diagonals move from the top-right to the bottom-left. It is possible to find the points of intersection using binary searches on the cross diagonals by comparing elements from A and B (Observation 1), making the complexity of finding the intersection $O(log(n))$.

By finding exactly p points on the path such that these points are equally distanced, we ensure that the merge stage is perfectly load balanced. By using equidistant cross diagonals, the work load is divided equally among the cores, see [115]. The merging of the sub-arrays in each partition is the same as the standard sequential merge that was discussed earlier in this chapter. The time complexity of this algorithm is $O(n/p + log(n))$. This is also the work load of each of the processors in the system. The work complexity of this algorithm is $O(n + p \cdot log(n))$.

8.2.2 GPU Partitioning

The above parallel selection algorithm can be easily implemented on a parallel machine as each core is responsible for a single diagonal. This approach can be implemented on the GPU; however, it does not fully utilize the GPU as SPs on each SM will frequently be idle. We present 3 approaches to implementing the cross diagonal binary search followed by a detailed description. We denote w as the size of the warp. The 3 approaches are:

1. w-wide binary search.

2. Regular binary search.

3. w-partition search.

Detailed description of the approaches is as follows:

1. w-wide binary search - In this approach we fetch w consecutive elements from each of the arrays A and B. By using CUDA block of size w, each SP / thread is responsible for fetching a single element from each of the global arrays, which are in the global memory, into the local shared memory. This efficiently uses the memory system on the GPU as the addresses are consecutive, thus incurring a minimal number of global memory requests. As the intersection is a single point, only one SP finds the intersection and stores the point of intersection in global memory, which removes the need for synchronization. It is rather obvious that the work complexity of this search is greater than the one presented in Merge Path[115] which does a regular sequential search for each of the diagonals; however, doing w searches or doing 1 search takes the same amount of time in practice as the additional execution units would otherwise be idle if we were executing only a single search. In addition to this, the GPU architecture has a wide memory bus that can bring more than a single data element per cycle making it cost-effective to use the fetched data. In essence for each of the stages in the binary search, a total of w operations are completed. This approach reduces the number of searches required by a factor of w. The complexity of this approach for each diagonal is: $Time = O(log(n) - log(w))$ for each core and a total of $Work = O(w \cdot log(n) - log(w))$.

2. Regular binary search - This is simply a single-threaded binary search on each diagonal. The complexity of this: $Time = O(log(n)))$ and $Work = O(log(n))$. Note that the SM utilization is low for this case, meaning that all cores but one will idle and the potential extra computing power is wasted.

3. w-partition search - In this approach, the cross diagonal is divided into 32 equal-size and independent partitions. Each thread in the warp is responsible for a

single comparison. Each thread checks to see if the point of intersection is in its partition. Similar to w-wide binary search, there can only be one partition that the intersection point goes through. Therefore, no synchronization is needed. An additional advantage of this approach is that in each iteration the search space is reduced by a factor of w rather than 2 as in binary search. This reduces the $O(log(n))$ comparisons needed to $O(log_w(n))$ comparisons. The complexity of this approach for each diagonal is: $Time = O(log_w(n) - log(w))$ and $Work = O(w \cdot log_w(n) - log(w))$. The biggest shortcoming of this approach is that for each iteration of that partition-search, a total of w global memory requests are needed(one for each of the partitions limits). As a result, the implementation suffers a performance penalty from waiting on global memory requests to complete.

In conclusion, we tested all of these approaches, and the first two approaches offer a significant performance improvement over the last due to a reduced number of requests to the global memory for each iteration of the search. This is especially important as the computation time for each iteration of the searches is considerably smaller than the global memory latency. The time difference between the first two approaches is negligible with a slight advantage to one or the other depending on the input data. For the results that are presented in this chapter we use the w-wide binary search.

8.2.3 GPU Merge

The merge phase of the original Merge Path algorithm is not well-suited for the GPU as the merging stage is purely sequential for each core. Therefore, it is necessary to extend the algorithm to parallelize the merge stage in a way that still uses all the SPs on each SM once the partitioning stage is completed.

For full utilization of the SMs in the system, the merge must be broken up into finer granularity to enable additional parallelism while still avoiding synchronization when possible. We present our approach on dividing the work among the multiple SPs for a specific workload.

For the sake of simplicity, assume that the sizes of the partitions that are created by the partitioning stage are significantly greater than the warp size, w. Also we denote the CUDA thread block size using the letter Z and assume that $Z \geq w$. For practical purposes $Z = 32$ or 64; however, anything that is presented in this subsection can also be used with larger Z. Take a window consisting of the Z largest elements of each of the partitions and place them in local shared memory (in a sequential manner for performance benefit). Z is smaller than the local shared memory, and therefore the data will fit. Using a Z thread block, it is possible to find the exact Merge Path of the Z elements using the cross diagonal binary search.

Given the full path for the Z elements it is possible to know how to merge all the Z elements concurrently as each of the elements are written to a specific index. The complexity for finding the entire Z-length path requires $O(log(Z))$ time in general iterations. This is followed by placing the elements in their respective place in the output array. Upon completion of the Z-element merge, it is possible to move on to unmerged elements by starting a new merge window whose top-left corner starts at the bottom-right-most position of the merge path in the previous window. This can be seen in Fig. 54 where the window starts off at the initial point of merge for a given SM. All the threads do a diagonal search looking for the path. Moving the window is a simple operation as it requires only moving the pointers of the sub-arrays according to the (x, y) lengths of the path. This operation is repeated until the SM finishes merging the two sub-arrays that it was given. The only performance requirement of the algorithm is that the sub-arrays fit into the local shared memory of the SM. If the sub-arrays fit in the local shared memory, the SPs can perform random memory access without incurring significant performance penalty. To further offset the overhead of the path searching, we let each of the SPs merge several elements.

8.2.4 Complexity analysis of the GPU merge

Given p blocks of Z threads and n elements to merge where n is the total size of the output array, the size of the partition that each of the blocks of threads receives is n/p. Following the explanation in the previous sub-section on the movement of the sliding window, the window moves a total of $(n/p)/Z$ times for that partition. For each window, each thread in the block performs a binary diagonal search that is dependent on the block size Z. When the search is complete, the threads copy their resulting elements into independent locations in the output array directly. Thus, the time complexity of merging a single window is $O(log(Z))$. The total amount of work that is completed for a single block is $O(Z \cdot log(Z))$. The total time complexity for the merging done by a single thread block is $O(n/(p \cdot Z) \cdot log(Z))$ and the work complexity is $O(n/p \cdot log(Z))$. For the entire merge the time complexity stays the same, $O(n/(p \cdot Z) \cdot log(Z))$, as all the cores are expected to complete at the same time. The work complexity of the entire merge is $O(n \cdot log(Z))$.

The complexity bound given for the GPU algorithm is different than the one given in [115, 114] for the cache efficient Merge Path. The time complexity given by Odeh et el. is $O(n/p + n/\hat{Z} \cdot \hat{Z})$, where $\hat{Z}$ refers to the size of the shared memory and not the block size. It is worth noting that the GPU algorithm is also limited by the size of the shared memory that each of the SMs has, meaning that Z is bounded by the size of the shared memory.

While the GPU algorithm has a higher complexity bound, we will show in the results section that the GPU algorithm offers significant speedups over the parallel multicore algorithm.

Table 16: GPU test-bench.

Card Type	CUDA HW	CUDA Cores	Frequency	Shared Memory	Global Memory	Memory Bandwidth
Tesla C1060	1.3	240 cores / 30 SMs	1.3 GHz	16KB	2GB	102 GB/s
Fermi M2070	2.0	448 cores / 14 SMs	1.15 GHz	64KB	6GB	150.3 GB/s

Table 17: CPU test-bench.

Processor	Cores	Clock Frequency	L2 Cache	L3 Cache	DRAM	Memory Bandwidth
2 x Intel Xeon E5530	4	2.4GHz	4 x 256 KB	8MB	12GB	25.6 GB/s

8.2.5 GPU Optimizations and Issues

1. Memory transfer between global and local shared memory is done in sequential reads and writes. A key requirement for the older versions of CUDA, versions 1.3 and down, is that when the global memory is accessed by the SPs, the elements requested be co-located and not permuted. If the accesses are not sequential, the number of global memory requests increases and the entire warp (or partial-warp) is frozen until the completion of all the memory requests. Acknowledging this requirement and making the required changes to the algorithm has allowed for a reduction in the number of requests to the global memory. In the local shared memory it is possible to access the data in a non-sequential fashion without as significant of a performance degradation.

2. The sizes of threads and thread blocks, and thus the total number of threads, are selected to fit the hardware. This approach suggests that more diagonals are computed than the number of SMs in the system. This increases performance for both of the stages in the algorithm. This is due to the internal scheduling of the GPU which requires a large number of threads and blocks to achieve the maximal performance through latency hiding.

8.3 Empirical Results

In this section we present comparisons of the running times of the GPU Merge Path algorithm on the NVIDIA Tesla and Fermi GPU architectures with those of the Thrust parallel merge [83] on the same architectures and Merge Path on a x86 Intel Nehalem

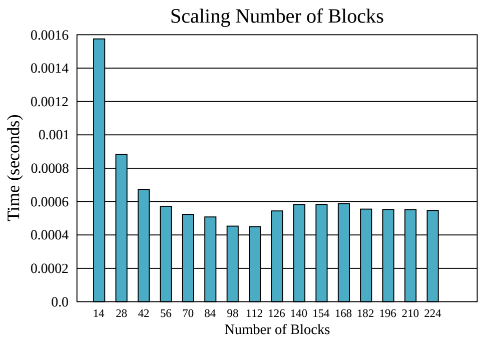


Figure 55: Merging one million single-precision floating point numbers in Merge Path on Fermi while varying the number of thread blocks used from 14 to 224 in multiples of 14.

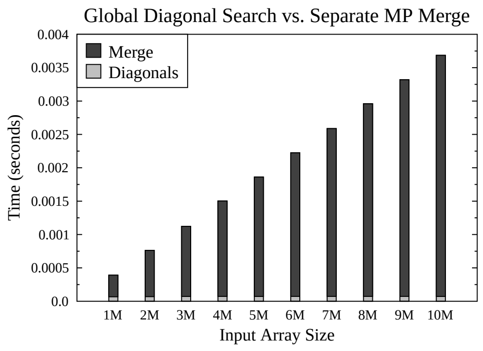
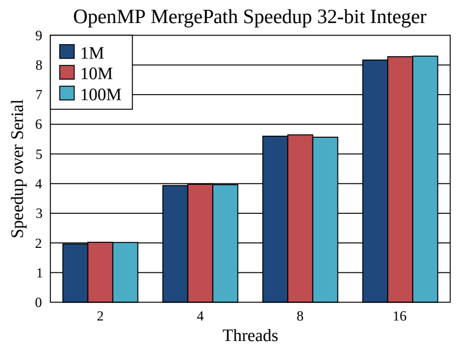


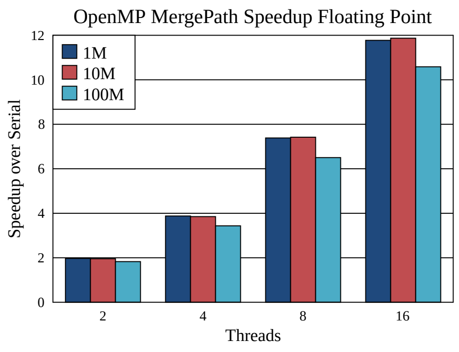
Figure 56: The timing of the global diagonal search to divide work among the SMs compared with the timing of the independent parallel merges performed by the SMs.

core system. The specifications of the GPUs used can be seen in Table 1, and the Intel multi-core configuration can be seen in Table 2. For the x86 system we show results for both a sequential merge and for OpenMP implementation of Merge Path. Other than the Thrust library implementation, the authors could not find any other parallel merge implementations for the CUDA architecture, and therefore, there are no comparisons to other CUDA algorithms to be made. Thrust is a parallel primitives library that is included in the default installation of the CUDA SDK Toolkit as of version 4.0.

The GPUs used were a Tesla C1060 supporting CUDA hardware version 1.3 and a Tesla C2070 (Fermi architecture) supporting CUDA hardware version 2.0. The

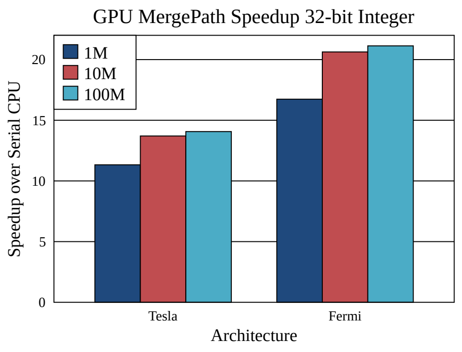


(a) 32-bit integer merging in OpenMP

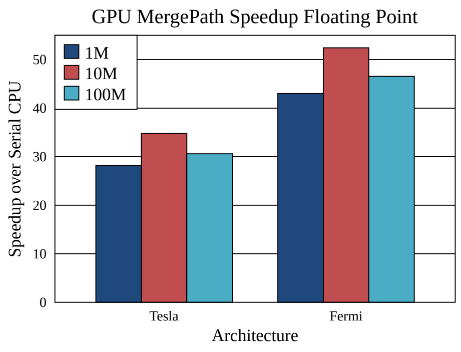


(b) Single precision floating point merging in OpenMP

Figure 57: The speedup of the OpenMP implementation of Merge Path on two hyper-threaded quad-core Intel Nehalem Xeon E5530s over a serial merge on the same system.



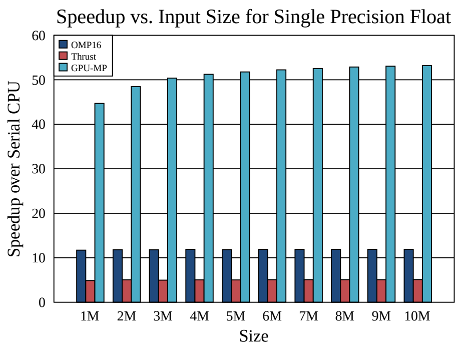
(a) 32-bit integer merging on GPU



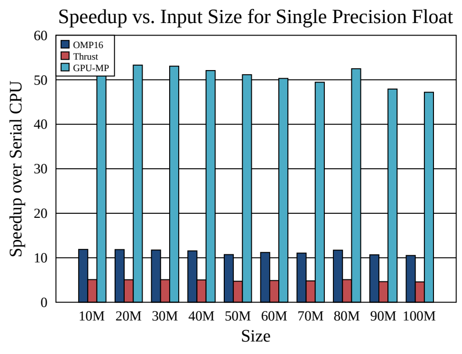
(b) Single precision floating point merging on GPU

Figure 58: The speedup of the GPU implementations of Merge Path on the NVIDIA Tesla and Fermi architectures over the x86 serial merge.

primary differences between the older Tesla architecture and the newer Fermi architecture are (1) the increased size of the local shared memory per SM from 16KB to 64KB, (2) the option of using all or some portion of the L1 local shared memory as a hardware-controlled cache, (3) increased total CUDA core count, (4) increased memory bandwidth, and (5) increased SM size from 8 cores to 32 cores. In our experiments, we use the full local shared memory configuration. Managing the workspace of a thread block in software gave better results than allowing the cache replacement

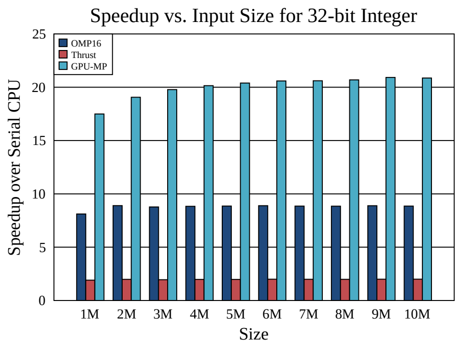


(a) 1M to 10M elements

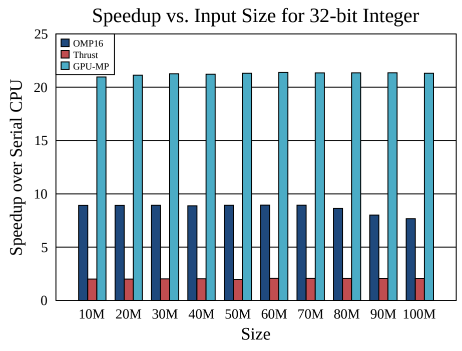


(b) 10M to 100M elements

Figure 59: Speedup comparison of merging single-precision floating point numbers using Parallel Merge Path on 16 threads, Merge Path on Fermi (112 blocks of 128 threads), and thrust::merge on Fermi. Size refers to the length of a single input array in elements.



(a) 1M to 10M elements



(b) 10M to 100M elements

Figure 60: Speedup comparison of merging 32-bit integers using Parallel Merge Path on 16 threads, GPU Merge Path on Fermi (112 blocks of 128 threads), and thrust::merge on Fermi. Size refers to the length of a single input array in elements.

policy to control which sections of each array were in the local shared memory.

An efficient sequential merge on x86 is used to provide a basis for comparison. Tests were run for input array sizes of one million, ten million, and one hundred million elements for a merged array size of two million, twenty million, and two hundred million merged elements respectively. For each size of array, merging was tested on both single-precision floating point numbers and 32-bit integers. Our results demonstrate that the GPU architecture can achieve a 2x to 5x speedup over using 16 threads on two hyper-threaded quad-core processors. This is an effect of both the greater parallelism and higher memory bandwidth of the GPU architecture.

Initially, we ran tests to obtain an optimal number of thread blocks to use on the GPU for best performance with separate tests for Tesla and Fermi. Results for this block scaling test on Fermi at one million single-precision floating point numbers using blocks of 128 threads can be seen in Fig. 55. Clearly, the figure shows the best performance is achieved at 112 blocks of 128 threads for this particular case. Similarly, 112 blocks of 128 threads also produces the best results in the case of merging one million 32-bit integer elements on Fermi. For the sake of brevity, we do not present the graph.

In Fig. 56, we show a comparison of the runtime of the two kernels: partitioning and merging. It can be seen that the partitioning stage has a negligible runtime compared with the actual parallel merge operations and increases in runtime only very slightly with increased problem size, as expected. This chart also demonstrates that our implementation scales linearly in runtime with problem size while utilizing the same number of cores.

Larger local shared memories and increased core counts allow the size of a single window on the Fermi architecture to be larger with each block of 128 threads merging 4 elements per window for a total of 512 merged-elements per window. The thread block reads 512 elements cooperatively from each array into local shared memory, performs the cross diagonal search on this smaller problem in local shared memory, then cooperatively writes back 512 elements to the global memory. In the Tesla implementation, only 32 threads per block are used to merge 4 elements per window for a total of 128 merged-elements due to local shared memory size limitations and memory latencies. Speedup for the GPU implementation on both architectures is presented in Fig. 58 for sorted arrays of 1 million,10 million, and 100 millions elements.

As the GPU implementations are benchmarked in comparison to the x86 sequential implementation and to the x86 parallel implementation, we first present these results. This is followed by the results of the GPU implementation. The timings for the sequential and OpenMP implementations are performed on a machine presented in Table 16. For the OpenMP timings we run the Merge Path algorithm using 2, 4, 8, and 16 threads on an 8-core system that supports hyper-threading (dual socket with quad core processors). Speedups are depicted in Fig. 57. As hyper-threading requires two threads per core to share execution units and additional hardware, the performance per thread is reduced versus two threads on independent cores. The hyper-threading option is used only for the 16 thread configuration. For the 4 thread configuration we use a single quad core socket, and for the 8 thread configuration we use both quad core sockets.

For each configuration, we perform three merges of two arrays of sizes 1 million, 10 million, and 100 million elements as with the GPU. Speedups are presented in Fig. 57. Within a single socket, we see a linear speedup to four cores. Going out of socket, the speedup for eight cores was 5.5X for integer merging and between 6X and 8X for floating point. We show a 12x speedup for the hyper-threading configuration for ten million floating point elements.

The main focus of our work is to introduce a new algorithm that extends the Merge Path concept to GPUs. We now present the speedup results of GPU Merge Path over the sequential merge in Fig. 58. We use the same size arrays for both the GPU and OpenMP implementations. For our results we use equal size arrays for merging; however, our method can merge arrays of different sizes.

8.3.1 Floating Point Merging

For the floating point tests, random floating point numbers are generated to fill the input arrays then sorted on the host CPU. The input arrays are then copied into the

GPU global memory. These steps are not timed as they are not relevant to the merge. The partitioning kernel is called first. When this kernel completes, a second kernel is called to perform full merges between the global diagonal intersections using parallel merge path windows per thread block on each SM. These kernels are timed.

For arrays of sizes 1 million, 10 million, 100 million we see significant speedups of 28X & 34X & 30X on the Tesla card and 43X & 53X & 46X on the Fermi card over the sequential x86 merge. This is depicted in Fig. 58. In Fig. 59, we directly compare the speedup of the fastest GPU (Fermi) implementation, the 16-thread OpenMP implementation, and the Thrust GPU merge implementation. We checked a variety of sizes. In Fig. 59 (a) there are speedups for merges of sizes 1 million elements to 10 million elements in increments of 1 millions elements. In Fig. 59 (b) there are speedups for merges of sizes 10 million elements to 100 million elements in increments of 10 millions elements. The results show that the GPU code ran nearly 5x faster than 16-threaded OpenMP and nearly 10x faster than Thrust merge for floating point operations. It is likely that the speedup of our algorithm over OpenMP is related to the difference in memory bandwidth of the two platforms.

8.3.2 Integer Merging

The integer merging speedup on the GPU is depicted in Fig. 58 for arrays of size 1 million, 10 million, 100 million. We see significant speedups of 11X & 13X & 14X on the Tesla card and 16 & 20X & 21X on the Fermi card over the sequenctial x86 merge. In Fig. 60, we directly compare the speedup over serial of the fastest GPU (Fermi) implementation, the 16-thread OpenMP implementation, and the Thrust GPU merge implementation demonstrating that the GPU code ran nearly 2.5X faster than 16-threaded OpenMP and nearly 10x faster on average than the Thrust merge. The number of blocks used for Fig. 60 is 112 blocks, similar to the number of blocks used in the floating point sub-section. We used the same sizes of sorted arrays for integer as we did for floating point.

8.4 Conclusion

We show a novel algorithm for merging sorted arrays using the GPU. The results show significant speedup for both integer and floating point elements. While the speedups are different for the two types, it is worth noting that the execution time for merging for both these types on the GPU are nearly the same. The explanation for this phenomena is that the execution time for merging integers on the CPU is 2.5X faster than the execution time for floating point on the CPU. This explains the reduction in the the speedup of integer merging on the GPU in comparison with speedup of floating point merging.

The new GPU merging algorithm is atomic-free, parallel, scalable, and can be adapted to the different compute capabilities and architectures that are provided by NVIDIA. In our benchmarking, we show that the GPU algorithm outperforms a sequential merge by a factor of 20X-50X and outperforms an OpenMP implementation of Merge Path that uses 8 hyper-threaded cores by a factor of 2.5X-5X.

This new approach uses the GPU efficiently and takes advantage of the computational power of the GPU and memory system by using the global memory, local shared-memory and the bus of the GPU effectively. This new algorithm would be beneficial for many GPU sorting algorithms including for a GPU merge sort algorithm.

CHAPTER IX

A COMPUTATIONALLY EFFICIENT ALGORITHM FOR THE 2D COVARIANCE METHOD

This chapter is an extension of the paper: O. Green, Y. Birk, "A Computationally Efficient Algorithm for the 2D Covariance Method", ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis, Denver, Colorado, 2013.

The estimated covariance matrix is a building block for many algorithms, including signal and image processing. The Covariance Method is an estimator for the covariance matrix, favored both as an estimator and in view of the convenient properties of the matrix that it produces. However, the considerable computational requirements limit its use. We present a novel computation algorithm for the covariance method, which dramatically reduces the computational complexity (both ALU operations and memory access) relative to previous algorithms. It has a small memory footprint, is highly parallelizable and requires no synchronization among compute threads. On a 40-core X86 system, we achieve 1200X speedup relative to a straightforward single-core implementation; even on a single core, 35X speedup is achieved.

9.1 Introduction

9.1.1 Background

The use of estimated covariance matrices originated in speech processing [105]. The Covariance Method is one method (estimator) for producing this matrix. Other applications requiring an estimated covariance matrix, not necessarily the one created by the Covariance Method, include nuclear magnetic resonance spectroscopy (currently better known as Magnetic Resonance Imaging, MRI) [33] and watermarking security based on a cryptanalysis [38]. The Covariance Method's main rationale is minimizing the error in the estimate of a covariance matrix of a time series. Good estimates can give insights pertaining to the data periodicity, and enable fast and accurate analysis of the input data. The Covariance Method is a biased estimator, and its output is a Hermitian, positive semi-definite matrix. Unlike other methods, it is guaranteed to be non-singular. This often causes the Covariance Method to be preferred [106, 90].

The 2D Covariance Method, to be described later, is widely used for image processing because of the 2D dimensionality. It has also been applied elsewhere: signal processing applications wherein the signal's temporal correlation is needed [48], synthetic aperture radar range-azimuth focusing [103], smoothing spatial clutter by averaging over given transposed data [64], to perform 2D spectral analysis [87], and more. In all the aforementioned cases, the generation of the estimated covariance matrix using the Covariance Method is an important computational building block.

Efficient implementation of the 1D Covariance Method is trivial due to the trivially exploitable overlap of computations that need not be repeated. The 2D Covariance Method is significantly more challenging due to the non-trivial overlap, and its straightforward implementation requires multiple dense vector-vector multiplications. We employ similar parallelization techniques to those presented in [47, 76]; there, redundant multiplications were eliminated without requiring synchronization among parallel threads, but redundant additions limited the effectiveness of the parallelization and limited a single thread execution speedup over the sequential to 2X-4X. In this work, we almost entirely eliminate the redundant additions while retaining the benefits of [47, 76], thereby achieving a dramatic performance improvement. The result is a sequential algorithm that is 30X-40X faster than the straightforward implementation (problem-size dependent); in conjunction with the parallelization of [47], we obtain an extremely efficient parallel algorithm: fine-grain parallelism, good load balancing, minimal repetition of work and no need for synchronization or communication among the parallel threads. Implementation results demonstrate these properties and a comparison of the actual load balancing and the theoretical load balancing is also discussed. The output of our algorithm is the same as that of the straightforward implementation of the Covariance Method.

Accelerators and HPC platforms such as the Cray XMT and NVIDIA's CUDA platform have been used for accelerating image processing algorithms. The Cray XMT2 was used for image segmentation based on a maxflow mincut approach on $32k \times 32k$ images [29]. CUDA is used for CT reconstruction [124]. Although we demonstrate our scheme on a multi-core X86 platform, the nature of the parallelizability (total independence) and the similarity of computation suggest that it can also be beneficial on various acceleration platforms, including SIMD-oriented ones.

The remainder of the chapter is organized as follows. In the rest of this section we introduce the Covariance Method, discuss related works, present some deficiencies of the straightforward algorithm and state the parallelization challenges. In Section 2 review the scheme of [47] which partitions the work into non-trivial units. This approach exposes unique characteristics of the Covariance Method, which we then utilize for reducing the total number of operations. Section 3 introduces our new algorithm, followed by a detailed complexity analysis. In Section 4, we compare the measured performance of the different algorithms. These results are close to the expected ones, despite the fact that the complexity analysis accounts for operations, ignoring issues such as the memory system. We then discuss the relevance to high performance computing. Finally, Section 5 offers some concluding remarks.

9.1.2 The Covariance Method

The input is an $N_r \times N_c$ matrix, A. It may represent pixel values of a 2D image, some other 2D spatial representation thereof (e.g., [48, 140]), or a block ("aperture") within

Field	Description	Storage
N_r	# rows in input matrix	1
N_c	# columns in input matrix	1
S_r	# rows in sliding window	1
S_c	# in sliding window	1
A	Input matrix	$N_r \times N_c$
$W^{k,l}$	Sliding window positioned at (k,l)	$S_r \times S_c$
$V^{k,l}$	Column stack of window $W^{k,l}$	$S_r \cdot S_c \times 1$
C	Output - estimated covariance matrix	$S_r \cdot S_c \times S_r \cdot S_c$
$C^{k,l}$	Partial sum array for $W^{k,l}$	$S_r \cdot S_c \times S_r \cdot S_c$
F^H	Conjugated transpose of F	
$Index$	Element in the output matrix and its location	

Table 18: Taxonomy of the Covariance Method

an image, with the estimated covariance matrix computed separately for each such block. Table 18 presents the taxonomy of the Covariance Method.

Computation of the estimated covariance matrix using the Covariance Method is based on the movement of a sliding window W of size $S_r \times S_c$ over the input matrix, starting from the top left corner and moving to the bottom right corner. Each of these windows can be considered as a sample from the sample space, the input matrix, where the dimensionality of the data is the size of the window. The considered window positions are all those that are fully encased within the boundaries of the input matrix. In the context of SAR imaging, this window is sometimes referred to as a sub-aperture. Fig. 61 depicts two 3×3 sliding window positions. A window in a given position is denoted $W^{k,l}$ where (k,l) is the position of the window's top left corner. There are $(N_r - S_r + 1) \times (N_c - S_c + 1)$ legal window positions. In the remainder of the chapter, we refer to the estimated covariance matrix produced by the Covariance Method as the *output matrix*, and use *window* also to refer to a window position.

For each window, $W^{k,l}$ is converted into a column stack $V^{k,l}$. A column stack of a matrix is a single column vector that is compromised the first column, followed by the second column, third column and all the way to the last column. This is followed by

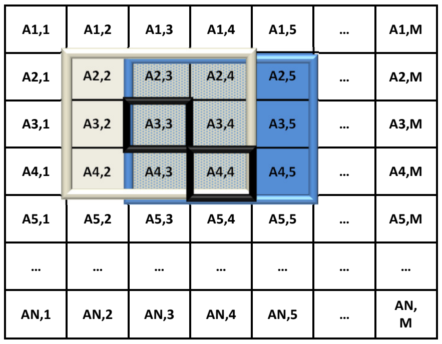

A1,1 A1,2 A1,3 A1,4 A1,5 … A1,M
A2,1 A2,2 A2,3 A2,4 A2,5 … A2,M
A3,1 A3,2 A3,3 A3,4 A3,5 … A3,M
A4,1 A4,2 A4,3 A4,4 A4,5 … A4,M
A5,1 A5,2 A5,3 A5,4 A5,5 … A5,M
… … … … … … …
AN,1 AN,2 AN,3 AN,4 AN,5 … AN, M

Figure 61: Two windows, silver (lighter) and blue (darker), and their overlapping sections. The black boxes around $A_{3,3}$ and $A_{4,4}$ refer to a product that needs to be computed for both windows.

a dense vector-vector outer multiplication of $V^{k,l}$ by its conjugate transpose $(V^{k,l})^H$. The result is a partial sum of pairwise element products, used for the construction of the output matrix, and is stored in $C^{k,l}$.

Definition 2 *The estimated covariance matrix produced by the Covariance Method is given by :*

$$C = \sum_{k=1}^{N_r-S_r} \sum_{l=1}^{N_c-S_c} C^{k,l} = \sum_{k=1}^{N_r-S_r} \sum_{l=1}^{N_c-S_c} \vec{V}^{k,l} \cdot (\vec{V}^{k,l})^H. \tag{16}$$

This expression can also be regarded as a serial and straightforward algorithm for computing C. Note that C is Hermitian, being the sum of Hermitian matrices, so only the upper triangle or the lower triangle needs to be computed. Note also that some algorithms normalize this equation by the number of windows while others do not; this is an implementation detail that does change the scheme or performance of the algorithm.

Remark. In an actual implementation, the values can be written directly to C (the output matrix) rather than being stored in temporary matrices in order to reduce the memory footprint. However, this comes at the expense of a sparse memory access pattern and may reduce the effectiveness of caching. We use temporary matrices.

9.1.3 Related Work

In [48], the covariance matrix is used as part of the Minimum Variance Method (MVM) for Synthetic Aperture Radar (SAR) image processing. Due to the high complexity of computing the estimated covariance method for MVM, DeGraaf [48] suggested an alternative and computationally cheaper estimated covariance matrix as part of the Reduced-Rank MVM (RRMVM). However, this approach reduces the effectiveness of the algorithm by introducing more noise, making the Covariance Method preferable. In [48], an image of size $1.6k \times 1.6k$ is used as a representative example. The image is divided into $10,000$ 25×25 input matrices (there is overlap between the input matrices). This requires computing 10^4 estimated covariance matrices per image or per frame in an image sequence.

Yadin *et al.* [140] use images of size $8k \times 8k$ with the MVM algorithm, computing $25 \cdot 10^4$ estimated covariance matrices (25X more matrices than were required by [48]). Both papers show that MVM gives a better output image than using FFT.

In [87] the Covariance Method is compared with the Toeplitz-Block-Toeplitz method as part of the classic Capon estimator [37] and with APES (Amplitude and Phase) spectral estimator [101, 102]. The covariance method is experimentally shown to be more computationally demanding than these methods by about an order of magnitude. In [88], the Capon and APES are extended for the creation of a new estimator which again uses the Covariance Method. Reducing the matrix computation time is important in view of the number of such computations. In fact, the amount of required computation may determine whether on-board computation (e.g., satellite) is practical, with interesting operational ramifications such as support of real-time decisions and adaptations. In [47, 76], parallelization of the Covariance Method is studied. An effective parallelization scheme is presented, and redundant multiplications are avoided. We will return to this later.

In this work, we reduce the required computations by avoiding both redundant additions and multiplication while maintaining the same effective parallelization scheme with improved load-balancing. By providing a more computationally efficient implementation of the functionally-advantageous Covariance Method, we broaden its applicability, be it in terms of image size or frame rate.

9.1.4 Deficiencies of the Straightforward Algorithm

The multiplication of $V^{k,l} \cdot (V^{k,l})^H$ in (16) is actually a multiplication of every element in each vector by every element in the other one. Therefore, when considering two sliding window positions $W^{k,l}$ and $W^{k',l'}$ such that both contain the product of some two elements x and y, both result matrices $C^{k,l}$ and $C^{k',l'}$ will contain the following products: $x \cdot \bar{x}, x \cdot \bar{y}, y \cdot \bar{x}$ and $y \cdot \bar{y}$. These products will also be added to the different partial sum matrices at different indices. Moreover, there is an overlap in the summation process as a given partial sum of products may be part of multiple indices.

In Fig. 61, for example, the product $A_{3,3} \cdot \overline{A_{4,4}}$ is required for all windows containing these two elements. Two of these windows are depicted in Fig. 61. Note the different positions of these products relative to the upper left corner of the sliding window in $C^{k,l}$ and $C^{k',l'}$. This difference determines the indices to which the product contributes. The straightforward algorithm computes each product multiple times, once per target index. Now consider the products $A_{3,3} \cdot \overline{A_{3,3}}$ and $A_{4,4} \cdot \overline{A_{4,4}}$. Each of these products can be found in up to 9 different windows. Also, the sum of these products will be accumulated to multiple indices in the output matrix.. The result is repetition of multiplications and additions. The challenge is to reduce the required number of operations without consuming much memory for temporary results or requiring inter-task synchronization that could jeopardize effective parallelization. Given that the different products impact different indices in the output matrix, the latter poses an additional challenge, namely to find the contribution pattern for a single product.

In Fig. 62, the dashed curve shows the number of elements in the estimated covariance matrix, and the solid curve shows the required number of floating point operations (multiplications as well as additions), both versus a square-window size. For this figure assume that the input is a 128×128 matrix. Note that the number of floating point operations is considerably high due to redundant operations (both multiplications and additions), and the curve for the number of operations vs. window size has a bell like shape. The bell shape stems from the fact that the number of operations is the product of the operations per window positions and the number of such positions. When the window is very small, the former is very small; when it is very large, the latter is very small; the maximum is achieved with intermediate windows sizes.

The number of multiplications required by the straightforward algorithm is:

$$(N_r - S_r + 1) \cdot (N_c - S_c + 1) \cdot S_r^2 \cdot S_c^2. \tag{17}$$

This is also the number of additions required by the straightforward algorithm, because each multiplication is carried out in the course of computing a sum of products.

9.1.5 Parallelization Challenges

Effective parallelization of the serial algorithm presents several challenges: 1) obviating the need for synchronization and atomic instructions, 2) limiting the required amount of memory for intermediate results, 3) utilizing all cores all the time (efficient load balancing), and 4) avoiding redundant computations. These must all be addressed concurrently.

Various intuitive parallelization approaches fail to meet all challenges. For example, assigning a different row of the output matrix to each core would result in redundant operations. An additional approach might be to give each of the P cores

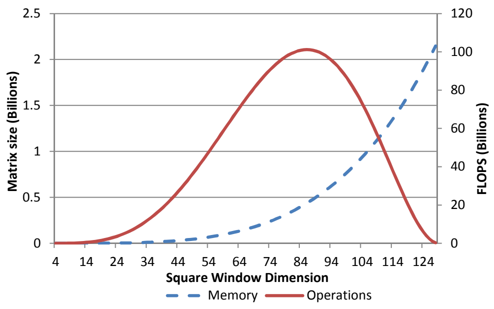


Figure 62: Given a 128×128 input window, this chart shows the expected number of operations and amount of memory required as a function of a square window. The x-axis denotes the window length=width. The dashed curve denotes the number of elements in the estimated covariance matrix. The solid curve denotes the number of floating operations need to compute the estimated covariance method for a given problem size using the straightforward formulation.

a near-equal number of windows to compute. In this approach, each core maintains a temporary output array for the accumulation process. When all the cores have completed, the temporary output arrays are summed up. This approach increases the memory requirements by a factor of P, which limits the problem size to which the algorithm can be applied, be it directly because of memory size or due to very poor memory access time in view of insufficient cache size and very slow execution as a result. Furthermore, this approach does not avoid redundant computations. A different approach would be to assign any given product to a single task that would add the product to the multiple temporary output matrices. This approach requires synchronization and possibly atomic operations, which also reduces the scalability of the algorithm and can limit portability.

For algorithms such as MVM [48], where a large number of estimated covariance matrices need to be computed, a coarse grain approach can be taken whereby each core is responsible for computing the estimated covariance matrix for a different input. This, however, increases the memory footprint, and does not address the problem of redundant operations in the computation of any given matrix.

9.2 Fine-Grain Multiplication - Efficient Parallelization

The key contribution of [47, 76] was the discovery and formulation of interesting relationships between relative positions of input-matrix elements, dubbed *combinations*, pairwise products, and positions (indices) in the output matrix. Based on these, [76] went on to propose a partitioning of the output-matrix indices amongst cores such that any given index is constructed by a single core, thereby obviating the need for inter-core synchronization; yet, each core is assigned the union of the indices to which any of the products that it computes contributes, obviating the need to repeat any multiplication by multiple cores or to require cores to synchronize or communicate with one another. Although this algorithm significantly reduces the number of multiplications, however, it does not reduce the required number of additions. This algorithm offers the speedup resulting from parallelism, enabling a reduction in the latency of computing the output matrix. However, its contribution to computation throughput (when multiple independent matrices need to be computed) is limited, despite the savings in multiplications, because of the additions. The sequential speedup (relative to the straightforward implementation) on an X86 platform is in on the order of $2X - 4X$, representing the savings in multiplications, an improved memory access pattern and a reduced memory footprint.

In the remainder of this section, we briefly describe this algorithm, which also serves as a starting point for our new algorithm. We include a summary of the key lemmas without any formal proofs. This section introduces relevant definitions that are also required by our new algorithm. The interested reader is referred to [76].

9.2.1 Product Based Partitioning

The challenge addressed in [47, 76] was partitioning the products among the cores such that any given output-matrix index is constructed by a single core, each product is computed only once and contributes to all the windows containing it, and cores need not communicate with one another or share data. The products are partitioned such that the relative position (row distance, column distance) of the elements of all products in any given group is the same, regardless of the window.

Definition 3 *Let A_{r_1,c_1} and A_{r_2,c_2} be two elements in the input matrix. Their inter-element distance (vector) is defined as:*

$$\Delta \triangleq (\Delta r, \Delta c) = (r_2 - r_1, c_2 - c_1). \tag{18}$$

Definition 4 *A* **combination** *is the set of all products of two input-matrix elements with the same inter-element distance; it is denoted by this distance, Δ, which must satisfy the following two conditions:*

$$-(S_r - 1) \leq \Delta r \leq S_r - 1 \wedge -(S_c - 1) \leq \Delta c \leq S_c - 1. \tag{19}$$

(The restrictions on the distances reflect the fact that we are only interested in products of elements that can be within the same $S_r \times S_c$ window. For example, the products of element pairs $(A_{3,3}, A_{4,4})$ and $(A_{5,5}, A_{6,6})$ both belong to combination $(\Delta r, \Delta c) = (1, 1)$. Reversing the order of a pair, e.g., $(A_{4,4}, A_{3,3})$, gives products that belongs to the combination $(\Delta r, \Delta c) = (-1, -1)$.)

Given that both products must be within the window, we define two disjoint sets of combinations. The first set comprises all the combinations wherein the first multiplicand is located at the top left corner of the window and the second element may be anywhere in the window. There are $S_r \cdot S_c$ possible positions of the second element. Each of these positions creates a different combination. The top left corner of the window is used for the purpose of finding the different combinations, whereas the position of the other multiplicand relative to the top left corner determines the indices in the output matrix to which this product will be added (accumulated). This first set of combinations, denoted POS, is formally defined as:

$$POS = \{(\Delta r, \Delta c) | (0 \leq \Delta r \leq S_r - 1) \wedge (0 \leq \Delta c \leq S_c - 1)\}. \tag{20}$$

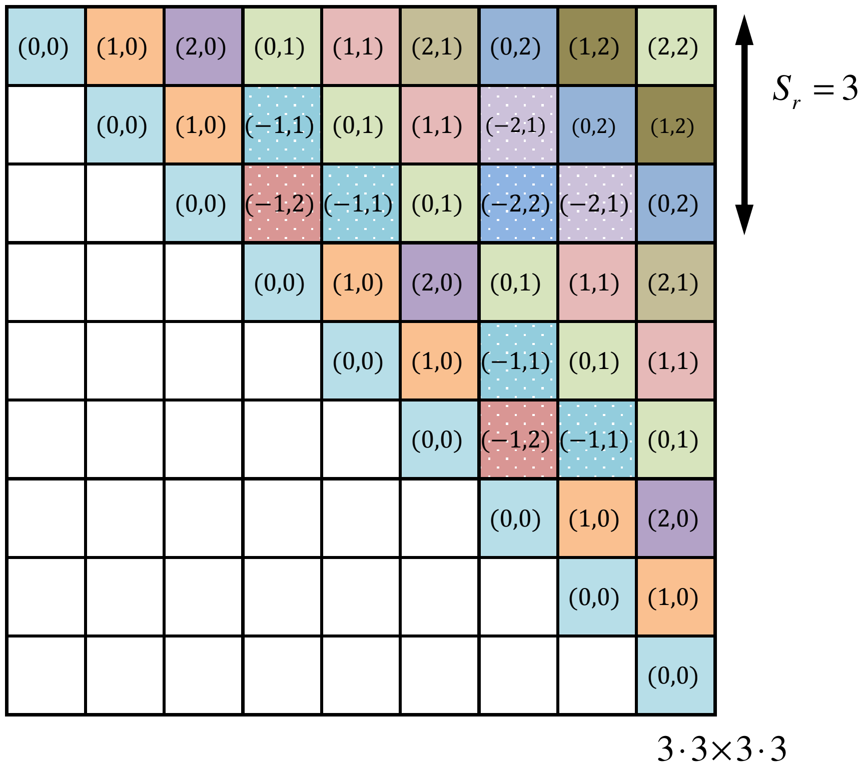
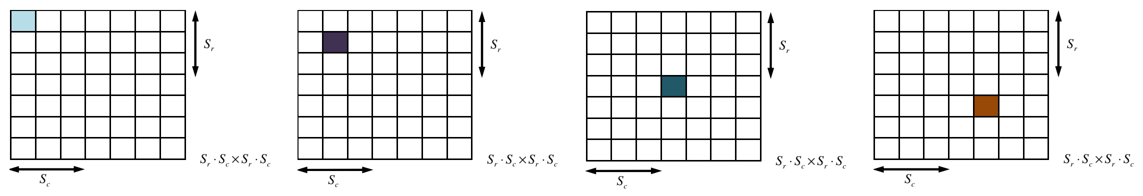


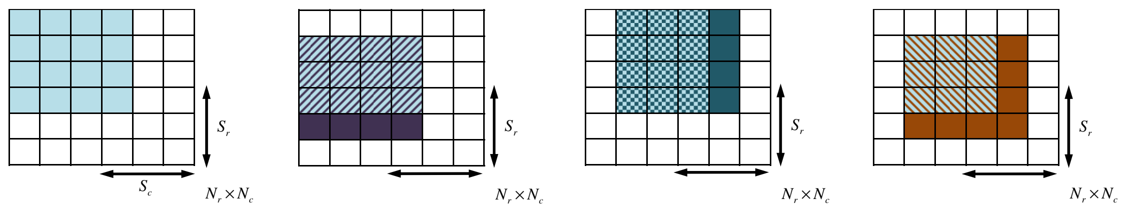
Figure 63: The following 9×9 output matrix is for a 3×3 sliding window. The output matrix is partitioned into the different indices of the combinations. Different combinations affect non overlapping sections of a specific diagonal. Each combination is depicted using a different color. The tuples in the matrix refer to the the combination. All the combinations in POS have a solid colored background and the combinations of NEG have white dots in the background.

The second set comprises all combinations wherein the first element is in the first column and the second element is to the right and above the first element (rather than in the same row or below as in POS). There are $S_r - 1$ possible ways to place the first element. For each of these, the second element can be in $S_c - 1$ different places. This allows for a total of $(S_r - 1) \cdot (S_c - 1)$ different combinations. This second set of combinations, denoted NEG, is formally defined as:

$$NEG = \{(\Delta r, \Delta c) | (-(S_r - 1) \leq \Delta r \leq -1) \wedge (1 \leq \Delta c \leq S_c - 1)\}. \quad (21)$$



(a) Top-left (b) First-row (c) First-per-block (d) Remaining-indices



(e) Top-left (f) First-row (g) First-per-block (h) Remaining-indices

Figure 64: Product centric approach for combination $(\Delta r, \Delta c) = (0, 0)$. These windows slightly vary (different height and width) for different combinations. (a)-(d) refer to the exact location in the output matrix. (e)-(h) refer to the set of products that are needed in order to compute the indices in (a)-(d).

Let UC denote the set of unique combinations. Note that for all the combinations in UC, $\Delta c \geq 0$, whereas Δr can be negative, zero or positive. Based on the observation that the order of the element-pair affects the distance, an immediate question arises as to why the combinations in which $\Delta c < 0$ are not included in UC. The reason is that the estimated covariance matrix is Hermitian, so it suffices to compute the upper triangle or lower triangle sub-matrices. Any two inverse-distance combinations (e.g. $(\Delta r, \Delta c) = (a, b)$ and $(\Delta r, \Delta c) = (-a, -b)$) consist of the same element pairs. As the multiplication is done by taking the conjugate of one of the elements, we can utilize the trivial identity $A_{x',y'} \cdot \overline{A_{x,y}} = \overline{A_{x,y} \cdot \overline{A_{x',y'}}}$ to further reduce the number of actual multiplications. As can be seen, Δc is always positive, so the second element of any product is to the right of the first element; consequently, all results are written to the upper triangle. The products belonging to $\Delta c < 0$ combinations, which "own" the lower triangle, need not be computed.Accordingly, the set of unique combinations is specified by the union of the two sets:

$$UC = POS \cup NEG. \tag{22}$$

The total number of combinations is:

$$|UC| = |POS| + |NEG| = S_r \cdot S_c + (S_r - 1) \cdot (S_c - 1). \tag{23}$$

In [76], the following claims are proved::

1) The combinations jointly cover the upper triangle of the output matrix.

2) Collision Freedom - the sets of indices affected by two different combinations are disjoint.

3) The indices affected by any given combination all lie along the same diagonal (Fig. 63).

The target indices of the combinations thus jointly cover the entire upper triangle of the output matrix, and any given index is affected to by a single combination. Additionally, no synchronization is required among the cores. By assigning any given combination and the associated output indices to a single core, it follows that all combinations can be computed in parallel with no need for synchronization among the threads and with no redundant multiplications.

9.3 The New Algorithm

The approach of [47, 76] can be viewed as product centric. Each product is computed exactly once, and is then added to all the output-matrix indices to which it contributes (i.e., belonging to its combination). This was done by sliding a window of the same dimensions as W around the product. Consequently, while the algorithm is parallelizable (by combination) and number of multiplications was held to the bare minimum, the number of additions was not reduced relative to the straightforward approach.

Algorithm 16: Parallel algorithm for computing the combinations in POS. The combinations in NEG are computed in a similar fashion.

```
$D_r \leftarrow N_r - S_r + 1$;
$D_c \leftarrow N_c - S_c + 1$;
**for** $(\Delta r, \Delta C) \in POS$ ***in parallel*** **do**
    $ind_r \leftarrow +1$;
    $ind_c \leftarrow S_r \cdot \Delta c + \Delta r + 1$;
    // Top-left
    **for** $r = 1$ *to* $(N_r - S_r + 1)$ **do**
        **for** $c = 1$ *to* $(N_c - S_c + 1)$ **do**
            $C_{ind_r,ind_c} \leftarrow C_{ind_r,ind_c} + A_{i,j} \cdot \overline{A_{i+\Delta r,j+\Delta c}}$;
    // First-block
    $ind_r \leftarrow 2$;
    $ind_c \leftarrow S_r \cdot \Delta c + \Delta r + 2$;
    $sum \leftarrow 0$;
    **for** $r = 2$ *to* $E(\Delta r)$ **do**
        **for** $c = 1$ *to* $D_c$ **do**
            $sum \leftarrow sum + A_{D_r+r,c} \cdot \overline{A_{D_r+r+\Delta r,c+\Delta c}} - A_{r-1,c} \cdot \overline{A_{r-1+\Delta r,c+\Delta c}}$;
        $C_{ind_r,ind_c} \leftarrow C_{ind_r-1,ind_c-1} + sum$;
        $ind_r \leftarrow ind_r + 1$;
        $ind_c \leftarrow ind_c + 1$;
        $sum \leftarrow 0$;
    // First-Per-Block
    $ind_r \leftarrow S_r + 1$;
    $ind_c \leftarrow (S_r + 1) \cdot \Delta c + \Delta r + 1$;
    $sum \leftarrow 0$;
    **for** $c = 2$ *to* $B(\Delta c)$ **do**
        **for** $r = 1$ *to* $D_r$ **do**
            $sum \leftarrow sum + A_{r,DC+c} \cdot \overline{A_{r+\Delta r,DC+c+\Delta c}} - A_{r,c-1} \cdot \overline{A_{r+\Delta r,c-1+\Delta c}}$;
        $C_{ind_r,ind_c} \leftarrow C_{ind_r-S_r,ind_c-S_r} + sum$;
        $ind_r \leftarrow ind_r + S_r$;
        $ind_c \leftarrow ind_c + S_r$;
        $sum \leftarrow 0$;
    // Remaining-indices
    $ind_r \leftarrow S_r + 2$;
    $ind_c \leftarrow S_r \cdot \Delta c + S_r + \Delta r + 2$;
    **for** $r = 2$ *to* $E(\Delta r)$ **do**
        **for** $c = 2$ *to* $B(\Delta c)$ **do**
            $a \leftarrow A_{r-1,c-1} \cdot \overline{A_{r-1+\Delta r,c-1+\Delta c;}}$
            $b \leftarrow A_{r-1,D_c+c} \cdot \overline{A_{r-1+\Delta r,D_c+c\Delta c;}}$
            $c \leftarrow A_{D_r+r,h-1} \cdot \overline{A_{D_r+r+\Delta r,c-1+\Delta c;}}$
            $d \leftarrow A_{D_r+r,D_c+c} \cdot \overline{A_{D_r+r+\Delta r,D_c+c\Delta c;}}$
            $\Delta sumrow \leftarrow C_{ind_r-S_r,ind_c-S_r} - C_{ind_r-S_r-1,ind_c-S_r-1;}$
            $C_{ind_r,ind_c} \leftarrow C_{ind_r-1,ind_c-1} + \Delta sumrow + a - b - c + d$;
            $ind_r \leftarrow ind_r + 1$;
            $ind_c \leftarrow ind_c + 1$;
        $ind_r \leftarrow r \cdot S_r + 2$;
        $ind_c \leftarrow (S_r) \cdot \Delta c + r \cdot S_r + \Delta r + 2$;
```

In this section we present our new algorithm. We use the combination-based partitioning of [76], so parallelizability is retained. Our focus is on more efficient computation of the output matrix elements whose indices belong to a common combination. Specifically, our aim is to reduce the required number of additions. In the sequel, we consider a single combination and the computation of the output-matrix elements (indices) that belong to it.

Instead of considering each product and the indices to which it contributes, we consider an index (position) in the output matrix and all the products that contribute to its value. We show that there is a substantial overlap between the sets of products contributing to different same-combination indices, and that this overlap has a repetitive pattern that is somewhat similar to the Inclusion-Exclusion principle. Based on this, we devise a scheme for incrementally computing an index by starting from the value of the previously computed (same-combination) one and then adding and subtracting the non-overlapping products. In fact, we mostly do not need to add and subtract each non-overlapping product separately; instead, partial sums can be manipulated. We refer to this as a *product window* approach.

9.3.1 Combination $(\Delta r, \Delta c) = (0, 0)$

As an example, consider the combination $(\Delta r, \Delta c) = (0, 0)$. This subsection gives the intuition behind the computation; this will subsequently be formalized, with Algorithm 16 stating the exact computation for $(\Delta r, \Delta c) = (0, 0)$ and more. The indices of $(\Delta r, \Delta c) = (0, 0)$ are the indices that make up the main diagonal, Fig. 63.

Consider the index $C_{1,1}$in the output matrix; it is denoted in light blue in Fig. 64(a). Note that this is the top left index of the output matrix. Remember that this index is the sum of all the top left indices for all the temporary matrices $C^{k,l}$. Specifically, this index equals the sum of the products of $V^{k,l}(1,1) \cdot \overline{V^{k,l}(1,1)}$ of all the windows. In Fig. 64(e), using light blue, we marked all the elements in the input matrix that are needed for computing $C_{1,1}$. Remember that each element is multiplied by its conjugate. For a different combination than (0,0) , Fig. 64(e) would be of a different dimension, as the number of window positions changes. Given that the sliding window must stay within the bounds of the input array, the different number of windows is $(N_r - (S_c - 1) \cdot (N_c - (S_c - 1)))$. Note that the matrices in the (a) and (e) do not have the same dimensions.

Now consider the (vector) multiplication of $V^{k,l}(2,1) \ \cdot \overline{V^{k,l}(2,1)}$. These products, for the different windows, also belong to combination (0,0). The sum of these products is written to $C_{2,2}$, as depicted in Fig. 64(b) marked in dark purple. As for the previous index $C_{1,1}$, all the needed products in the input array are marked in dark purple, Fig. 64(f). The products that are common to indices $C_{2,2}$ and $C_{1,1}$ are marked with blue and purple diagonal stripes.

The difference between the sums constituting these two indices is the sum of the first row from Fig. 64(e), which must be removed (subtracted), and the sum of the newly added row from Fig. 64(f), which must be added. This is repeated for all the product windows wherein the leftmost product is in the first column of the input matrix. There are $S_r - 2$ such windows, given that the first and top-left most window requires computing all the products. For all $1 \leq g \leq (S_r - 2)$, the following can be stated:

$$C_{g+1,g+1} = C_{g,g} + \sum row_{new} - \sum row_{old}. \tag{24}$$

Note that $C_{g+1,g+1}$ depends on $C_{g,g}$ which depends on $C_{g-1,g-1}$. Therefore, it is preferable to compute in the order $C_{1,1}, C_{2,2}, C_{3,3}, ..., C_{S_r-1,S_r-1}$.

Similarly, moving the product window to the right allows computing additional elements of the same combination. The first move of the product window computes C_{S_r+1,S_r+1} as depicted in Fig. 64(c). Each additional move of the product window to the right will compute elements of the following format: $C_{S_r \cdot h+1,S_r \cdot h+1}$ for $1 \leq h \leq S_c - 2$. These indices can be computed as follows:

$$C_{S_r \cdot h+1, S_r \cdot h+1} \quad = \quad C_{S_r \cdot (h-1)+1, S_r \cdot (h-1)+1} + \sum column_{new} - \sum column_{old}. \quad (25)$$

Again, it is preferable to compute in the order $C_{1,1}, C_{S_r+1,S_r+1}, C_{2 \cdot S_r+1, 2 \cdot S_r+1}, \ldots,$ $C_{(S_c-1)\cdot S_r-1,(S_c-1)\cdot S_r-1}$.

Up to now, we have shown how to compute $S_r + S_c - 1$ indices of a specific combination. There still remain $S_r \cdot S_c - (S_r + S_c - 1)$ indices that need to be computed for the combination. These indices could be computed using the same techniques as discussed. However, there is a less computationally demanding approach to compute these indices, using additional overlapping information. In Fig. 64(h) we see the products common to the index in Fig. 64(d) and to $C_{1,1}$. Note that the overlap between the turquoise product-window in Fig. 64(g) and the brown product-window in Fig. 64(h) includes everything except for the top row and bottom row. In Fig. 65, the overlap of the same product window from Fig. 64(h) is shown with overlap of the additional product windows. The value of C_{S_r+2,S_r+2} includes products that are not part of the product window of C_{S_r+1,S_r+1}. Note that there is only a single product that was not computed as part of the other product windows. We show that using an inclusion-exclusion like principle we can compute this product window using previously computed values. We show this principle in steps. Given the overlap with the product window of C_{S_r+1,S_r+1}, we add this value to C_{S_r+2,S_r+2}. Obviously some corrections need to be made to this sum, as the first row (light blue) need not be considered and the last row (purple) needs to be considered. Note that the first row has one turquoise product in addition to the light blue products and the new row has one brown in addition to the purple products. We make the corrections in two phases. Similar to the process that was shown earlier, the sum of the new purple row minus the sum of the first light blue row is computed as follows:

$$\Delta sum_{row} = C_{2,2} - C_{1,1}. \quad (26)$$

We handle entire row and column sums, so "corner" elements are handled twice. These must be handled individually. We next address this issue in detail. In the process of computing Δsum_{row}, it was assumed that the entire row overlaps with the turquoise product-window, when in fact the top-left element denoted as a in Fig. 65 is not part of the overlapping window. Given that it has been subtracted, it needs to be added back to the sum. The first product of the bottom purple row was also added to the sum. Therefore, it needs to be subtracted, this product is denoted c. a and c have corrected the summation process of (26), but two additional corrections remain to be considered. By using the product-window of C_{S_r+1,S_r+1}, an additional error was introduced into the summation process; this is the top-right turquoise product denoted as b in Fig. 65. This can be corrected by subtracting this product from the final sum. Finally, the brown product window has a new value that does not overlap with any previous windows; this is denoted as d in Fig. 65. This value has to be added to the final summation. In summary, C_{S_r+2,S_r+2} can be written as follows:

$$C_{S_r+2,S_r+2} = C_{S_r+1,S_r+1} + \Delta sum_{row} + a - b - c + d \tag{27}$$

For C_{S_r+2,S_r+2}, a, b, c, and d are defined as follows:

$$a = A_{1,1} \cdot \overline{A_{1,1}}, \tag{28}$$

$$b = A_{1,N_c-S_c} \cdot \overline{A_{1,N_c-S_c}}, \tag{29}$$

$$c = A_{N_r-S_r,1} \cdot \overline{A_{N_r-S_r,1}}, \tag{30}$$

and

$$d = A_{N_r-S_r,N_c-S_c} . \overline{A_{N_r-S_r,N_c-S_c}}. \tag{31}$$

This overlap procedure is the same for all remaining indices of the combination. Algorithm 16 provides the exact "recipe".

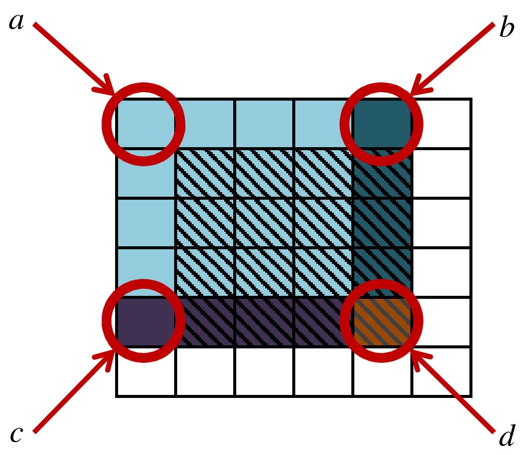
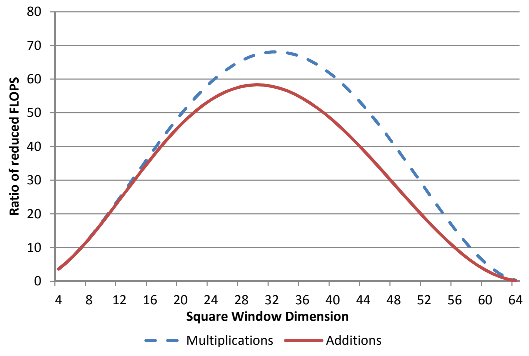
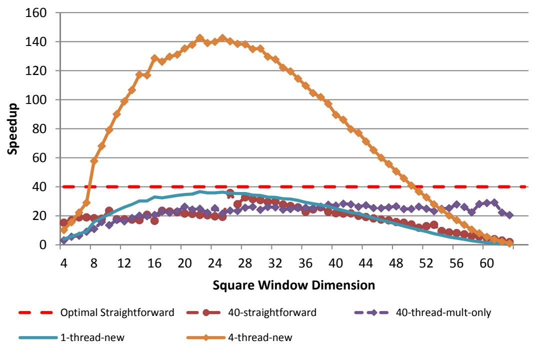


Figure 65: Zoom-in on the overlapping windows. a, b, c, and d denote unique elements that need to be added/subtracted individually with the new approach.



(a)



(b)

Figure 66: Given a 64×64 input matrix: (a) Reduction (factor) in additions and multiplication relative to the straightforward algorithm. (b) Actual speedups based on the execution of three different parallel algorithms (with different thread counts): straightforward, multiplication-reducing only algorithm [47, 76], and the new algorithm. No more than 4 threads are displayed for the new algorithm for figure-scaling purposes. Its speedup scales almost linearly to 40 threads.

9.3.2 Remaining Combinations

In this subsection, we discuss how to compute the remaining combinations.

Definition 5 *The number of blocks in combination*

$(\Delta r, \Delta c)$ *is* $B(\Delta c) = S_c - |\Delta c|$.

Definition 6 *The number of rows in a block in combination* $(\Delta r, \Delta c)$ *is* $E(\Delta r) = S_r - |\Delta r|$.

Element type	***Multiplications***	***Additions***	***# elements of type***
Top-left	$(N_r - S_r + 1) \cdot (N_c - S_c + 1)$	$(N_r - S_r + 1) \cdot (N_c - S_c + 1)$	1
First-block	$2 \cdot (N_c - S_c + 1)$	$2 \cdot (N_c - S_c + 1)$	$E(r) - 1$
First-per-block	$2 \cdot (N_r - S_r + 1)$	$2 \cdot (N_r - S_r + 1)$	$B(c) - 1$
Remaining-indices	4	7	$(E(r) - 1) \cdot (B(c) - 1)$

Table 19: The number of operations needed for each combination based on the four types of product-window shift.

Example: for the $(\Delta r, \Delta c) = (0, 0)$ example, $B(0) = S_c$ and $E(0) = S_r$.

Each combination can be divided into four unique groups:

Definition 7 *We denote the four different computation scenarios as Top-left, First-block, First-per-block, Remaining-indices.*

Pseudo-code for computing the combinations in POS can be found in Algorithm 16. Most of the explanations for computing these combinations are similar to the explanation given for combination $(\Delta r, \Delta c) = (0, 0)$. Table 19 presents the number of operations for each combination based on the foregoing definitions.

For the sake of brevity, we only provide necessary observations for computing the remaining combinations. The first group, Top-left, comprises a single index in the output matrix. The index of the Top-left depends on the combination and the window dimensions. Computation of this index appears in Line 3 of Algorithm 16. This single index can be more computationally demanding than computing all the remaining indices of a combination, because here we prepare the partial sums that are later used to incrementally modify the value of one index in order to obtain that of the next one. This will be discussed in depth in the next subsection. The Top-Left index dictates which diagonal in the output matrix will be affected by the combination. The Top-Left for combinations in NEG will be computed slightly differently, but it too designates the target diagonal.

9.3.3 Complexity Analysis

In this section we analyze the work complexity of the new method. For simplicity, we analyze the complexity for the combinations in the set POS. The number of operations for a combination (a, b) is equal to the number of operations in combination $(a, -b)$. Also, $|POS| > |NEG|$, so that computing the combinations in POS is more work than computing those in NEG. Therefore, deriving the complexity of POS and doubling it yields a conservative approximation. We examine the four types of window shifts, and derive the complexity for each. Finally, we add up the complexities of the different window shifts and double it. For three out of the four types of shifts, the numbers of additions and multiplications are the same.

For the first type of indices, Top-Left, the number of multiplications over all the combinations is:

$$\sum_{\Delta r=0}^{S_r-1}\sum_{\Delta c=0}^{S_c-1}(N_r - S_r + 1)\cdot(N_c - S_c + 1)\,. \tag{32}$$

There are an equal number of additions. Note that the number of elements in the sum is independent of the values of Δr and Δc. Therefore, the number of operations is:

$$S_r \cdot S_c \cdot (N_r - S_r + 1)\cdot(N_c - S_c + 1)\,. \tag{33}$$

For the second type, First-block, the number of multiplications for each combination is $2 \cdot (N_c - S_c + 1)$, and this is summed over the combinations:

$$\sum_{\Delta r=0}^{S_r-1}\sum_{\Delta c=0}^{S_c-1} 2\cdot(N_c - S_c + 1)\cdot(E(\Delta r) - 1). \tag{34}$$

There are an equal number of additions. Once again using simple arithmetic, which includes the sum of an arithmetic series, the number of multiplications and additions is:

$$S_c \cdot S_r \cdot (N_c - S_c + 1)\cdot(S_r - 1)\,. \tag{35}$$

The number of multiplications and additions required by the third type, First-per-block, is computed in a similar fashion and is:

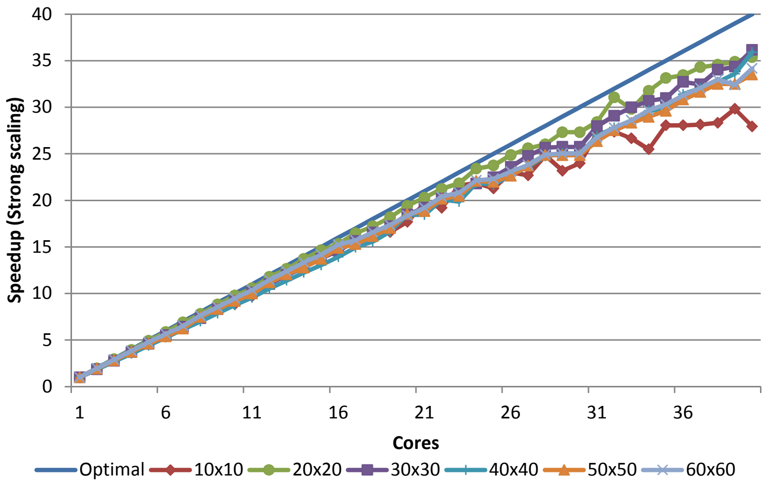


Figure 67: Strong scaling speedup of the new algorithm for multiple window sizes.

$$S_r \cdot S_c \cdot (N_r - S_r + 1) \cdot (S_c - 1)\,. \tag{36}$$

For the fourth type, Remaining-indices, we show the number of operations required by all the combinations. As the difference between the number of multiplications and additions is a constant, we use α to denote the constant. Upon completion, α can be substituted with 4 for multiplications (required for a, b, c, d) and 7 (a, b, c, d and the three other operands) for additions.

$$\sum_{\Delta r=0}^{S_r-1} \sum_{\Delta c=0}^{S_c-1} \alpha \cdot (E(\Delta r) - 1) \cdot (B(\Delta c) - 1). \tag{37}$$

This is reduced to:

$$\frac{\alpha}{4} \cdot S_r \cdot S_c \cdot (S_r - 1) \cdot (S_c - 1). \tag{38}$$

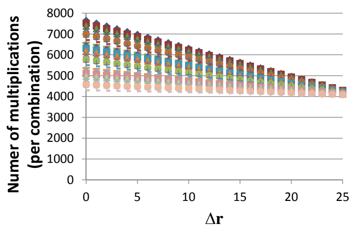
Finally we sum (33), (35), (36), (38) and multiply them by two:

$$
\begin{aligned}
Total = 2 \cdot (S_r \cdot S_c \cdot (N_r - S_r + 1) \cdot (N_c - S_c + 1) +& \\
S_c \cdot S_r \cdot (N_c - S_c + 1) \cdot (S_r - 1) +& \\
S_r \cdot S_c \cdot (N_r - S_r + 1) \cdot (S_c - 1) +& \\
\frac{\alpha}{4} \cdot S_r \cdot S_c \cdot (S_r - 1) \cdot (S_c - 1)).& \quad (39)
\end{aligned}
$$

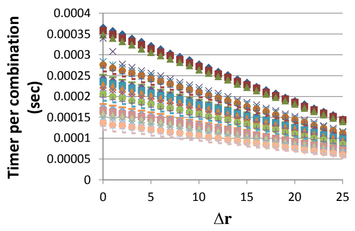
Fig. 66(a) shows the ratio between the number of operations required by the straightforward approach as given in (17) and those required by the new approach for both multiplications and additions. This ratio is also indicative of the possible sequential speedup. Note that these curves have the same bell like shape as the straightforward algorithm, Fig. 62. In the Results section, we confirm that the speedups achieved by the new algorithm for both single-core and multi-core follow this curve.

9.3.4 Additional Implementation Details

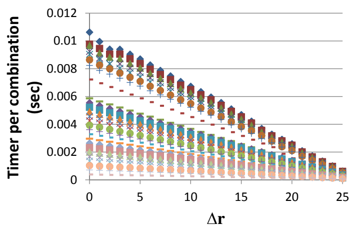
In [76], several algorithmic optimizations were presented that are also somewhat relevant to this algorithm. The first is that the results are not accumulated into the final output matrix, but rather to a temporary array before being written to the final output matrix. The motivation for this is that in the older algorithm the indices of a given combination were accessed numerous times. Given that a combination accesses indices on a given diagonal, the older algorithm had a non-sequential access pattern to the output array, causing poor cache performance. By using a temporary sequential (one dimensional) array of size $S_r \cdot S_c$ for each thread, this bad access pattern is avoided. Given that the largest combination writes to $S_r \cdot S_c$ indices, this is an upper bound on the memory used by a given thread. This increases the memory requirement by a total of $p \cdot S_r \cdot S_c$, but improves cache performance. This increase is small relative to the size of the output matrix, $S_r \cdot S_c \times S_r \cdot S_c$. The memory usage



(a)



(b)



(c)

Figure 68: Given a 64×64 input matrix with a 26×26 window: (a) theoretical number of multiplications for each combination of the new algorithm, (b) run times for each combination of the new algorithm, and (c) run times for each combination using the previous combination-based algorithm [47]. There are 676 combinations that are presented in all these sub-figures. Note the different units and scales of the ordinate for the different sub-figures.

can be reduced even further, such that each combination allocates the exact amount of memory it needs. For our new algorithm, this non-sequential access pattern is not an issue, as each index is written to exactly once. The pseudo code in Algorithm 16 assumes that the writing is done to the final output matrix. In reality the writing is done to temporary arrays for better performance. For presentation purposes we do not use the temporary arrays.

The algorithm in [76] computes an optimal number of multiplications. Our new algorithm computes some products more than once - as such the new algorithm is not optimal. Our new algorithm does about two to four times more multiplications than optimal. The new algorithm can be optimized to compute the optimal number of multiplications at the cost of adding algorithmic overhead, but this is beyond the scope of this work.

9.4 Experimental Results

In this section we present performance measurements for the new algorithm, focusing on its scalability to multiple compute cores.

9.4.1 Experimental Setup

We use a a 4-socket Intel multicore system, each containing an Intel Xeon E7-8870 10-core hyperthreaded 2.4GHz processor with a 30MB L3 cache. There are thus 40 physical cores and support up to 80 logical cores. In our tests, we did not use the HyperThreading option. All the algorithms were implemented in C and used OpenMP. The server is equipped with 256 GB of 1066 MHz DDR3 DRAM. Our algorithm was implemented such that intermediate results are stored in dense temporary arrays rather than in the output matrix, thereby increasing cache efficiency.

We show strong scaling results for the different algorithms relative to their sequential implementation. Strong scaling results are for a given input size and window size where the number of cores is increased. We also show the performance of these algorithms relative to the sequential implementation of the straightforward dense vector-vector multiplication approach. We implemented the straightforward algorithm using Intel's MKL [8] and received a performance gain from these optimized functions. The MKL library has many optimized kernels for matrix multiplication that use SIMD. The speedup attained from MKL for a single core is approximately 2X for a single thread of the straightforward implementation. For 40 threads there was a very small performance gain of using MKL over a straightforward implementation. All the algorithms, including our new algorithm, would benefit from SIMD multiplications and additions. These low-level optimizations were outside the scope of this work. For this reason, we compare our new algorithm to the straightforward implementation. We do our best to report the best possible execution times for these algorithms, and note that all the algorithms benefit from the same compiler optimizations.

9.4.2 Moderate-Size Problems

Initially, we show that the single core execution of the new algorithm behaves as expected. Fig.66(b) depicts the speedup of the parallel straightforward algorithm and of

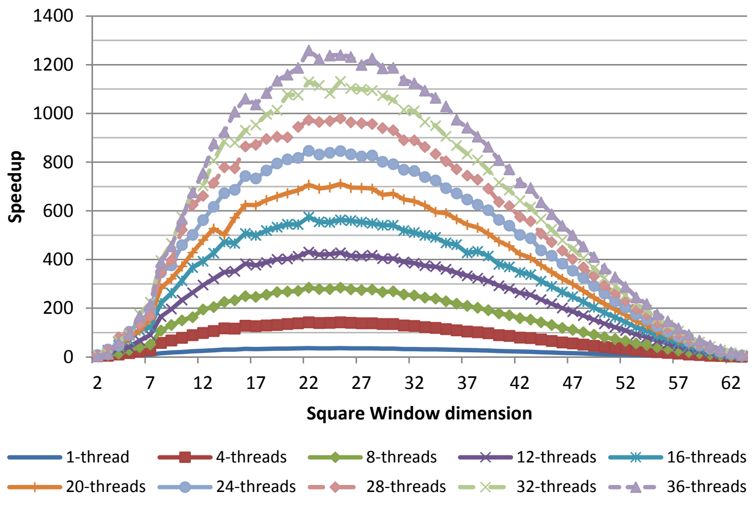


Figure 69: Speedup of the our new algorithm for a 64×64 input matrix with square windows relative to the straightforward sequential algorithm for with the same window size. x-axis is the window size. The speedup curves are in multiple of 4 threads.

the new algorithm with several thread counts relative to the sequential straightforward algorithm (dense vector-vector multiplication). The curves indeed reflect the reduction in the number of operations as depicted in Fig.66(a). The figure also depicts the parallelization speedup of the straightforward algorithm. (It is trivially parallelizable because the computations for each index are carried out “from scratch”, not using any partial results.) The maximum possible speedup of this algorithm is 40X (limited by the number of cores), but it is not attained, possibly due to poor cache performance. In fact, the single-core (sequential) execution of the new algorithm outperforms the 40-core execution of the straightforward algorithm. The speedup of the algorithm from [47, 76] is also presented.

Fig. 67 depicts the strong scaling of the new algorithm. Six different window sizes were selected: $10 \times 10, 20 \times 20, 30 \times 30, 40 \times 40, 50 \times 50$, and 60×60. While the strong scaling is not perfectly linear, 85%-90% of maximum system utilization is achieved. The actual run times on the 40-core system are considerably short for the new algorithm, from 0.7ms for the 10×10 window and up to 77ms for the 60×60. With up to 35 cores, the execution times were reasonably consistent for the smaller

Input Size	Window Size	Straightforward 40 threads (sec.)	New algorithm 40 threads (sec.)	Speedup
$0.5k \times 0.5k$	50×50	341	1.07	$321X$
$0.5k \times 0.5k$	100×100	2965	4	$1121X$
$1k \times 1k$	50×50	2525	3.225	$463X$
$1k \times 1k$	100×100	—	17	—
$32k \times 32k$	50×50	—	2584	—
$32k \times 32k$	100×100	—	10530	—

Table 20: Performance analysis of larger instances. Several cases timed out for the straightforward algorithm timed out.

window sizes. With more cores, carefully timed runs were necessary given that even a brief system call can significantly change the execution times. This was not a problem for the other algorithms as they are considerably slower, so a system call doesn't significantly affect their timing .

The scalability of the new algorithm is due to its intrinsically balanced load, improved over that of [47] and to the elimination of nearly all redundant additions.. In Fig. 68(a) we show the number of computations required by the POS combinations of a 26×26 window for a 64×64 input matrix. The x-axis represents the Δr for each of the combinations. For each of Δc, we plotted the number of operations required by the combination. As such, there are 26 curves in the graph, each with 26 points. Note that the ratio between the most computationally demanding combination and the least computationally demanding one is less than 2. This imbalance will only be felt when the number of cores is on the same order of magnitude as the number of combinations. For verification purposes, we timed the computation of each combination; these are depicted in Fig.68(b). Note the 6:1 ratio of actual execution times, larger than the ratio of the number of operations but still moderate, becoming insignificant when there are many more combinations than cores.

Fig. 68(c) depicts the execution times of the combinations for the older algorithm [47]. Clearly, the additions are the bottleneck. Also, the ratio between the most computationally demanding combination (taking 10.6 ms) and the least computationally demanding combination (0.057 ms) is 186:1, which causes load balancing problems for this algorithm [47]. This is due to the greater variability in the number of additions (when carried out from scratch) among combinations than the variability in the number of products. This problem is discussed in depth in [47, 76]. The execution times for the combinations using this algorithm are depicted in Fig.68(c).

Fig. 69 depicts the speedup of the new algorithm versus a square window size. Curves are provided for thread counts in increments of four. Also, the curves are nearly equally separated, which can be expected in view of the near linear scaling.

9.4.3 Scaling Up

We have thus far evaluated our algorithm relative to prior art for considerably small (albeit relevant to some applications) input sizes for which the previous algorithmsâĂŹ completion times were still reasonable. We now consider larger problem sizes. From a practical perspective, both the input-matrix size and the rate at which matrices must be computed are relevant. Different matrices can be trivially processed concurrently by different threads/cores. However, so doing on a multi-core machine with a shared cache increases the memory footprint and can reduce cache performance, so even this is not trivial. In any case, computational efficiency is important.

Consider, for example, the MVM algorithm [48]; here, 10k estimated covariance matrices need to be computed for every image, and images arrive are streamed from the sensors.

Let us next consider larger matrices. Computation time for those are provided in Table 20. For both the straightforward algorithm and our new algorithm, all 40 threads of the system are used. For typical image sizes of 500×500 pixels, our new algorithm completes the computation in a matter of seconds whereas the straightforward algorithm takes minutes or even up to an hour. For larger inputs such as $32k \times 32k$, straightforward algorithm simply times out. Assuming a $300X$ speedup of the new algorithm over the baseline, the straightforward algorithm would require

215 hours (nearly 9 days) and 877 hours (nearly 36 days) to compute the 50×50 and 100×100 , respectively. This is most likely an under-estimation, as the likely speedup is higher for the 500×500 input matrix and the speedup increases with the size of the input due to reduction in operations. These considerably smaller computation times allow analyzing larger inputs that previously were not possible.

Even with our CPU- and memory-efficient algorithm, the time needed to compute the estimated covariance matrix for the large matrix is quite long, especially for applications with streaming data that are moreover latency bound. One example is satellite imaging, in which the satellites sensors are pointed based on the analysis of the image. Such applications require systems that are larger than a single wide shared memory system. Because our algorithm has virtually no synchronization between the threads except for the barrier at the end of the parallel-for loop, it can be easily distributed to larger system with a higher thread count even for a single matrix. The only cost for distributing this computation is creating a copy of the input matrix at each distributed node and the communication costs of transmitting it. The memory required for holding a $32k \times 32k$ input is approximately 4GB, which most HPC systems have per node. Sending this input array to all nodes is still a negligible effort relative than the computation time. Further, as each thread only requires a subset of the diagonal, the memory requirement per thread isn't significant. Using a window size of $13k \times 13k$, which is a 40% window size, allows creating over $100M$ independent threads.

9.5 Conclusions

In this chapter we presented a new approach for computing the estimated covariance matrix. It is dramatically more efficient than the dense vector-vector multiplication as expressed in the formulation of the Covariance Method. The new approach reduces the total number of floating point operations by almost completely avoiding redundant operations. Also, it requires fewer memory accesses as each datum is used fewer times. All this allows for a faster sequential algorithm - 35X faster than the straightforward algorithm, and 17X faster than the algorithm in [47], which minimizes the required number of multiplications but suffers from redundant additions.

The key improvement of the new algorithm over [47], is an incremental approach to computing the partial sums of products of input-matrix element pairs required for computing each output-matrix element. Doing so dramatically reduces the number of additions and subtractions, as well as the number of memory accesses.

In addition to saving multiplications, the main contribution of [47] was an elegant partitioning of the output-matrix elements among compute threads, such that no inter-thread synchronization is required. This permits fine-grain parallelism. The shortcoming of [47] is that it did not reduce the number of additions, which limited its sequential performance and also, due to a highly variable amount of work for different partitions, created a load-balancing problem for the parallel version.

The new algorithm adopted the partitioning of [47], so it is easily parallelizable. Moreover, the savings in additions and subtractions also sharply reduced the work-variability among partitions, so the parallelism translates more smoothly to high performance. With 40 single-thread compute cores, its parallel version is $1200X$ faster than the single-core implementation of the straightforward algorithm, and some $40X$ faster than [47] (based on the values taken from Fig. 66 for the older algorithm and Fig. 69 for the new algorithm).

The new algorithm thus apparently dominates the prior art. Moreover, the dramatic performance and efficiency improvements suggest that it may make the Covariance Method more broadly applicable, and in many applications it may be possible to execute it in real time, on mobile platforms, etc., with important impact on the usage mode of those applications. In fact, the problems sizes, be it input-image size or the rate at which images are provided, continuously increase. The fact that our algorithm can be parallelized efficiently, with the level of parallelism increasing with input size, makes it an excellent candidate for tackling these increasing challenges.

In this chapter, we focused on general-purpose multi-core architectures. However, the algorithm appears to lend itself to other platforms, with the extremely high count of independent threads that we were able to achieve while avoiding redundant computation, are generic properties. We therefore expect the benefits to carry over to additional platforms. Two interesting candidates are GPGPUs and FPGA; the adaptation of our algorithm to those platforms is left for future research.

CHAPTER X

CONCLUSIONS

10.1 Impact of Thesis

This thesis focused on improving performance for irregular algorithms, specifically (but not exclusive to) social network analytics by designing new algorithms that increase the parallel scalability and have the ability to deal with streaming input for larger data-sets than was previously possible. Many of our new algorithms focus on avoiding wasteful computations, load-balancing, better memory usage, and utilizing massively multi-threaded systems.

The outcome of this thesis has brought us several steps closer to online analysis of social networks. The next section briefly summarizes the contributions of this thesis. This is followed by a section that shows the relationships (connection) between the multiple analytics and algorithms. This be followed with a discussion on what still remains ahead of us and what we can still do to improve our computational capabilities.

10.2 Summary of Results

This thesis showed multiple contributions for the efficient computation of betweenness centrality. Initially we showed a novel approach for computing betweenness centrality in dynamic graphs. The new dynamic graph algorithm is less computationally demanding than doing a full static graph computation. The amount of work required by the new dynamic graph algorithm reduces the number of computations by several orders of magnitude in comparison with a full static graph computation. In practice we showed that using the dynamic graph algorithm can be up to $8000X$ faster

Table 21: New memory bounds for multiple betweenness centrality algorithms.

Algorithm	Graph	Previous	New [74]
Exact [30]	Static	$O(V+E)$	$O(V)$
Approximate [17]	Static	$O(V+E)$	$O(V)$
Parallel fine-grain [15, 104]	Static	$O(V+E)$	$O(V)$
Parallel coarse-grain [74]	Static	$O(P\cdot(V+E))$	$O(P\cdot V)$
Exact [77]	Dynamic	$O(V\cdot(V+E))$	$O(V^2)$
Approximate	Dynamic	$O(K\cdot(V+E))$	$O(K\cdot V)$
Parallel fine-grain approximate	Dynamic	$O(K\cdot(V+E))$	$O(K\cdot V)$
Parallel coarse-grain approximate	Dynamic	$O(K\cdot(V+E))$	$O(K\cdot V)$

than a full static graph computation. As the new dynamic algorithm requires additional storage to maintain state between updates, we revisited the data structures used for computing betweenness centrality. We showed a way to reduce the storage complexity for multiple betweenness centrality algorithms, Table 21, by doing additional edge traversals rather than maintaining the parent-list which requires an $O(E)$ storage. In practice, many of these new algorithms have better performance as the parent-list dominated the cache which causes cache thrashing - kicked out useful necessary information. An immediate outcome of this improvement, which we called the neighbor-traversal approach, is parallel computation of betweenness centrality at a coarse-grain using a large number of threads. Previously this was not possible. We showed the first large-scale implementation of this for both static and dynamic graphs. Lastly, we show the first parallel dynamic graph algorithm and implementation for computing betweenness centrality. Our algorithm can do both exact and approximate computations.

We then proceeded to create a parallel algorithm for finding connected components in dynamic graphs using insights we gain from the dynamic betweenness centrality algorithm. Our algorithm continuously monitors the numerous connected components in the graph and can deal with up to $1.2M$ updates per second. This new algorithm takes advantage of graph properties such as the “small-world” property and the shrinking diameter property. We showed that by maintaining a list of pointers to neighbors that are closer to the root of the connected component, it is possible in many cases to answer in $O(1)$ time if the connected components have changed due to an edge deletion. We show that edge insertions are considerably easier to deal with.

Merge Path is a visual approach for doing a parallel merge of two sorted arrays. This algorithm is highly scalable as the work is evenly distributed to the many cores in a system where the cost of partitioning is low. We also showed the first cache-aware merging algorithm. We show that this algorithm has an optimal number of cache misses and show that if certain conditions are met that cache thrashing is avoided. This cache-aware algorithm, allowed us to develop a merge algorithm for the GPU - the first of its kind. Our GPU algorithm was up-to $52X$ faster than a sequential implementation. GPU Merge Path was adopted by NVIDIA for its CUDA Thrust software library. The Merge Path concept can also be used for list intersection which is building block for clustering coefficients.

For efficient computation of clustering coefficients we show two optimizations. Our first optimization reduces the time complexity for computing clustering coefficients by finding a vertex cover. This approach uses counting corrections to avoid redundant counting of the same triangle. In practice this approach can give a $1.1X - 1.3X$ speedup for triangle counting (the speedup includes the overhead introduced by the vertex cover). We extend this approach for finding larger circuits. We showed that for counting squares (circuits of length 4) the speedup is typically $1.5X - 2X$. We used a simple linear-time algorithm for finding a vertex cover and showed that the overhead of finding the vertex cover is not great. Using the insight that linear-time work is relatively inexpensive in comparison with the remaining work required by the algorithm, we developed two parallel and scalable approaches for computing clustering coefficients. These load-balancing approaches are also linear in time. To implement the load-balancing, we developed a simple workload estimating model for clustering coefficients. We show that our estimation does a good job in predicting the parallel performance. We then show that our two approaches have better scalability than previous approaches, thus we achieve significant speedups and high system utilization.

Lastly, we show a scalable algorithm for efficient computation of the estimated covariance matrix based off the Covariance Method formulation. The covariance method is a building for many signal and image processing algorithms. We show that there is significant overlap between different computations of the algorithm. We divide the algorithm into four different computational phases and show that by using a specific ordering of these phases, we can reduce the total number of computations by several orders of magnitude which in turn gives a speedup which is order of magnitude faster than before. We then show that our new approach is highly scalable and gets near linear speedups on a 40 core system where previous algorithms did not scale because either their storage requirements or the communication requirements. Our algorithm requires the same amount of memory as a sequential implementation and has very little synchronization (a single barrier at the end).

10.3 The "Melting-Pot"

Fig. 70 depicts the relationship between the three different analytics: connected components, betweenness centrality, and clustering coefficients. While these analytics focus on answering different questions, they share common traits.

Having the set of connected components is useful for both betweenness centrality and clustering coefficients as the size of the connected component can help normalize the significancence of players. For example, being a key player with a high betweenness centrality value in a connected component with four players is far easier than being a key player in a connected component with millions of players.

All these analytics can potentially get better performance by utilizing prefix summation. We showed this clearly for clustering coefficients with two different load-balancing approaches. We also showed that both phases of betweenness centrality are inherently imbalanced. While previous work focused on balancing the first phase,

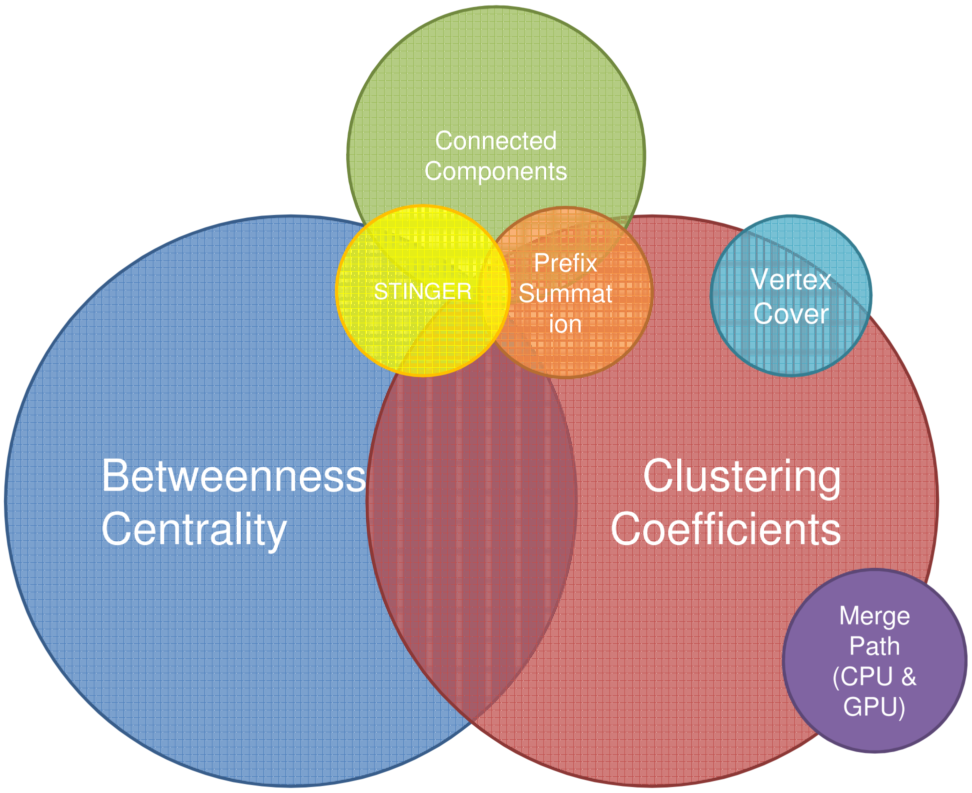


Figure 70: Analytic interaction diagram.

the breadth first search, the dependency accumulation can be just as imbalanced. Further, as our connected component algorithm is based on an breadth-first search principle, it is likely to benefit from better load-balancing with parallel prefix summation.

STINGER was the right data structure for our streaming analytics and simplified implementation. While betweenness centrality is a computationally demanding analytic, if we had to recreate the CSR representation for every update (insertion or deletion), we would spend far too much time in the "graph update" phase instead of the "analytic update" phase. For our dynamic connected component algorithm, STINGER allowed us to do as many as $1.2M$ updates per second to the connected component while also updating the graph representation.

We showed how vertex covers can reduced the time complexity for finding the triangles need for clustering coefficients. We extended the vertex cover approach to support counting larger circuits and showed how to use it for circuits of length four ("squares"). In all likelihood, vertex covers can also be applied to additional analytics.

Merge Path and GPU Merge Path are fast algorithms for parallel merging of two sorting arrays. GPU Merge Path was adopted by NVIDA as part of the CUDA Thrust library. Given the similarity of merging and sort list intersection, the Merge Path concepts can be extended to support fast adjacency list intersections that are needed by for clustering coefficients especially for intersections of large lists.

10.4 On the Road to Exascale and Beyond - Future Research Directions

In recent years exascale systems have become a goal for the high performance computing community. The need for exascale is apparent for application specialists of multiple domains, including those in the field of social network analysis. Just the mere sizes of social networks such Twitter and Facebook, that have over one billion users and with hundreds of billions of relationships, present numerous challenges for designing algorithms and analytics. Further, these sizes require that the algorithm designer consider the use of distributed systems that typically increase the computational power and storage available. The availability of such systems will require designing streaming and parallel algorithms for these massively multithreaded systems.

In this work we showed that there are still techniques for increasing system utilization for multiple analytics for shared-memory systems. The switch from shared-memory systems to distributed systems will present numerous challenges that we did not face on shared-memory systems. The switch will require developing new approaches for maximizing system utilization. Some of these optimizations will include data partitioning that will help reduced communication cost. To avoid the cost of communication or reduce its relative cost, it be necessary to compute multiple analytics concurrently on a specific compute node. Another possibility will be to move the “work” to the data rather than the data to work. This means that we will try to reduce moving massive amounts of data between different compute nodes and instead we will move the application and part of the data structure to wherever the required data is situated.

Social network analysis and other domains can benefit from existing software packages that have been implemented for distributed systems. This may require transforming the problem statement of interests, for example transforming a graph representation to matrix representation and using linear algebra operators. While this is not always desirable, the benefit is the increase in the available compute power.

Finally, I believe that the road(s) to exascale will include and require algorithm and architecture co-design. We can no longer continue using hardware that was designed initially in the 70’s or 80’s when the types of problems that were interesting included several thousands of elements whereas today our problem sizes include millions to billions of data elements. Further, the recent trend system design has focused on designing high performance for matrix multiplication problems - this is not always helpful for data scientists.

Our role as computational and applications specialists will be to help “direct” computer architects to better design the types of system that we need for irregular algorithms. Our insights on the type of functionality required by the application are crucial in understanding which hardware units are necessary and which are not. Further, as the cost of commodity hardware goes down and the number of unique platforms (accelerators included) become available, we will need to do better matching between application and computing platform.

REFERENCES

[1] *GraphCT: A Graph Characterization Toolkit.*

[2] *SNAP: Small-world Network Analysis and Partitioning.*

[3] “EE Times - Analysis: ŚHypercoreŠ Touts 256 CPUs Per Chip,” 2007.

[4] “Hpcs scalable synthetic compact applications 2 graph analysis,” Sep 2007.

[5] “Plurality - HyperCore Software DeveloperŠs Handbook,” 2009.

[6] “10th dimacs implementation challenge - graph partitioning and graph clustering,” 2013.

[7] “Facebook,” 2013.

[8] “Intel Math Kernel Library,” 2013.

[9] “Twitter,” 2013.

[10] Abdo, A. H. and de Moura, A. P. S., “Clustering as a Measure of the Local Topology of Networks,” *arXiv physics/0605235*, 2006.

[11] Aggarwal, A. and Vitter, Jeffrey, S., “The Input/Output Complexity of Sorting and Related Problems,” *Commun. ACM*, vol. 31, pp. 1116–1127, September 1988.

[12] Akl, S. G. and Santoro, N., “Optimal Parallel Merging and Sorting Without Memory Conflicts,” *Computers, IEEE Transactions on*, vol. C-36, pp. 1367 – 1369, nov. 1987.

[13] Albert, R., Jeong, H., and Barabási, A., "Internet: Diameter of the world-wide web," *Nature*, vol. 401, pp. 130–131, Sep 09 1999.

[14] Amdahl, G. M., "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, pp. 483–485, ACM, 1967.

[15] Bader, D. and Madduri, K., "Parallel algorithms for evaluating centrality indices in real-world networks," in *Parallel Processing, 2006. ICPP 2006. International Conference on*, pp. 539 –550, aug. 2006.

[16] Bader, D. and Madduri, K., "Snap, small-world network analysis and partitioning: an open-source parallel graph framework for the exploration of large-scale networks," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pp. 1–12, IEEE, 2008.

[17] Bader, D., Kintali, S., Madduri, K., and Mihail, M., "Approximating betweenness centrality," in *Algorithms and Models for the Web-Graph*, vol. 4863 of *Lecture Notes in Computer Science*, pp. 124–137, Springer Berlin / Heidelberg, 2007.

[18] Bader, D. A., Berry, J., Amos-Binks, A., Chavarría-Miranda, D., Hastings, C., Madduri, K., and Poulos, S. C., "STINGER: Spatio-Temporal Interaction Networks and Graphs (STING) Extensible Representation," tech. rep., Georgia Institute of Technology, 2009.

[19] Bader, D. A., Meyerhenke, H., Sanders, P., and Wagner, D., *10th DIMACS Implementation Challenge on Graph Partitioning and Graph Clustering*, vol. 588. American Mathematical Society, 2013.

[20] Bar-Yehuda, R. and Even, S., "A Linear-Time Approximation Algorithm for the Weighted Vertex Cover Problem," *Journal of Algorithms*, vol. 2, no. 2, pp. 198–203, 1981.

[21] Bar-Yossef, Z., Kumar, R., and Sivakumar, D., "Reductions in streaming algorithms, with an application to counting triangles in graphs," in *ACM-SIAM Symposium on Discrete algorithms*, SODA '02, (Philadelphia, PA, USA), pp. 623–632, Society for Industrial and Applied Mathematics, 2002.

[22] Barabási, A.-L. and Albert, R., "Emergence of Scaling in Random Networks," *Science*, vol. 286, no. 5439, pp. 509–512, 1999.

[23] Barabási, A.-L. and Oltvai, Z. N., "Network Biology: Understanding the Cell's Functional Organization," *Nature Reviews Genetics*, vol. 5, no. 2, pp. 101–113, 2004.

[24] Barrett, B. W., Berry, J. W., Murphy, R. C., and Wheeler, K. B., "Implementing a portable multi-threaded graph library: The MTGL on Qthreads," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pp. 1–8, IEEE, 2009.

[25] Batcher, K., "Sorting Networks and Their Applications," in *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pp. 307–314, ACM, 1968.

[26] Beamer, S., Asanovic, K., and Patterson, D., "Direction-optimizing breadth-first search," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pp. 1–10, IEEE, 2012.

[27] Becchetti, L., Boldi, P., Castillo, C., and Gionis, A., "Efficient Semi-streaming Algorithms for Local Triangle Counting in Massive Graphs," in *Proceedings of the 14th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining*, pp. 16–24, ACM, 2008.

[28] Blelloch, G. E., "Prefix sums and their applications," 1990.

[29] Bokhari, S. H., Catalyurek, U. V., and Gurcan, U. V., "Massively Multithreaded Maxflow for Image Segmentation on the Cray XMT2," 2013.

[30] Brandes, U., "A faster algorithm for betweenness centrality," *Journal of Mathematical Sociology*, vol. 25, no. 2, pp. 163–177, 2001.

[31] Broder, A., Kumar, R., Maghoul, F., Raghavan, P., Rajagopalan, S., Stata, R., Tomkins, A., and Wiener, J., "Graph Structure in the Web," *Computer Networks*, vol. 33, no. 1, pp. 309–320, 2000.

[32] Broder, A., Kumar, R., Maghoul, F., Raghavan, P., Rajagopalan, S., Stata, R., Tomkins, A., and Wiener, J., "Graph structure in the web," *Computer Networks*, vol. 33, pp. 309 – 320, 2000.

[33] Brüschweiler, R. and Zhang, F., "Covariance Nuclear Magnetic Resonance Spectroscopy," *The Journal of chemical physics*, vol. 120, p. 5253, 2004.

[34] Buluç, A. and Gilbert, J. R., "The Combinatorial BLAS: design, implementation, and applications," *International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 496–509, 2011.

[35] Buriol, L. S., Frahling, G., Leonardi, S., Marchetti-Spaccamela, A., and Sohler, C., "Counting Triangles in Data Streams," in *Proceedings of the 25th ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pp. 253–262, ACM, 2006.

[36] Bye, R., Schmidt, S., Luther, K., and Albayrak, S., "Application-level simulation for network security," in *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, (ICST, Brussels, Belgium, Belgium), pp. 33:1–33:10, 2008.

[37] Capon, J., "High-Resolution Frequency-Wavenumber Spectrum Analysis," *Proceedings of the IEEE*, vol. 57, no. 8, pp. 1408–1418, 1969.

[38] Cayre, F., Fontaine, C., and Furon, T., "Watermarking Security: Theory and Practice," *Signal Processing, IEEE Transactions on*, vol. 53, no. 10, pp. 3976–3987, 2005.

[39] Chakrabarti, D., Zhany, Y., and Faloutsos, C., "R-MAT: A recursive model for graph mining," in *SIAM Proceedings Series*, pp. 442–446, 2004.

[40] Chen, S., Qin, J., Xie, Y., Zhao, J., and Heng, P., "An efficient sorting algorithm with cuda," *Journal of the Chinese Institute of Engineers*, vol. 32, no. 7, pp. 915–921, 2009.

[41] Chowdhury, R., Silvestri, F., Blakeley, B., and Ramachandran, V., "Oblivious Algorithms for Multicores and Network of Processors," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1–12, IEEE, 2010.

[42] Cole, R. and Ramachandran, V., "Resource Oblivious Sorting on Multicores," *Automata, Languages and Programming*, pp. 226–237, 2010.

[43] Conway, P., Kalyanasundharam, N., Donley, G., Lepak, K., and Hughes, B., "Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor," *Micro, IEEE*, vol. 30, no. 2, pp. 16–29, 2010.

[44] Coppersmith, D. and Winograd, S., “Matrix Multiplication via Arithmetic Progressions,” *Journal of Symbolic Computation*, vol. 9, no. 3, pp. 251–280, 1990.

[45] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C., *Introduction to Algorithms.* New York: The MIT Press, 2001.

[46] Dagum, L. and Menon, R., “OpenMP: an industry standard API for shared-memory programming,” *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[47] David, L., Galperin, A., Green, O., and Birk, Y., “Efficient Parallel Computation of the Estimated Covariance Matrix,” in *IEEE 26th Convention of Electrical and Electronics Engineers in Israel*, 2010.

[48] DeGraaf, S., “SAR Imaging Via Modern 2d Spectral Estimation Methods,” *Image Processing, IEEE Transactions on*, vol. 7, no. 5, pp. 729–761, 1998.

[49] Deo, N., Jain, A., and Medidi, M., “An optimal parallel algorithm for merging using multiselection,” *Information Processing Letters*, 1994.

[50] Deo, N. and Sarkar, D., “Parallel Algorithms for Merging and Sorting,” *Information Sciences*, vol. 56, pp. 151 – 161, 1991.

[51] Dillencourt, M. B., Samet, H., and Tamminen, M., “A general approach to connected-component labeling for arbitrary image representations,” *Journal of the ACM (JACM)*, vol. 39, no. 2, pp. 253–280, 1992.

[52] Douglas, L., “The Importance of ŚBig DataŠ: A Definition,” *Gartner (June 2012)*, 2012.

[53] Ediger, D., Jiang, K., Riedy, J., and Bader, D., “Massive streaming data analytics: A case study with clustering coefficients,” in *Parallel & Distributed*

Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on, pp. 1–8, IEEE, 2010.

[54] Ediger, D., McColl, R., Riedy, J., and Bader, D., "Stinger: High performance data structure for streaming graphs," in *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*, pp. 1–5, IEEE, 2012.

[55] Ediger, D., McColl, R., Riedy, J., and Bader, D., "Stinger: High performance data structure for streaming graphs," in *Proc. High Performace Embedded Computing Workshop (HPEC 2012)*, (Waltham, MA), Sept. 2012.

[56] Ediger, D., Riedy, J., Bader, D., and Meyerhenke, H., "Tracking structure of streaming social networks," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pp. 1691 –1699, may 2011.

[57] Ediger, D., Jiang, K., Riedy, J., and Bader, D., "Graphct: Multithreaded algorithms for massive graph analysis," 2012.

[58] Edmonds, N., Hoefler, T., and Lumsdaine, A., "A space-efficient parallel algorithm for computing betweenness centrality in distributed memory," in *International Conference on High Performance Computing (HiPC), 2010*, pp. 1 –10, Dec. 2010.

[59] Edmonds, N., Hoefler, T., and Lumsdaine, A., "A Space-efficient Parallel Algorithm for Computing Betweenness Centrality in Distributed Memory," in *High Performance Computing (HiPC), 2010 International Conference on*, pp. 1–10, IEEE, 2010.

[60] Eppstein, D., Galil, Z., Italiano, G. F., and Nissenzweig, A., "Sparsification-a technique for speeding up dynamic graph algorithms," *Journal of the ACM (JACM)*, vol. 44, no. 5, pp. 669–696, 1997.

[61] Erdös, P. and Rényi, A., "On random graphs I," *Publicationes Mathematicae*, pp. 290–297, June 1959.

[62] Erdös, P. and Rényi, A., "The evolution of random graphs," *Magyar Tud. Akad. Mat.*, pp. 17–61, 1960.

[63] Faloutsos, M., Faloutsos, P., and Faloutsos, C., "On Power-Law Relationships of The Internet Topology," in *ACM SIGCOMM Computer Communication Review*, pp. 251–262, ACM, 1999.

[64] Fante, R., Barile, E., and Guella, T., "Clutter Covariance Smoothing by Subaperture Averaging," *Aerospace and Electronic Systems, IEEE Transactions on*, vol. 30, no. 3, pp. 941–945, 1994.

[65] Ferragina, P., "Static and dynamic parallel computation of connected components," *Information processing letters*, vol. 50, no. 2, pp. 63–68, 1994.

[66] Floyd, R. W., "Algorithm 97: Shortest path," *Commun. ACM*, vol. 5, pp. 345–345, June 1962.

[67] Freeman, L. C., "A set of measures of centrality based on betweenness," *Sociometry*, vol. 40, no. 1, pp. pp. 35–41, 1977.

[68] Frigo, M., Leiserson, C., Prokop, H., and Ramachandran, S., "Cache-Oblivious Algorithms," in *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pp. 285 –297, 1999.

[69] Garey, M. R. and Johnson, D. S., *Computers and Intractability*, vol. 174. freeman New York, 1979.

[70] Geisberger, R., Sanders, P., and Schultes, D., "Better Approximation of Betweenness Centrality," in *ALENEX*, pp. 90–100, 2008.

[71] Girvan, M. and Newman, M. E. J., "Community structure in social and biological networks," *Proceedings of the National Academy of Sciences*, vol. 99, no. 12, pp. 7821–7826, 2002.

[72] Govindaraju, N. K., Raghuvanshi, N., Henson, M., Tuft, D., and Manocha, D., "A cache-efficient sorting algorithm for database and data mining computations using graphics processors," tech. rep., 2005.

[73] Green, O., McColl, R., and Bader, D., "Gpu merge path: a gpu merging algorithm," in *Proceedings of the 26th ACM International Conference on Supercomputing*, pp. 331–340, ACM, 2012.

[74] Green, O. and Bader, D. A., "Faster Betweenness Centrality Based on Data Structure Experimentation," in *International Conference on Computational Science (ICCS)*, Elsevier, 2013.

[75] Green, O. and Bader, D. A., "Faster Clustering Coefficients Using Vertex Covers," in *5th ASE/IEEE International Conference on Social Computing*, SocialCom, 2013.

[76] Green, O., David, L., Galperin, A., and Birk, Y., "Efficient Parallel Computation of the Estimated Covariance Matrix," *arXiv preprint arXiv:1303.2285*, 2013.

[77] Green, O., McColl, R., and Bader, D. A., "A Fast Algorithm For Streaming Betweenness Centrality," in *Proceedings of the 4th ASE/IEEE International Conference on Social Computing*, SocialCom '12, 2012.

[78] Harris, M., Sengupta, S., and Owens, J. D., "Parallel prefix sum (scan) with CUDA," *GPU gems*, vol. 3, no. 39, pp. 851–876, 2007.

[79] Hennessy, J. L. and Patterson, D. A., *Computer Architecture : A Quantitative Approach.* Boston: Morgan Kaufmann, 2007.

[80] Henzinger, M. R., King, V., and Warnow, T., "Constructing a tree from homeomorphic subtrees, with applications to computational evolutionary biology," *Algorithmica*, vol. 24, pp. 1–13, 1999.

[81] Henzinger, M. R. and King, V., "Randomized fully dynamic graph algorithms with polylogarithmic time per operation," *J. ACM*, vol. 46, pp. 502–516, July 1999.

[82] Hirschberg, D. S., Chandra, A. K., and Sarwate, D. V., "Computing connected components on parallel computers," *Commun. ACM*, vol. 22, pp. 461–464, Aug. 1979.

[83] Hoberock, J. and Bell, N., "Thrust: A parallel template library," 2010. Version 1.3.0.

[84] Hochbaum, D. S., "Approximation Algorithms for the Set Covering and Vertex Cover Problems," *SIAM Journal on Computing*, vol. 11, no. 3, pp. 555–556, 1982.

[85] Hopcroft, J. and Tarjan, R., "Algorithm 447: efficient algorithms for graph manipulation," *Commun. ACM*, vol. 16, pp. 372–378, June 1973.

[86] Israeli, A. and Shiloach, Y., "An Improved Parallel Algorithm for Maximal Matching," *Information Processing Letters*, vol. 22, no. 2, pp. 57–60, 1986.

[87] Jakobsson, A., Marple Jr, S., and Stoica, P., "Computationally Efficient Two-Dimensional Capon Spectrum Aanalysis," *Signal Processing, IEEE Transactions on*, vol. 48, no. 9, pp. 2651–2661, 2000.

[88] Jakobsson, A., Alty, S. R., and Benesty, J., "Estimating and Time-updating the 2d Coherence Spectrum," *Signal Processing, IEEE Transactions on*, vol. 55, no. 5, pp. 2350–2354, 2007.

[89] Kas, M., Wachs, M., Carley, K. M., and Carley, L. R., "Incremental algorithm for updating betweenness centrality in dynamically growing networks," in *IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, 2013.

[90] Kay, S., *Modern Spectral Estimation.* Pearson Education, 1988.

[91] Khuller, S., Vishkin, U., and Young, N., "A Primal-dual Parallel Approximation Technique Applied to Weighted Set and Vertex Covers," *J. Algorithms*, vol. 17, no. 2, pp. 280–289, 1994.

[92] Kleen, A., "A NUMA API for Linux," *Novel Inc*, 2005.

[93] Konecny, P., "Introducing the cray xmt," (Seattle, WA, USA), may 2007.

[94] Kruskal, C. P., Rudolph, L., and Snir, M., "Efficient parallel algorithms for graph problems," *Algorithmica*, vol. 5, no. 1-4, pp. 43–64, 1990.

[95] Kumar, V., Ramesh, K., and Rao, V. N., "Parallel Best-First Search of State-space Graphs: A Summary of Results," in *Proceedings of the 1988 National Conference on Artificial Intelligence. Morgan Kaufmann*, Citeseer, 1988.

[96] Lam, M., Rothberg, E., and Wolf, M., "The Cache Performance and Optimizations of Blocked Algorithms," in *ACM SIGARCH Computer Architecture News*, vol. 19, pp. 63–74, ACM, 1991.

[97] Lee, M.-J., Lee, J., Park, J. Y., Choi, R. H., and Chung, C.-W., "QUBE: a Quick algorithm for Updating BEtweenness centrality," in *Proceedings of the 21st international conference on World Wide Web*, pp. 351–360, ACM, 2012.

[98] Leist, A., Hawick, K., Playne, D., and Albany, N. S., "GPGPU and Multi-Core Architectures for Computing Clustering Coefficients of Irregular Graphs," in *Proc. Int'l Conf. on Scientific Computing (CSC'11)*, 2011.

[99] Leskovec, J., Kleinberg, J., and Faloutsos, C., "Graphs Over Time: Densification Laws, Shrinking Diameters and Possible Explanations," in *Proceedings of the 11th ACM SIGKDD Int'l Conf. on Knowledge Discovery in Data Mining*, pp. 177–187, ACM, 2005.

[100] Leskovec, J., Kleinberg, J., and Faloutsos, C., "Graph evolution: Densification and shrinking diameters," *ACM Trans. Knowl. Discov. Data*, vol. 1, no. 1, 2007.

[101] Li, J. and Stoica, P., "An Adaptive Filtering Approach To Spectral Estimation and SAR Imaging," *Signal Processing, IEEE Transactions on*, vol. 44, no. 6, pp. 1469–1484, 1996.

[102] Liu, Z., Li, H., and Li, J., "Efficient Implementation of Capon and APES for Spectral Estimation," *Aerospace and Electronic Systems, IEEE Transactions on*, vol. 34, no. 4, pp. 1314–1319, 1998.

[103] López-Dekker, P. and Mallorquí, J., "Capon-and APES-based SAR processing: Performance and Practical Considerations," *Geoscience and Remote Sensing, IEEE Transactions on*, vol. 48, no. 5, pp. 2388–2402, 2010.

[104] Madduri, K., Ediger, D., Jiang, K., Bader, D., and Chavarria-Miranda, D., "A Faster Parallel Algorithm and Efficient Multithreaded Implementations for Evaluating Betweenness Centrality on Massive Datasets," in *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2009.

[105] Makhoul, J., "Linear Prediction: A Tutorial Review," *Proceedings of the IEEE*, vol. 63, no. 4, pp. 561–580, 1975.

[106] Marple Jr, S. and Carey, W., "Digital Spectral Analysis with Applications," *The Journal of the Acoustical Society of America*, vol. 86, p. 2043, 1989.

[107] McColl, R., Green, O., and Bader, D., "Parallel streaming connected components using "parent-neighbor" subgraphs," in *IEEE International Conference on High Performance Computing*, 2013.

[108] Merrill, D., Garland, M., and Grimshaw, A., "Scalable gpu graph traversal," in *ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, (New York, NY, USA), pp. 117–128, ACM, 2012.

[109] Milgram, S., "The Small World Problem," *Psychology Today*, vol. 2, no. 1, pp. 60–67, 1967.

[110] Murphy, R. C., Wheeler, K. B., Barrett, B. W., and Ang, J. A., "Introducing the Graph 500," *Cray UserŠs Group (CUG)*, 2010.

[111] Newman, M. and Girvan, M., "Finding and evaluating community structure in networks," *Physical review E*, vol. 69, no. 2, p. 026113, 2004.

[112] Nvidia, C., "Programming Guide, 2010," *NVIDIA Corporation.*

[113] NVIDIA Corporation, "Nvidia cuda programming guide," 2011.

[114] Odeh, S., Green, O., Mwassi, Z., Shmueli, O., and Birk, Y., "Merge path - cache-efficient parallel merge and sort," tech. rep., CCIT Report No. 802, EE Pub. No. 1759, Electrical Engr. Dept., Technion, Israel, Jan. 2012.

[115] Odeh, S., Green, O., Mwassi, Z., Shmueli, O., and Birk, Y., "Merge path - parallel merging made simple," in *Parallel and Distributed Processing Symposium, International*, may 2012.

[116] Prountzos, D. and Pingali, K., "Betweenness centrality: Algorithms and implementations," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 35–46, ACM, 2013.

[117] Puzis, R., Zilberman, P., Elovici, Y., Dolev, S., and Brandes, U., "Heuristics for speeding up betweenness centrality computation," in *Proceedings of the 4th ASE/IEEE International Conference on Social Computing*, pp. 302–311, 2012.

[118] Roditty, L. and Zwick, U., "A fully dynamic reachability algorithm for directed graphs with an almost linear update time," in *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, STOC '04, (New York, NY, USA), pp. 184–191, ACM, 2004.

[119] Rubinov, M. and Sporns, O., "Complex network measures of brain connectivity: Uses and interpretations," *NeuroImage*, vol. 52, no. 3, pp. 1059 – 1069, 2010. Computational Models of the Brain.

[120] Sariyüce, A. E., Kaya, K., Saule, E., and Çatalyürek, Ü. V., "Betweenness Centrality on GPUs and Heterogeneous Architectures," in *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, pp. 76–85, ACM, 2013.

[121] Sariyuce, A. E., Kaya, K., Saule, E., and Catalyurek, U. V., "Incremental Algorithms for Network Management and Analysis based on Closeness Centrality," *arXiv preprint arXiv:1303.0422*, 2013.

[122] Satish, N., Harris, M., and Garland, M., "Designing efficient sorting algorithms for manycore gpus," *Parallel and Distributed Processing Symposium, International*, vol. 0, pp. 1–10, 2009.

[123] Schank, T. and Wagner, D., "Finding, Counting and Listing All Triangles in Large Graphs, an Experimental Study," in *Experimental and Efficient Algorithms*, pp. 606–609, Springer, 2005.

[124] Scherl, H., Keck, B., Kowarschik, M., and Hornegger, J., "Fast GPU-based CT Reconstruction Using the Common Unified Device Architecture (CUDA)," in *Nuclear Science Symposium Conference Record, 2007. NSS'07. IEEE*, IEEE, 2007.

[125] Shavitt, Y. and Shir, E., "DIMES: Let the Internet Measure Itself," *ACM SIGCOMM Computer Communication Review*, vol. 35, no. 5, pp. 71–74, 2005.

[126] Shiloach, Y. and Vishkin, U., "Finding the maximum, merging, and sorting in a parallel computation model," *Journal of Algorithms*, vol. 2, pp. 88 – 102, 1981.

[127] Shiloach, Y. and Even, S., "An on-line edge-deletion problem," *J. ACM*, vol. 28, pp. 1–4, January 1981.

[128] Shiloach, Y. and Vishkin, U., "An O(logn) Parallel Connectivity Algorithm," *Journal of Algorithms*, vol. 3, no. 1, pp. 57 – 67, 1982.

[129] Shiloach, Y. and Vishkin, U., "An o(logn) parallel connectivity algorithm," *Journal of Algorithms*, vol. 3, no. 1, pp. 57 – 67, 1982.

[130] Shun, J. and Blelloch, G. E., "Ligra: a lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '13, (New York, NY, USA), pp. 135–146, ACM, 2013.

[131] Sintorn, E. and Assarsson, U., "Fast parallel gpu-sorting using a hybrid algorithm," *Journal of Parallel and Distributed Computing*, vol. 68, no. 10,

pp. 1381 – 1388, 2008. General-Purpose Processing using Graphics Processing Units.

[132] Stenstrom, P., "A Survey of Cache Coherence Schemes for Multiprocessors," *Computer*, vol. 23, no. 6, pp. 12–24, 1990.

[133] Strogatz, S. H., "Exploring Complex Networks," *Nature*, vol. 410, no. 6825, pp. 268–276, 2001.

[134] Tan, G., Sreedhar, V., and Gao, G., "Analysis and performance results of computing betweenness centrality on IBM Cyclops64," *The Journal of Supercomputing*, vol. 56, pp. 1–24, 2011.

[135] Tan, G., Tu, D., and Sun, N., "A parallel algorithm for computing betweenness centrality," in *Parallel Processing, 2009. ICPP '09. International Conference on*, pp. 340 –347, sept. 2009.

[136] Ugander, J., Karrer, B., Backstrom, L., and Marlow, C., "The Anatomy of the Facebook Social Graph," *arXiv preprint arXiv:1111.4503*, 2011.

[137] Warshall, S., "A theorem on Boolean matrices," *J. ACM*, vol. 9, pp. 11–12, Jan. 1962.

[138] Watts, D. J. and Strogatz, S. H., "Collective Dynamics of "Small-World" Networks," *Nature*, vol. 393, no. 6684, pp. 440–442, 1998.

[139] Watts, Duncan J.Strogatz, S. H., "Collective dynamics of 'small-world' networks.," *Nature*, vol. 393, 1998.

[140] Yadin, E., Olmar, D., Oron, O., and Nathansohn, R., "SAR Imaging Using a Modern 2d Spectral Estimation Method," in *Radar Conference, 2008. RADAR'08. IEEE*, pp. 1–6, IEEE, 2008.

VITA

Oded is a PhD student at Georgia Institute of Technology, advised by Prod. David A. Bader. Oded started his PhD in January 2011. Oded received a M.Sc in Electirical Engineering in 2010 and a B.Sc in Computer Engineering in 2008, both from the Technion (Israel Institute of Technology). Oded's MSc advisor was Prof. Yitzhak Birk from the Technion.

Oded received the Sam Nunn Security Fellowship (2012). He received Best Talk award at IBM Student Workshop for Frontiers of Cloud Computing (2013). As a graduate student, Oded has advised both undergraduate research projects and junior graduate students. Several of these projects have been awarded best project award and have been followed by publications.

Oded's services includes review papers for over ten different conferences and journals, partaking in student governments, founding a graduate student seminar (Hot Topics in Computational Science and Engineering), and being on the Technion's Squash varsity team.

Oded has approximately 15 years in software development and has partaken in several open-source projects.

Oded is a member of ACM, IEEE, and SIAM.

High Performance Computing for Irregular Algorithms and Applications with an Emphasis on Big Data Analytics

Oded Green

259 Pages

Directed by Professor David A. Bader

High Performance Computing for Irregular Algorithms and Applications with an Emphasis on Big Data Analytics

Oded Green

259 Pages

Directed by Professor David A. Bader

Irregular algorithms such as graph algorithms, sorting, and sparse matrix multiplication, present numerous programming challenges, including scalability, load balancing, and efficient memory utilization. In this age of Big Data we face additional challenges since the data is often streaming at a high velocity and we wish to make near real-time decisions for real-world events. For instance, we may wish to track Twitter for the pandemic spread of a virus. Analyzing such data sets requires combing algorithmic optimizations and utilization of massively multithreaded architectures, accelerator such as GPUs, and distributed systems. My research focuses upon designing new analytics and algorithms for the continuous monitoring of dynamic social networks.

Achieving high performance computing for irregular algorithms such as Social Network Analysis (SNA) is challenging as the instruction flow is highly data dependent and requires domain expertise. The rapid changes in the underlying network necessitates understanding real-world graph properties such as the small world property, shrinking network diameter, power law distribution of edges, and the rate at which updates occur. These properties, with respect to a given analytic, can help design load-balancing techniques, avoid wasteful (redundant) computations, and create streaming algorithms.

In the course of my research I have considered several parallel programming paradigms for a wide range systems of multithreaded platforms: x86, NVIDIA's

CUDA, Cray XMT2, SSE-SIMD, and Plurality's HyperCore. These unique programming models require examination of the parallel programming at multiple levels: algorithmic design, cache efficiency, fine-grain parallelism, memory bandwidths, data management, load balancing, scheduling, control flow models and more. This thesis deals with these issues and more.