BOUNDED ASYNCHRONY AND NESTED PARALLELISM FOR SCALABLE GRAPH PROCESSING

A Dissertation

by

ADAM K FIDEL

Submitted to the Graduate and Professional School of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Nancy Amato
Co-Chair of Committee	Lawrence Rauchwerger
Committee Members,	Jennifer Welch
	Nick Duffield
Head of Department,	Scott Schaefer

December 2021

Major Subject: Computer Science

ABSTRACT

Processing large-scale graphs has increasingly become a critical component in a variety of fields, from scientific computing to social analytics. The size of graphs of interest are becoming explosively large, preventing them from fitting into the memory of a single-processor system and highlighting the need for fast and efficient methods to process such graphs. Because of this, there exists a clear need for distributed data structures and parallel algorithms to facilitate the processing of these large graphs.

Graph traversals – wherein a computation proceeds from one vertex to another along the edges of a graph – are an important type of algorithm, as they form the backbone of several other important graph algorithms (e.g., shortest paths, centrality metrics and connected components). Improving the performance of traversals therefore in turn benefits all algorithms dependent on them. Despite receiving a great deal of attention from many researchers for several decades [1, 2, 3, 4, 5], traversal-based computations remain notoriously difficult to parallelize effectively.

In this proposal, we will discuss two broad techniques for improving the performance of graph traversals and general parallel graph algorithms:

1. **Asynchrony.** Increasing the asynchrony of the algorithm allows one to avoid global synchronization, while still being mindful of the negatives of unbounded asynchrony including wasted work.

2. **Nested parallelism.** Allowing to express graph algorithms in a naturally nested parallel manner enables us to fully exploit all of the available parallelism inherent in graph algorithms.

DEDICATION

To all that love me.

ACKNOWLEDGMENTS

I would like to thank my advisors Prof. Nancy M. Amato and Prof. Lawrence Rauchwerger for their support, guidance and mentorship throughout my PhD studies. From my undergraduate research internship through the end of my graduate studies, they have helped me navigate and overcome the immense and numerous challenges that have appeared over the many years of research. Were it not for their support, I truly believe that I would not have been able to complete this journey. I would like to express my deepest gratitude for their support, for without it, this dissertation would not exist.

I would like to thank Prof. Jennifer L. Welch for the discussions and guidance she gave me with respect to the more formal sections of this document, as well as her kind presence that made me feel that I could make it through my studies. I am also thankful for Prof. Nick Duffield for the interesting and thought-provoking conversations we had with respect to potential collaborative research projects.

Everyone in the Parasol Lab is deserving of thanks for their support, collaboration, camaraderie and friendship. I would especially like to thank Harshvardhan for the work that we were able to accomplish together and for the many late nights spent on paper submissions. I am thankful for the post-doctoral fellows that I have had the pleasure of working with throughout the years, including Timmie Smith, Nathan Thomas, Milan Hanuš and Mauro Bianco. I would like to thanks all of the fellow students that I have worked with, including Ioannis Papadopoulos, Mani Zandifar, Antal Buss, Sam Ade Jacobs, Francisco Coral Sabido, Colton Riedel, Dielli Hoxha, Brandon West, Shishir Sharma, Gabriel Tanase, Alireza Majidi, Olga Pearce, Xiabing Xu, Nedhal Mourad, Vincent Marsy, Nicolas Castet, Joshua Wright, Jeremy Vu and Daniel Latypov.

Finally, I would like to thank my friends and family for being there for me, especially

when there was no light to be seen at the end of the tunnel. I would like to especially thank Anna Villarreal, for which her love and support has truly made it possible for me to be here today.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a dissertation committee co-chaired by Professor Nancy M. Amato and Professor Lawrence Rauchwerger of the Department of Computer Science and Engineering, as well as committee members Professor Jennifer L. Welch of the Department of Computer Science and Engineering and Professor Nick Duffield of the Department of Electrical and Computer Engineering.

The Bounded Asynchrony section of Chapter 3 was conducted in collaboration with Harshvardhan and published in an article in 2014 [6] and the motion planning case study was in collaboration with Shishir Sharma and Sam Ade Jacbobs and published in an article in 2014 [7].

The work in Chapter 4 was conducted with assistance from Francisco Coral Sabido and Colton Riedel and published in an article in 2016 [8].

Elements of Chapter 5 were developed in conjunction with Ioannis Papadopoulos and published in an article in 2015 [9].

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

Funding Sources

Graduate study was partially supported by a diversity fellowship from Texas A&M University.

This research supported in part by NSF awards CNS-0551685, CCF-1439145, CCF-1423111, IIS-0916053, IIS-0917266, EFRI-1240483, RI-1217991, by NIH NCI R25 CA090301-11, and by DOE awards DE-NA0002376, B575363. by Samsung, IBM, In-

tel, and by Award KUS-C1-016-04, made by King Abdullah University of Science and Technology (KAUST).

This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

NOMENCLATURE

STAPL	Standard Template Adaptive Parallel Library
SGL	STAPL Graph Library
KLA	k-level-asynchronous
BFS	Breadth-first search
MTEPS	Millions of Traversed Edges Per Second
BSP	Bulk synchronous parallel

TABLE OF CONTENTS

Page

ABSTRACT ii

DEDICATION iii

ACKNOWLEDGMENTS iv

CONTRIBUTORS AND FUNDING SOURCES vi

NOMENCLATURE viii

TABLE OF CONTENTS ix

LIST OF FIGURES xii

LIST OF TABLES xv

1. INTRODUCTION 1

    1.1 Contributions 2
    1.2 Outline 3

2. PARALLEL GRAPH PROCESSING 5

    2.1 Challenges 5
    2.2 Design Decisions 6
        2.2.1 Synchronization and Communication 6
        2.2.2 Push vs Pull 7
        2.2.3 Partitioning 8
    2.3 Related Work 9
        2.3.1 Big Data 9
        2.3.2 High Performance Computing 11

3. THE STAPL GRAPH LIBRARY 14

    3.1 Library Overview 14
    3.2 Algorithm Model 15
        3.2.1 Vertex Operator 16
        3.2.2 Neighbor Operator 17

3.2.3 Initialization Operator . . . 18
3.2.4 Application Interface . . . 18
3.2.5 Graph Processing Engine . . . 20
3.3 Bounded Asynchrony . . . 21
3.3.1 Determining k . . . 23
3.3.2 Experimental Evaluation . . . 25
3.4 Load Balancing: A Motion Planning Case Study . . . 27
3.4.1 Parallel Motion Planning . . . 28
3.4.2 Load Balancing For Parallel Motion Planning . . . 33
3.4.3 Experimental Evaluation . . . 37
3.4.4 Implementation in STAPL Graph Library . . . 37
3.4.4.1 Model PRM Environment . . . 38
3.4.4.2 Experimental Results . . . 40

4. APPROXIMATE GRAPH COMPUTING . . . 45

4.1 Approximate Breadth-First Search . . . 46
4.1.1 Algorithmic Description . . . 46
4.1.2 Error Bounds . . . 49
4.1.3 Bounds with Tolerance . . . 51
4.1.4 Combined Bounds . . . 53
4.2 Implementation . . . 55
4.3 Experimental Evaluation . . . 55
4.3.1 Breadth-First Search . . . 56

5. NESTED PARALLEL GRAPH PROCESSING . . . 60

5.1 Adjacency List Partitioning . . . 62
5.1.1 Gang Specification . . . 62
5.1.2 Edge Container Partitioning . . . 63
5.1.3 Distribution Strategies . . . 65
5.2 Algorithmic Expression . . . 66
5.3 Implementation . . . 68
5.3.1 The STAPL Graph Library . . . 68
5.4 Experimental Evaluation . . . 69
5.4.1 Graph Algorithms . . . 70

6. GRAPH PROCESSING PARADIGMS . . . 73

6.1 KLA Machine . . . 73
6.1.1 Processing Loop . . . 76
6.1.2 Breadth-First Search . . . 77
6.1.2.1 Example . . . 77

6.2 Using the KLA Model . . . 79
6.2.1 Proof Through Witness . . . 79
6.2.2 Proof Through Impossible Schedule . . . 80
6.3 Results . . . 80
6.3.1 Priority Scheduler Bounds . . . 80
6.3.1.1 Problem Description. . . . 81
6.3.1.2 Worst-case Schedule . . . 81
6.4 Conclusion . . . 82

7. IMPLEMENTATION FOR PRACTICAL PERFORMANCE . . . 83

7.1 Frontier Representation . . . 83
7.1.1 Algorithm Characterization . . . 85
7.1.1.1 Active Vertex Ratio . . . 85
7.1.1.2 Superstep Occupancy . . . 85
7.1.1.3 Ordering . . . 85
7.1.2 Frontier Selection Framework . . . 88
7.1.3 Experimental Evaluation . . . 90
7.2 Graph Runtime Implementation . . . 91
7.2.1 Graph Processing Engine . . . 91
7.2.2 Graph Data Structure . . . 92

8. CONCLUSION . . . 94

REFERENCES . . . 95

APPENDIX A. A FRAMEWORK FOR PARALLEL SOFTWARE EXPERIMENTS 115

A.1 Motivation and Related Work . . . 115
A.2 Experiment Organization . . . 117
A.2.1 Experiment . . . 118
A.2.2 Result . . . 119
A.2.3 Observation . . . 119
A.2.4 Experiment Set . . . 121
A.3 Job Submission Workflow . . . 121
A.4 Conclusion . . . 123

APPENDIX B. NESTED PARALLELISM FOR HIERARCHICAL GRAPH ALGORITHMS . . . 124

B.1 Parallel Wavefront Computation . . . 125
B.1.1 Technique . . . 126
B.2 Preliminary Implementation . . . 127

LIST OF FIGURES

FIGURE Page

3.1 Common parallel graph algorithms provided by STAPL Graph Library. . . 15

3.2 Example label-propagation algorithm with its vertex, neighbor and initialization operators 16

3.3 API for user-provided operators and visitation methods. 19

3.4 Example usage of `sgl::execute` to start the graph processing engine. 19

3.5 Performance of KLA vs. Level-Sync BFS on synthetic road network to 98,304 cores on Hopper. Reprinted with permission from [6]. 26

3.6 Performance of KLA pointer jumping on a single-root list on 4096 cores on Hopper. 27

3.7 Nested KLA interoperability with Green-Marl for (a) PageRank and (b) conductance, up to 24,000 cores. Reprinted with permission from [6]. . . 28

3.8 A 2D environment subdivided into 4 regions and the corresponding region graph. Reprinted with permission from [7]. 31

3.9 Example of uniform radial subdivision for a 2D $\mathbb{C}_{space}$. Each process concurrently builds a branch (using sequential RRT) rooted at q_{root} and biased toward a target (e.g., q_k for the blue process). Reprinted with permission from [7]. 33

3.10 An environment where a regular partition would result in load imbalance. Reprinted with permission from [7]. 35

3.11 Experimental validation of (a) measure of load imbalance and (b) potential improvement in model environment. Reprinted with permission from [7]. 36

3.12 Evaluation of (a) execution time and (b) coefficient of variation and (c) load distribution for PRM on HOPPER using med-cube. Reprinted with permission from [7]. 38

3.13 Evaluation of computing roadmap in the med-cube environment for a rigid body robot on Hopper. Reprinted with permission from [7]. 41

3.14 Breakdown of (a) the various phases of PRM (b) and the effect of load balancing on remote accesses. Reprinted with permission from [7]. 41

3.15 Execution time for PRM with various load balancing strategies in (a) med-cube (b) small-cube (c) and free environment. Reprinted with permission from [7]. 42

3.16 Breakdown of the amount of tasks stolen vs. executed locally for PRM on (a) 96 and (b) 768 cores on HOPPER. Reprinted with permission from [7]. 42

3.17 Execution time for RRT with various load balancing strategies in (a) mixed (b) mixed-30 (c) and free environment. Reprinted with permission from [7]. 43

4.1 Example graph showing two different paths from the source to a vertex v. Reprinted with permission from [8]. 52

4.2 Computed distance $d_k^\tau(v)$ vs actual distance $d_0(v)$ for multiple τ and fixed k. Reprinted with permission from [8]. 55

4.3 Approximate BFS with tolerance heuristic on TX road network with 512 cores on Cray evaluating (a) runtime and (b) error. Reprinted with permission from [8]. 56

4.4 Approximate BFS with tolerance heuristic on TX road network with 512 cores on CRAY-XK7 evaluating (a) number of repropagations that occur during traversal and (b) speedup over the fastest k. Reprinted with permission from [8]. 57

4.5 Strong scaling of approximate BFS on IBM-BG/Q platform evaluating sensitivity of (a) runtime and (b) error. Reprinted with permission from [8]. 58

4.6 Approximate BFS with tolerance heuristic on random neighborhood network ($n = 1,000,000$ and $m = 16$) with 512 cores on CRAY-XK7. Reprinted with permission from [8]. 58

5.1 All locations gang specifier 63

5.2 Striped gang specifier 63

5.3 Adjacency list partitioning strategies for an (a) input graph. 65

5.4 The vertex, neighbor and scatter operators for single-source shortest path. 67

5.5 Graph 500 breadth-first search on Cray varying (a) the number of hubs on 512 processors and (b) the number of processors for a weak scaling experiment. Reprinted with permission from [9]. 69

5.6 Graph 500 (a) breadth-first search with various adjacency distributions on BG/Q and (b) various graph analytics algorithms on Cray. Reprinted with permission from [9]. 71

6.1 The vertex- and neighbor-operators for breadth-first search. 74

6.2 Example graph that shows the worst-case bounds even in the presence of priority scheduling. The vertices in the graph contain the computed distances and the associated diamonds are the true distances for vertices of interest. 82

7.1 Architecture of frontier selection framework 89

7.2 Preferable frontier data structure based on workload features. 90

7.3 Implicit frontier optimization vs using an explicit frontier with bitmap storage with the PageRank algorithm 91

7.4 Effect of interleaved ordering restriction vs no ordering restriction for breadth-first search on a single core of CRAY-XK7. 92

7.5 Runtime of a breadth-first search using task-based vs raw message based interface on BG/Q. 93

A.1 Organization of an (a) experiment and (b) result in Dimebox. 117

A.2 Example Dimebox experiment file describing a graph workload. 118

A.3 Organization of a set of experiments in Dimebox. 121

A.4 An example Dimebox workflow to run an experiment. 122

B.1 Example decomposition of original spatial domain into a multi-level hierarchical graph 126

LIST OF TABLES

TABLE Page

7.1 Example algorithm superstep occupancy 86

1. INTRODUCTION

Graphs are data structures that represent relationships, and there are many types of relationships that appear in the real world. Recently, there has been an explosion in the amount of data that has been captured through real-world processes. Because of this, analyzing extremely large graphs has become a critical component in a variety of fields of study, including physics, chemistry, biology, social analytics and many others. In bioinformatics, metagenomic assembly is performed by processing large de Bruijn graphs to reconstruct genomes from DNA sequencing reads into a metagenome [10]. For robotic motion planning, large graphs called roadmaps are used to plan the motion of robots [11, 12] through various kinds of physical environments. Traversals of graphs representing a physical domain are used to model neutron transport [13] to solve various nuclear physics problems. Particle movement problems, modeled as n-body simulations, can be solved using various large tree methods such as Barnes-Hut [14]. In national security use cases, large scale graph processing can be used to predict key players in covert networks [15] through various centrality metrics, such as betweenness centrality [16] and closeness centrality [17].

As the size of real-world graphs becoming increasingly large, it is often impossible to fit modern data sets into the memory of a single-processor system. Further, their size highlights the need for fast and efficient methods to process such graphs. Because of this, there exists a clear need for distributed data structures and parallel algorithms to facilitate the processing of these large graphs. However, the irregular access pattern for graph workloads, coupled with complex graph structures, varying topology, and large data sizes make efficiently executing parallel graph workloads challenging.

One major challenge that arises for parallel graph processing is workload imbalance between the processors, which is exacerbated by the straggler effect [18] with typical level-

synchronous [19] execution strategies. In this dissertation, we provide several ways to approach this challenge, from load balancing the problem statically to using an execution strategy that is more asynchronous in nature.

Many real-world graphs that have scale-free [20] properties present another challenge for distributed graph processing. Scale-free graphs have a small number of vertices–named *hub vertices*–that have a disproportionately large number of incident edges which would overload any processor that is assigned these vertices. We introduce a technique to process hub vertices in a nested parallel manner to better distribute the work and communication associated with the hub, as well as exploit the natural hierarchical nature of modern computer architectures.

Lastly, an empirical study of many distributed graph processing frameworks [21] shows that an optimized sequential implementation of a graph algorithm can often outperform a highly parallel execution utilizing hundreds of cores due to the fact that many parallel frameworks have significantly high serial overhead. This dissertation provides guidance on how a graph framework can tailor its constituent components to extract the most performance from a given workload by selecting an appropriate graph data structure, runtime implementation and active vertex set representation.

1.1 Contributions

In this dissertation, we develop several broad techniques for improving the performance of parallel graph traversals and general parallel graph algorithms:

- **Allowing a traversal to proceed asynchronously in a bounded manner.** Increasing asynchrony in a bounded manner avoids costly global synchronization at scale, while still alleviating the penalty of unbounded asynchrony including redundant work [6]. In addition, asynchronous processing enables a new family of approximate algorithms when applications are tolerant to a fixed amount of error [8].

- **Exploiting the machine hierarchy using nested parallelism.** Allowing to express graph algorithms in a naturally nested parallel manner enables us to fully exploit all of the available parallelism inherent in graph algorithms [9].

- **Tailoring the graph framework's components for a given workload.** Graph processing workloads are highly input-sensitive and require careful consideration about the implementation of the various components in a graph framework. We provide guidance to inform the choice between various execution policies, frontier storage strategies, data structure representations and runtime implementations.

Using bounded asynchrony, we are able to scale a breadth-first search (BFS) workload to 98,304 cores [6], where traditional techniques stop scaling at smaller core counts. Through the use of nested parallelism, we are able to process scale-free graphs up to 1.6x faster [9] and are able to fit graphs into memory that would otherwise overload the memory capacity of a node using traditional methods. By tailoring the graph framework's components to a given workload, we are able to achieve an order of magnitude in performance improvements compared with a naive and input-agnostic approach.

1.2 Outline

The content of this dissertation is organized into seven chapters as follows. Chapter 2 describes parallel graph processing in general, the challenges that arise in such workloads and a summary of related work in the field.

In Chapter 3, we describe the design and implementation of the STAPL Graph Library (SGL) and a summary of its components, design decisions and a few use cases.

In Chapter 4, we introduce a novel approximate algorithm for breadth-first search that utilizes the concept of bounded asynchrony and provides direction for further approximate graph algorithms using the same technique.

In Chapter 5, we demonstrate how nested parallelism can be used to improve the performance of graph algorithms for certain kinds of scale-free graphs.

Chapter 6 unifies the various execution policies of the STAPL Graph Library and provides a mathematical foundation of the SGL execution model.

Finally, Chapter 7 outlines various considerations that a real-world graph processing framework must take into account to achieve not just scalability, but high performance sequentially and on single shared-memory nodes.

Appendix A introduces a framework for reproducible systems software research while Appendix B provides a case study for a nested hierarchical wavefront algorithm.

2. PARALLEL GRAPH PROCESSING

Parallel graph processing systems can be broadly placed into two related but distinct categories: graph-targeted big-data processing systems [22] and high-performance graph computing. Although these areas are highly related, there are clear distinctions between them. With big-data processing systems, the architecture that such systems are designed to work with are typically clusters of commodity hardware, with off-the-shelf networking equipment. Such setups are often provisioned using a cloud compute service, such as Amazon Web Services [23] , Google Cloud Platform or Microsoft Azure.

Although the work in this dissertation is mainly targeted for high-performance computing scenarios, much of the presented techniques can be easily adapted to big-data processing systems. Conversely, work in the area of big-data processing systems has influenced work in this dissertation as well.

2.1 Challenges

Computational patterns that emerge in graph algorithms offer many challenges for executing such workloads efficiently. Lumsdaine et. al [24] concisely describe several fundamental characteristics of graph algorithms which hinder their performance:

- **Data-driven computation.** The computation of a graph algorithm often follows the edges of the graph itself, rather than being explicitly expressed in the program. Thus, the computation and communication of a parallel graph algorithm directly corresponds to the graph structure and is highly input dependent.

- **Unstructured problems.** Unlike structured problems, such as those operating on regular arrays of data, the irregularity of graph data makes it difficult to establish a partition of the problem where computation and communication is balanced across

the parallel machine.

- **Poor locality.** Because graph traversals typically compute through the relationships in the graph itself, access to the entities exhibit irregular patterns and suffer from poor spatial locality. Additionally, data is usually accessed once and therefore, there is little temporal locality.

- **Low amount of computation per unit of data.** In many graph algorithms, the actual computation for a single vertex or edge is usually a small number of operations, resulting in most of the execution time being spent accessing and communicating data, rather than performing operations on the data.

In the following section, we describe the design decisions we make in this dissertation to overcome these challenges and how these decisions enable the creation of a scalable and high-performance graph processing framework.

2.2 Design Decisions

As there exist many frameworks and approaches for processing large graphs in parallel, the design decisions made for each approach have impacts in many aspects, including scalability, ease of use and the types of workloads that are amenable with the approach.

2.2.1 Synchronization and Communication

For distributed graph processing frameworks, their synchronization and communication models are two of the most fundamental aspects that affect the performance of workloads using that framework. Firoz el. al [25] categorize features of graph processing frameworks into various classification, one of which being the orderedness of the computation. Some graph frameworks provide an ordered model, such as the level-synchronous model [26, 5] where the computation proceeds iteratively in the graph level by level. Another point on this spectrum is an unordered model such as asynchronous where global

synchronizations are replaced with point-to-point synchronizations, which can increase the degree of parallelism, but which may also introduce redundant work.

There are many frameworks that utilize a level-synchronous model [27, 3] and many that use an asynchronous model [28, 29]. Neither model provides superior performance against a diverse set of graph analytics workloads, and thus the type of model to use is dependent on the workload itself.

In this dissertation, we utilize a hybrid model named k-level asynchronous (KLA) where the asynchrony is bounded, which allows parametric control of asynchrony ranging from completely asynchronous execution to partially asynchronous execution to level-synchronous execution.

2.2.2 Push vs Pull

When a framework processes a single vertex in a graph algorithm, the direction in which data flows characterizes whether that framework is push-based or pull-based. A *push-based* framework sends data from that vertex outward to its neighbors. A *pull-based* framework reads data from its neighbors to compute the vertex's new value. Some hybrid frameworks, such as Ligra [30] and Grazelle [31] are able to adaptively switch between push and pull based on the given workload.

Distributed graph frameworks naturally lend themselves to push-based models, as vertices can asynchronously send data to their neighbors and overlap this communication with processing of other vertices. Pregel[3] popularized the push-based method and many graph frameworks have followed the same model.

It is typical for pull-based models to be used in shared-memory graph frameworks, due to the need for reads of arbitrary portions of the graph for each vertex that is processed. A naive implementation of a pull-based model in distributed memory would incur a large amount of synchronous reads, resulting in degraded performance. However, there exist

distributed graph frameworks such as GraphLab [32] and PowerGraph [33] which utilize a pull-based model. These frameworks are able to use techniques such as caching of local vertices on remote processors to minimize the number of fine-grained remote reads.

In this dissertation, we utilize a graph algorithm model that follows the push method. As our focus is on distributed-memory architectures, the push model will allow us to achieve high asynchrony and thus high scalability.

2.2.3 Partitioning

How a graph's elements are partitioned across a distributed system has a large impact on any computation that is performed over that graph. As many graph computations involve a traversal of the graph's edges, an edge spanning vertices that are stored in different memory address spaces represents communication across the network to traverse the edge. Minimizing the number of *cut edges*–that is, edges that span multiple partitions– is crucial for obtaining scalable performance with low amounts of communication. At the same time, it is important to balance the amount of work in each partition by maintaining a relatively equal amount of vertices (or in some cases, the work that a vertex represents) for each partition. The problem of partitioning a graph that both minimizes the number of cut edges and equalizes the vertex set across the partitions is well known to be NP-complete [34], and thus many heuristic algorithms have been proposed.

Multilevel graph partitioners, such as METIS [35], have been shown to work well in practice. Many general and reusable parallel packages [36, 37] that partition a graph have been proposed and implemented that work well for a broad variety of graphs.

It has been observed that graphs whose degree distributions follow a power-law are especially difficult to partition effectively [38]. Many techniques have been introduced to more efficiently work with power-law scale-free graphs in parallel. One heuristic that has been show to be scalable and produce good quality partitions is using label-propagation

community detection algorithms to create the partitioning [39].

However the presence of hub vertices presents challenges to keeping a balanced number of vertices and edges per processor using traditional vertex-centric partitioning, as the placement of a hub could overload any one processor. More sophisticated types of partitioning have been proposed, including checkerboard 2D adjacency matrix partitioning [40], edge list partitioning [41] and specialized techniques for distributing and processing hub vertices [28, 33, 42].

In addition to the partitioning of the graph, how the vertices are ordered within a partition can also have a large impact on performance of typical graph workloads. McSherry et. al. [21] show that using a Hilbert curve ordering for vertices within a partition can increase locality and improve performance of PageRank. Another work shows that reordering vertices based on frequency (a derivative of degree ordering) improves cache line reuse in some cases [43].

2.3 Related Work

The vertex-centric programming model, popularized by Pregel [3] and its open-source equivalent Giraph [44], has become a standard in parallel graph processing. The so-called *think like a vertex* [45] paradigm allows algorithm writers to express their computation in terms of vertex programs, which describe the operations to be executed on a single vertex and its incident edges.

2.3.1 Big Data

Although the term *big data* has no formal definition and opinions differ on what exactly constitutes big data, the seminal big-data project Hadoop [46] defined big data as "datasets which could not be captured, managed, and processed by general computers within an acceptable scope" [47]. One of the key differences of a big-data framework vs a high-performance computing framework is the design philosophy of big-data frameworks

being devised for clusters of commodity hardware, whereas high-performance computing frameworks are tailored with supercomputer architectures in mind. For example, common big data frameworks such as Hadoop and Spark [48] often use a distributed file system to store intermediate results during a computation, whereas a high-performance computing framework will typically assume that the machine has enough memory to store all necessary immediate values and avoid using I/O altogether during computation.

When it comes to big data processing of graph data, Pregel and Giraph sparked a trend in graph-specific frameworks to perform large scale analysis of relational data. GraphX [27] is a state-of-the-art big data framework for graph processing built on top of Spark and is widely used in many industries. Many other frameworks [49, 50, 51, 52, 53, 54, 55, 56] have been since introduced to process large graph inputs on clusters of commodity architectures. Although these frameworks introduce concepts that are in the same spirit of the work in this dissertation, we focus mainly on supercomputing architectures.

Many big-data graph processing frameworks assume that the input graph is too large to fit into memory, even when utilizing the combined memory capacity of an entire cluster. For these scenarios, optimized usage of I/O is key and semi-external memory graph processing frameworks [57, 41] often develop techniques to cache and reuse data as much as possible to minimize disk access. Similarly, there has been a large amount of work to process large graph inputs on single commodity machines, which often use out-of-core memory techniques to process inputs too large to fit into memory. These frameworks (e.g., GraphChi [58], GridGraph [59] and GraphMP [60]) typically split the graph inputs into smaller shards and process them one at a time, or process the algorithm in a sliding-window fashion such as in GraphMP.

In addition to graph processing systems, there are several projects that aim to provide tools for online transaction processing (OLTP) and online analytical processing (OLAP) specifically for graph workloads. Many of these projects aim to provide graph databases

that can easily represent relational data and perform online queries about the graph. Neo4j [61] is a popular graph database management system that provides a persistent data store as well as an API to issue graph queries in a custom graph query language. Apache TinkerPop [62] is a project that aims to standardize OLTP and OLAP graph frameworks by providing an extensive framework with several graph storage backends and querying systems. Although graph OLTP and OLAP handle querying potentially distributed graphs, their use cases are typically distinct from standard graph processing and thus are often not used for the type of large scale, highly parallel workloads that we target in this dissertation.

Also in the realm of big-data processing of large graph datasets are several tools and frameworks that help bridge commodity big-data processing frameworks to run in high-performance computing environments. Singularity [63] is a solution to run containers on standard supercomputing platforms, which allows for developers and analysts to package their big-data suite of tools into a Docker [64] container and execute their workloads on a highly-coupled modern supercomputer with only modest loss of performance. Magpie [65] is another tool whose aim is to allow big-data frameworks based on Spark [48] to run on standard supercomputers. Although these approaches help users migrate their existing big-data workloads to supercomputers, the resulting performance of such solutions is often much lower than using a framework that is specifically designed to be run in a supercomputing environment.

2.3.2 High Performance Computing

Many general purpose frameworks and runtimes [66, 67, 30, 1] for graph processing have been proposed and are used in practice. Galois is an amorphous data parallel processing framework with support for many vertex-centric paradigms [68]. Grappa [66] is a distributed shared memory framework designed specifically for data-intensive applications. Graph-based domain-specific libraries [69] exist and have been shown to perform

well in practice. Abstract Graph Machine [70] is a model for parallel graph algorithms that allows for the expression of various graph algorithms and schedulings. PGX.D [71] is a graph library that follows a BSP model with a remote reads and provides several optimizations for edge partitions and ghost-noding. Various frameworks have been proposed that exploit the non-uniform memory access (NUMA) nature of modern multi-core architectures including Polymer [72] and GraphGrind [73].

There exist many frameworks outside of the vertex-centric model, ranging from subgraph-centric graph processing [74] to path-centric models [75] to sparse linear algebraic representations [76, 77, 78, 79]. Some hybrid models have been proposed as well, including DRONE [80] which presents a hybrid vertex-centric and subgraph-centric framework. Although these models have been shown to provide better performance for certain types of graph analytics, we focus on the vertex-centric model because of its ease of use and expressivity.

Many techniques have been proposed specifically to improve breadth-first search. Most notably, the Graph 500 benchmark [81] has sparked much research into improving breadth-first search on scale-free networks for distributed-memory architectures [82, 40]. A hybrid top-down bottom-up breadth-first search was presented in [83] that shows large improvement on scale-free networks. Checconi and Petrini [84] provide insight into the kinds of optimizations needed to achieve performance surpassing trillions of traversed edges per second and introduce a hybrid 1D-2D partition where hubs are distributed across processors, while non-hubs follow a traditional 1D partition.

As accelerators increase in popularity, there has been a large range of work for accelerating graph workloads on these devices. There exist many recent frameworks [85, 86, 87, 88, 89] for graph processing on GPUs. In addition, many specific graph algorithms have been shown to benefit from accelerators, including breadth-first traversals [90, 91, 92, 93], connected components [94], graph coloring [95] and PageRank [96]. As accelerators are

typically coupled with a multi-core processor in modern architectures, there have been attempts to develop unified heterogeneous graph frameworks that utilize both the CPU and accelerator such as in Gluon [97]. Beyond GPUs, graph specific accelerators have been proposed [98, 99, 100] and have been shown to work well in practice. In this dissertation, we focus primarily on graph processing on parallel systems comprised of CPUs. However, there is potential for future work that adapts the work presented in this dissertation to heterogeneous systems.

3. THE STAPL GRAPH LIBRARY[1]

The work in this dissertation is implemented using the STAPL Graph Library (SGL) [101, 102, 6]. SGL is a generic parallel graph library that provides a high-level framework that abstracts the details of the underlying distributed environment. It consists of parallel graph containers (`pGraph` s), views that allow these containers to be accessed in custom ways, a collection of parallel graph algorithms, and a graph execution model that allows developers to easily create their own custom graph algorithms.

3.1 Library Overview

The fundamental component of SGL is the `pGraph` container. The `pGraph` container is a distributed data store built using the `pContainer` framework (PCF) [103] provided by the Standard Template Adaptive Parallel Library (STAPL) [104]. It provides a shared-object view of graph elements across a distributed-memory machine. The STAPL Runtime System (STAPL-RTS) [105] and its communication library ARMI (Adaptive Remote Method Invocation) [106] use the remote method invocation (RMI) abstraction to allow asynchronous communication on shared objects while hiding the underlying communication layer (e.g MPI, OpenMP).

Containers store actual graph elements (i.e., vertices and edges), but algorithms operate on `pViews` [107], which provide an abstraction of the container. The `pView` construct operates as an intermediary between the data and how it is accessed, similar to the iterator concept in the C++ standard library [108].

[1]Part of this chapter is published in "Harshvardhan, Adam Fidel, Nancy M. Amato, and Lawrence Rauchwerger. 2014. KLA: a new algorithmic paradigm for parallel graph computations. In Proceedings of the 23rd international conference on Parallel architectures and compilation (PACT '14). ACM, New York, NY, USA, 27-38.". Additionally part of this chapter is reprinted with permission from Adam Fidel, Sam Ade Jacobs, Shishir Sharma, Nancy M. Amato, Lawrence Rauchwerger, "Using Load Balancing to Scalably Parallelize Sampling-Based Motion Planning Algorithms," 2014 IEEE 28th International Parallel and Distributed Processing Symposium, May 2014. Copyright 2014 IEEE.

Algorithm	Description
Breadth-first search	Unweighted shortest path tree from a single vertex
Single-source shortest path	Weighted shortest path tree from a single vertex
Connected components	Determine connectivity of the graph
PageRank	Estimate importance of vertices based on random walks
Betweenness centrality	Centrality of vertices based on shortest path betweenness
Approximate diameter	Estimate diameter of graph d within O($2d$)
Triangle count	Count the number of closed triangles
Link prediction	Predict new edges based on current connectivity
k-core	Compute smaller representative version of a large graph
Community detection	Community vertex membership based on label propagation

Figure 3.1: Common parallel graph algorithms provided by STAPL Graph Library.

A suite of common parallel graph algorithms is provided for users to immediately use, including breadth-first search, connected components, PageRank, single-source shortest path, betweenness centrality, triangle count and many more. Table 3.1 provides a small sample of the algorithms provided by the library along with a short description of their computation. In total, there are 30 parallel graph algorithms provided by the STAPL Graph Library. The following section describes the model in which these algorithms are written and provides the background necessary for algorithm developers to create their own algorithms in the same model.

3.2 Algorithm Model

The algorithm model for SGL is a push-based vertex-centric model consisting of three fine-grained operators: a vertex operator, a neighbor operator and an initialization operator. Each operator receives a single vertex as a parameter and returns a boolean. The meaning of the return differs depending on the operator. The algorithm processes these operators in an iterative manner, where each iteration is called a superstep.

```
struct my_vertex_operator {
  bool operator()(Vertex v, Visitor vis) {
    // Initiate visitation to all neighbor visitors and pass some value
    int label = v.property();
    vis.visit_all_edges(neighbor_operator{label});
    // Vote to continue
    return true;
  }
};

struct my_neighbor_operator {
  int incoming_value;
  bool operator()(Vertex v) {
    // Update property if incoming label is smaller
    if (incoming_value < v.property()) {
      v.property() = incoming_value;
      // Continue traversal
      return true;
    }
    // Do not re-invoke vertex operator
    return false;
  }
};

struct my_initialization_operator {
  bool operator()(Vertex v) {
    // Set the initial label to be its ID
    v.property() = v.descriptor();
    return true;
  }
};
```

Figure 3.2: Example label-propagation algorithm with its vertex, neighbor and initialization operators

3.2.1 Vertex Operator

A vertex operator defines the computation that occurs on a single vertex and is triggered for all active vertices in the active set, often called the *frontier*. Typically, the vertex operator will perform one or both of two actions

1. Mutate the vertex's properties based on the actions that have been performed either during initialization or during the execution of visits from the neighbor operator.

2. Send some value derived from the vertex and its properties to other vertices in the graph.

For example, in the PageRank [109] algorithm, the vertex operator takes its current rank value–which has either been initialized from the beginning or has been updated through neighbor visits in previous iterations–and computes its new rank value. Then this new rank value, divided by the vertex's degree, is sent to all of its neighbors.

As a comparison with other graph frameworks and models, the vertex operator is similar to the sending portion of Pregel's `Compute` method [3], the enqueuing portion of HavoqGT's visitor [41] and the `CreateMessage` method in ASYMP's programming model [29].

The return value of the vertex operator is a vote to continue. If all of the vertex operators return false in a given iteration, then the execution terminates.

3.2.2 Neighbor Operator

The algorithm's neighbor operator describes the computation that occurs when a vertex is visited from another vertex through an edge. Typically, the neighbor operator carries some state in it which will be used to either update the vertex's property or determine whether or not the traversal will continue along the path including this vertex.

The return value of the neighbor operator indicates whether or not the vertex should be active in the next superstep. That is, a return value of true indicates that the vertex will be added to the frontier of the next superstep and the vertex operator will be invoked on it.

Figure 3.2 gives a vertex operator and a neighbor operator for a simple label-propagation algorithm. Whenever a vertex is processed by the vertex operator, the vertex's property is sent to all if its neighbors. In the neighbor operator, the incoming label is compared against its current label and updated if it is smaller. The traversal will continue only if the label was updated. In the end, a vertex v will be labeled with the smallest initial label from a vertex u provided that u is reachable from v.

Note that in our owner-computes programming model, each vertex is only ever pro-

cessed by exactly one processing element (e.g., thread). Therefore, there is no need to guard the operators with any mutual exclusion technique.

The neighbor operator is similar to the individual operations of Pregel's `Compute` method that process messages, [3], the beginning portion of HavoqGT's visitor [41] and the `ReceiveMessage` method in ASYMP's programming model [29]. Unlike similar graph models, such as in Pregel and HavoqGT, a graph algorithm in SGL splits its sending and receiving computations into two separate operators. By having two operators, we are provided flexibility on when to schedule these operators. This is useful for our different execution policies (particularly k-level asynchronous) which will be discussed in a later section.

3.2.3 Initialization Operator

The final operator used to specify a graph algorithm is the initialization operator. It is a fine-grained vertex centric operator that initializes the state of each vertex and returns a boolean indicating whether or not a given vertex should be processed in the first superstep. Specifically, the return value determines if a vertex appears in the frontier, which is described in detail in Section 7.1.

In Figure 3.2, the initialization operator first sets the label of each vertex to be its ID and then marks all vertices as active in the first superstep.

3.2.4 Application Interface

A synopsis of the current interface for the user-provided operators and visitation methods is provided in Table 3.3.

Once the user has written a vertex-operator and neighbor-operator to describe their algorithm, they can start execution of the algorithm by passing it to the function `sgl::execute`. This function represents the graph processing engine, described further in Section 3.2.5. Figure 3.4 provides an example graph algorithm implementation that an algorithm devel-

Method	Description
`bool vertex_operator::operator()(Vertex, Visitor)`	Method that is called for every active vertex in the frontier.
`bool neighbor_operator::operator()(Vertex)`	Method that is called upon visitation from a neighbor vertex.
`bool init_operator::operator()(Vertex)`	Method that is called once at the beginning of the algorithm to initialize the frontier and vertex state.
`void Visitor::visit_all_edges(Vertex, op)`	Visit all of the neighbors of the given vertex and apply the neighbor operator.
`void Visitor::visit_edge(Vertex, op, target)`	Visit a single neighbor and apply the neighbor operator.
`void Visitor::visit_edge_if(Vertex, op, pred)`	Visit all neighbors of a vertex that match a given predicate and apply the neighbor operator.

Figure 3.3: API for user-provided operators and visitation methods.

```
void my_algorithm(Graph g) {
  sgl::execute(
    execution_policy,
    my_vertex_operator,
    my_neighbor_operator,
    my_init_operator,
    superstep_occupancy,
    ordering
  );
}
```

Figure 3.4: Example usage of `sgl::execute` to start the graph processing engine.

oper would write. Besides the operators described previously, there are a few additional parameters to `sgl::execute` that control the behavior of the execution engine.

The first parameter, the execution policy, is an object describing the method by which the graph processing engine will execute the operators. The type of policies provided by SGL are level-synchronous, asynchronous [101], k-level asynchronous [6], hierarchical out-degree, hierarchical in-degree [102] and out-of-core [110].

The last two parameters (`superstep_occupancy` and `ordering`) are parameters that enable optimizations with the frontier representation. They are described in further detail in Section 7.1.

```
1  Frontier_current ← ∅
2  Frontier_next ← ∅
3  foreach v ∈ V in parallel
4  |  active ← InitOp(v)
5  |  if active then
6  |  |  Frontier_current ← Frontier_current ∪ v
7  |  end
8  end
9  continue ← true
10 while continue do
11 |  continue ← false
12 |  foreach v ∈ Frontier_current in parallel
13 |  |  continue ← continue∨ VertexOp(v, visitor)
14 |  end
15 |  Frontier_current ← ∅
16 |  Swap(Frontier_current, Frontier_next)
17 end
```

Algorithm 1: Logical outer loop of graph processing engine

3.2.5 Graph Processing Engine

The user-provided operators describe the semantics of a graph algorithm in its entirety. The method by which these operators are used to execute the graph algorithm are explained in Algorithm 1 and Algorithm 2.

In Algorithm 1, we see that the initial set of active vertices is empty. Next, the initialization operators are applied on all of the vertices in parallel and the vertices which the operator returned true are then added to the frontier of the first superstep. Afterwards, there is an outer loop over all supersteps. For a single superstep, each vertex has a vertex operator applied on it in parallel. The result of the vertex operator determines whether or not the algorithm should continue for another superstep or if it should end.

The user-provided vertex operator is free to make calls to the visitor object to spawn visitations to neighbors. One such visitation method (`Visitor::visit_all_edges`)

```
1 Function Visitor::visit_all_edges(v, NeighborOp)
2     foreach (s,t) in OutNeighbors(v) do
3         async
4             reinvoke ← NeighborOp(t)
5             if reinvoke then
6                 Frontier_next ← Frontier_next ∪ t
7             end
8         end
9     end
```

Algorithm 2: Internal visitation algorithm that applies neighbor operators. This is an example using the level-synchronous execution policy.

is described in Algorithm 2. In this function, a list of all out-going neighbors is retrieved from the vertex and each neighbor is visited asynchronously. At a lower-level, this visitation is handled using STAPL's remote-method invocation [105] support through MPI for distributed-memory or thread-local queues for shared-memory. During this asynchronous visitation, the user-provided neighbor operator is called on the neighbor vertex. Based on the return value of the neighbor operator, the vertex is then added to the frontier of the next superstep.

Note that Algorithm 2 provides an idealized visitation policy that implements a level-synchronous execution. For a fully asynchronous execution, the vertex operator could be directly invoked again instead of adding the vertex to the next frontier. In general, a new execution policy such as k-level asynchronous [6] and hierarchical communication reduction [102] will replace the neighbor visitation routine with a specialized version.

3.3 Bounded Asynchrony

Parallel graph algorithms are often expressed in level-synchronous or asynchronous paradigms. Level-synchronous paradigms [26, 5] iteratively process vertices of a graph level by level. This model guarantees the current level's computation to have completed

before starting the next one through the use of global synchronizations at the end of each level. Level-synchronous algorithms tend to perform well when the number of levels is small, but suffer from poor scalability when the number of levels is large. Bulk synchronous parallel (BSP) algorithms [19] can naturally be expressed in this paradigm. Iterative application of the map/reduce pattern [111] is one common way to implement level-synchronous algorithms. It proceeds by spawning tasks for neighbors of active vertices, and each task may proceed independently. The asynchronous paradigm replaces global synchronizations with point-to-point synchronizations, which can increase the degree of parallelism, but which may also require the completion of redundant work. The asynchronous paradigm replaces expensive global synchronizations with less expensive point-to-point fine-grained synchronizations, which potentially increases the available degree of parallelism and thus may perform better on graphs with many levels. However, asynchronous algorithms may perform redundant work. For example, an asynchronous breadth-first search may re-visit vertices multiple times as shorter paths are discovered [112]. Furthermore, as a practical limitation with most large-scale systems, too many messages in flight may choke the communication-layer, which is usually designed for few large messages as opposed to many small messages. For example, a large-degree vertex may generate many messages (one for each neighbor), which may lead to flooding the system. In general, the level-synchronous paradigm is better suited for small-diameter graphs with high-degree vertices, while the asynchronous paradigm is better for high-diameter graphs with low-degree vertices. Choosing the right paradigm depends on the system, input graph, and algorithm. This implies different implementations and optimizations for algorithms, with no easy way to switch between them.

The STAPL Graph Library provides the k-level asynchronous (KLA) paradigm [6], which allows parametric control of asynchrony ranging from completely asynchronous execution to partially asynchronous execution to level-synchronous execution. Partial asyn-

chrony is achieved by generalizing the BSP model to allow each superstep to process up to k levels of the algorithm asynchronously, balancing the cost of global synchronizations with the cost of redundant work to obtain improved execution time. In this respect, KLA may be viewed as a generalized BSP model where each superstep runs an asynchronous algorithm. A related approach is also used in the Δ-stepping single-source shortest path algorithm [113], which is a special-case of KLA.

This paradigm works in phases, similar to the BSP model. However, each phase is allowed to proceed asynchronously up to k steps by creating asynchronous tasks on active vertices. When $k = 1$, KLA processes the graph one level at a time, i.e., in a level-synchronous fashion. Assuming the level-synchronous variant of the algorithm performs d iterations, if $k \geq d$, then KLA is equivalent to the asynchronous paradigm. However, for $1 < k < d$, the algorithm proceeds in $\lceil \frac{d}{k} \rceil$ phases, referred to as *KLA supersteps* (KLA-SS). Each superstep processes up to k levels asynchronously. The KLA paradigm requires that the algorithmic invariant holds at the point of synchronization. However, in between synchronizations, no guarantee is implied.

This section is reprinted with permission from [6].

3.3.1 Determining k

The performance of KLA algorithms is dependent on the choice of a good value for k. However, it is non-trivial to determine the optimal value of k given an arbitrary input graph, machine and algorithm. There exist guidelines regarding choosing a reasonable value for k that are fairly simple. For example, it is often a good choice to use $k = 1$ for any scale-free graph due to the high potential for wasted work when encountering a hub vertex. Conversely, a large value of k is recommended for acyclic graphs, or any graph where wasted work would be minimal.

Besides following simple guidelines and manually choosing levels of asynchrony, we

also present an adaptive strategy that picks an appropriate k based on the current local topology of the graph as the algorithm progresses. This strategy is applicable in the general case.

Adaptive Determination of k

As it is often not feasible to run the same workload with different levels of asynchrony to choose the best value of k, the STAPL Graph Library provides a simple adaptive strategy (Adaptive KLA) that dynamically varies k to process the graph. The strategy dynamically chooses the best value for the next iteration based on information from current and previous iterations.

The algorithm starts with $k = 1$ (or some user-defined start value) and doubles the value of k after each superstep, provided the following conditions hold:

1. A high out-degree vertex has not been encountered
2. The asynchronous penalty (e.g., wasted work) has not exceeded a threshold
3. The per-vertex processing time of the current superstep has not risen sharply compared to previous supersteps

If any of these conditions are not met for the current superstep the k value for the next superstep is halved. In addition, any high out-degree vertices found in the current superstep are postponed for processing in the next superstep. This is to reduce the penalty for processing them prematurely. Lastly, if the asynchrony penalty or processing cost exceed a maximum threshold (we empirically found 20% to work well for the machines on which we tested), the value for k is capped, so the algorithm does not suffer from greater asynchronous penalties attempting higher values.

While this technique may not provide the best performance compared to knowing *a priori* the value of k_{opt}, it is inexpensive in practice.

3.3.2 Experimental Evaluation

In this section we evaluate the performance of KLA algorithms on different categories of graphs, both real-world and synthetic. We observed that for certain graphs, a value of $k > 1$ provides the best performance.

First we consider k-level asynchronous breadth-first search. In order to create a large graph with amenable characteristics, we generated a synthetic road network by stitching together multiple copies of the European road network from the SNAP collection [114], resulting in an input with 9.63 billion vertices and 10.2 billion edges. Figure 3.5 compares the scalability of KLA BFS with level-synchronous BFS on this road network. The level-synchronous BFS scales to 32,768 cores, while the KLA BFS is able to scale better until 98,304 cores and yield faster performance due to better balancing the synchronization costs with wasted work. This experiment demonstrates that the traditional level-synchronous and asynchronous paradigms are not always the best method for processing every graph input.

Besides breadth-first search, we evaluated KLA on a well-known pointer-jumping algorithm [115]. We implemented and ran our algorithm on a list graph, and observed that the pattern of computation for pointer-jumping can gain significant advantage from the use of the KLA paradigm. We considered two variants of the pointer-jumping algorithm: a fine-grained approach and a two-level nested KLA approach in which each list element contains a sublist which is processed sequentially. For the simple fine-grained algorithm, we observed that varying k gave a 50% benefit compared to the asynchronous algorithm, and 70% benefit compared to the level-synchronous version (Figure 3.6). The two-level version of pointer-jumping with the lower-level implementing sequential pointer-jumping resulted in further benefits due to decreased communication. Figure 3.6 shows the comparison of the two versions, varying k for the same input.

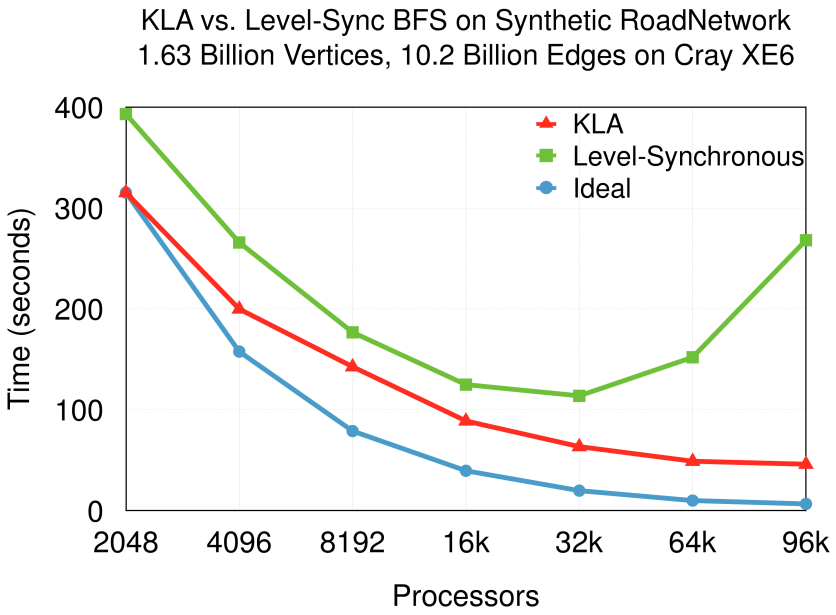


Figure 3.5: Performance of KLA vs. Level-Sync BFS on synthetic road network to 98,304 cores on Hopper. Reprinted with permission from [6].

Recursive Application of KLA and Interoperability.

KLA algorithms may also be called recursively in nested sections. We describe such computations using a tuple of k values, where each value in the tuple represents the k for each algorithm being executed in that level of nesting and the arity of the tuple denotes the number of levels of nesting in the algorithm. If the algorithm in the nested section is sequential or non-KLA, the k value for it is ignored ($k = x$). For example, using a shared-memory library for intra-node computation and KLA for inter-node computation, the tuple would be $\langle x, k_1 \rangle$. Figure 3.7 demonstrates the benefit of using nested-parallel execution in the multi-level coarse grained paradigm, which can take advantage of an optimized shared-memory library (Green-Marl [69]) for processing partitions of the graph on a shared-memory node and use KLA version of the algorithm on the upper-level to

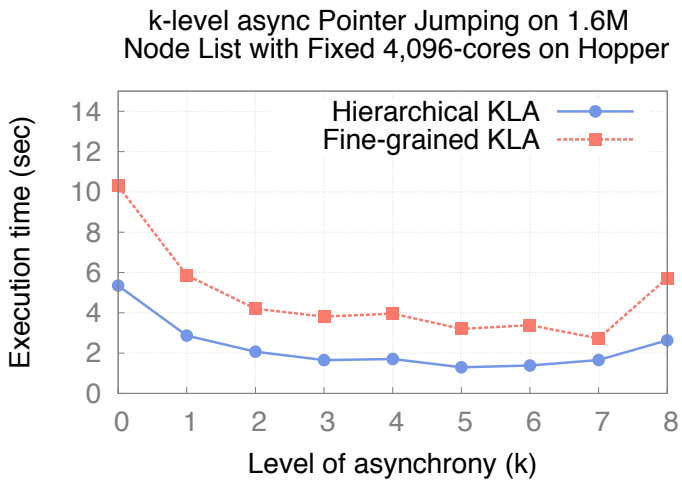


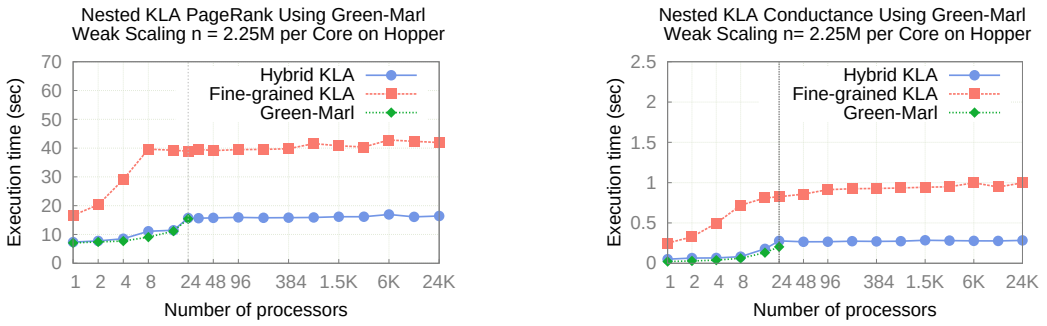
Figure 3.6: Performance of KLA pointer jumping on a single-root list on 4096 cores on Hopper.

extend it across the distributed nodes in the machine.

In our experiments, we ran two different algorithms – PageRank and cut-conductance – on a mesh graph using the Green-Marl implementation within the node and KLA off-node, and compared them with their single-level KLA implementations in SGL.

3.4 Load Balancing: A Motion Planning Case Study

Many graph applications suffer from issues with load imbalance due to the dynamic and irregular nature of graph algorithms. The STAPL Graph Library has rich support for load balancing of graph applications that come in two forms. The first form of load balancing is through achieving a well-balanced initial partition of the graph on the machine. This approach is often used when a good estimate of the computational load per graph element (i.e., vertex, edge) is known *a priori* and can be used to effectively induce a partition that balances the workload. The second form of load balancing is dynamic scheduling of work, typically through a method such as work stealing. This method is often used for dynamic applications or cases where it is difficult or costly to estimate the work of a given



(a) (b)

Figure 3.7: Nested KLA interoperability with Green-Marl for (a) PageRank and (b) conductance, up to 24,000 cores. Reprinted with permission from [6].

processor.

In this section, we discuss an example graph application from the field of motion planning and provide a case study of how load balancing techniques from STAPL Graph Library were utilized to provide a balanced allocation of work across a large machine. This section is reprinted with permission from [7].

3.4.1 Parallel Motion Planning

While motion planning has its roots in robotics, it now finds applications in other areas of scientific computing including protein folding [116], drug design [117] and virtual prototyping and computer-aided design [118].

Due to the infeasibility of exact motion planning, sampling-based methods are now the state-of-the-art for solving motion planning problems. Sampling-based motion planning is essentially a large-scale graph problem that first constructs a graph and subsequently solves queries using graph traversals. Sampling-based motion planning algorithms have been highly successful at solving previously unsolved problems [119], and much research has focused on developing more sophisticated variants of them. Sampling-based approaches are efficient and can be applied to problems with many degrees of freedom

(e.g., robotic manipulators with many links or proteins with many amino acids). While not guaranteed to find a solution, they are known to be probabilistically complete, meaning that the probability of finding a solution, given one exists, increases with the number of samples generated [120].

Even so, substantial resources in time and hardware are still required to solve complex applications. For example, modeling the motion of a small protein using sequential sampling-based motion planning techniques can take days on a typical desktop machine [121]. Thus, it is practically infeasible to study larger proteins or to significantly increase the detail and accuracy at which their motions are modeled. Hence, there is a need for more efficient methods and parallel processing is a natural option to explore.

Previous work [12, 122] proposed methods based on uniform workspace subdivision for parallelizing representatives of the two major classes of sampling-based motion planning algorithms. By subdividing the space and restricting the locality of graph connection attempts, inter-processor communication associated with nearest neighbor searches – a well-known bottleneck in parallelizing sampling-based motion planning algorithms [123, 124, 125, 126] – can be substantially reduced. This approach achieves better and more scalable performance on different parallel machines than previous methods. Fundamentally, uniform spatial subdivision methods are limited in the types of motion planning problems they can solve efficiently. In particular, for most non-trivial environments, as the problem is subdivided, the variance in the amount of work performed by the subdivisions will increase. Because of the difference in complexity of the subdivided regions, there will be a corresponding increase in load imbalance.

In this work, we apply and analyze two techniques to address the problem of load imbalance for parallel sampling-based motion planning algorithms. The first is an adaptive work stealing approach that permanently migrates both the region and the work related to it to improve the load balance. The second approach is a bulk-synchronous redistribution

technique that redistributes regions among processors to have a more balanced distribution of data. We propose a method based on the complexity of a region to approximate the amount of work per region and use it to attempt to balance work across processors, while also preserving the spatial geometry of the subdivision.

Background and Motivation.

The motion planning problem is to find a valid path (e.g., one that is collision-free and satisfies any joint limit and/or loop closure constraints) for a movable object starting from a specified start configuration to a goal configuration in an environment [119]. A single configuration is specified in terms of the movable object's d independent parameters or degrees of freedom (DOF). The set of all possible configurations (both feasible and infeasible) defines a configuration space ($\mathbb{C}_{space}$). $\mathbb{C}_{space}$ is partitioned into two sets: $\mathbb{C}_{free}$ (the set of all feasible configurations) and $\mathbb{C}_{obstacle}$ (the set of all infeasible configurations). Motion planning then becomes the problem of finding a continuous sequence of points in $\mathbb{C}_{free}$ connecting the start and goal configuration.

A complete solution of the motion planning problem is considered computationally intractable and has been shown to be PSPACE-hard with the best known upper bound being doubly exponential in the movable object's DOFs [127]. As an alternative, randomized and approximate solutions have been shown to be efficient and practical. Sampling-based methods [119] are the state-of-the-art approach to solving motion planning problems in practice. Sampling-based methods are broadly classified into two main classes: roadmap or graph-based methods such as the Probabilistic Roadmap Method (PRM) [120] and tree-based methods such as the Rapidly-exploring Random Tree (RRT) [128]. In the following, we describe previous approaches to parallelize methods from both classes.

Parallelizing PRM with Uniform Subdivision.

The Probabilistic Roadmap Method (PRM) [120] constructs a graph $G = (V, E)$, called a roadmap, to capture the connectivity of $\mathbb{C}_{free}$. A node in G represents a valid con-

figuration and an edge represents a valid trajectory (path) between configurations. Nodes are generated using some sampling strategy and connections are attempted between a node and its k-nearest neighbors as computed using some distance metric. Once the roadmap is constructed, query processing is done by connecting the start and goal configurations to the roadmap and extracting a path through the roadmap that connects them.

A uniform $\mathbb{C}_{space}$ subdivision method for parallelizing PRM was presented in [12]. In that work, the $\mathbb{C}_{space}$ representing the movable object is subdivided into regions. For example, in three-dimensional environments, the planning space may be subdivided into regions using the $\mathbb{C}_{space}$ positional degrees of freedom, i.e., the x, y and z dimensions. A simple illustration of a 2D environment subdivided into four regions is shown in Figure 3.8(a). The subdivision is represented by a *region graph*, whose vertices represent regions and whose edges encode the adjacency information between regions. Figure 3.8(b) shows the region graph corresponding to the subdivision shown in Figure 3.8(a).

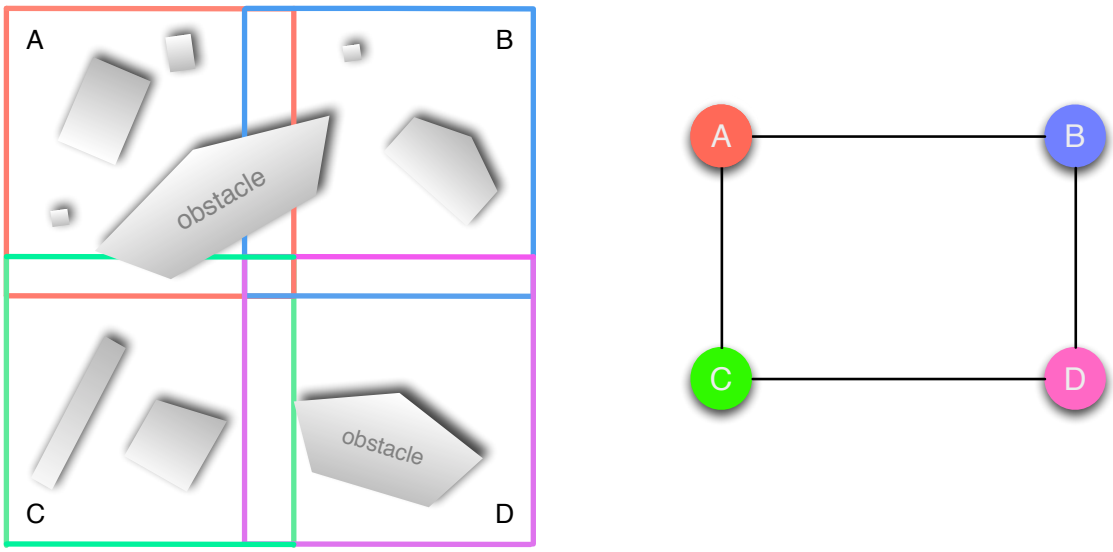


Figure 3.8: A 2D environment subdivided into 4 regions and the corresponding region graph. Reprinted with permission from [7].

Roadmaps are constructed in parallel in each region. This is done by invoking the (sequential) PRM planner [120] in each region. Lastly, the regional roadmaps are connected to form a roadmap of the entire $\mathbb{C}_{free}$. The region graph facilitates the process for connecting regional roadmaps by identifying adjacent regions between which connections are attempted. Some user-defined overlap is allowed between regions to allow sampling in the portion of space at the boundaries that may facilitate connections between regional roadmaps.

Parallelizing RRT with Uniform Radial Subdivision. The Rapidly-exploring Random Tree (RRT) method is another sampling-based motion planning approach particularly well suited for non-holonomic and kinodynamic motion planning problems [128]. The basic sequential RRT grows a tree rooted at the start configuration that expands outward into unexplored areas of $\mathbb{C}_{space}$. To build a tree, RRT first generates a uniform random sample q_{rand}, and identifies the closest node q_{near} in the tree to q_{rand}, and then q_{near} is "extended" toward q_{rand} for a step size of at most Δq. If successful, q_{new} is added to the tree as a node and the pair (q_{near}, q_{new}) is added as an edge. To solve a particular motion planning query, RRT repeats this process until the goal configuration can be connected to the tree.

Uniform radial subdivision [122] is particularly suited for parallelizing RRTs. In their method, $\mathbb{C}_{space}$ is subdivided into conical regions and subtrees are built in each region using the sequential RRT planners. Similar to uniform subdivision described earlier, regional subtrees are later connected to subtrees in neighboring regions. A 2D illustration of radial subdivision of $\mathbb{C}_{space}$ is shown in Figure 3.9.

First, a hypersphere is created in d-dimensional $\mathbb{C}_{space}$ S^d centered at $q_{root} \in R^d$ with radius r. Next, N_r points $q_i \in R^d$ are sampled on the surface of S^d. Each point defines a conical region centered around the ray $\overrightarrow{q_{root}q_i}$. A region graph $G(V, E)$ is then constructed. Each vertex $v_i \in V$ represents a region defined by q_i and an edge (v_i, v_j) is added if q_j is in the k closest neighbors of q_i. After region graph construction, independently (in parallel) a

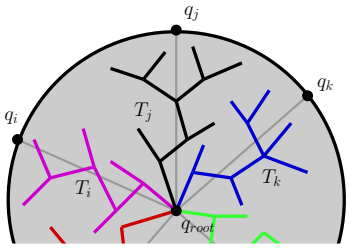


Figure 3.9: Example of uniform radial subdivision for a 2D $\mathbb{C}_{space}$. Each process concurrently builds a branch (using sequential RRT) rooted at q_{root} and biased toward a target (e.g., q_k for the blue process). Reprinted with permission from [7].

sequential RRT is used in each region. The subtree growth in each region is biased toward the region candidate defined by the random ray $\overrightarrow{q_{root}q_i}$. Some overlap between regions is allowed so branches can explore part of the space in adjacent regions. Lastly, using the adjacency information provided by the region graph, connection attempts are made between each region branch and the branches in adjacent regions. If any edge connection creates a cycle, the tree is pruned so as to remove the cycle.

3.4.2 Load Balancing For Parallel Motion Planning

Work stealing [129, 130] is an important technique used to balance an imbalanced computation. In this method the computation is logically divided into a collection of tasks. When a processing element runs out of its local tasks, it attempts to steal tasks from potential victims. This strategy is well suited for shared-memory systems. In distributed-memory systems, there are two variations on the way data can be made available to the thief: replication and ownership transfer. In the case of replication, some sort of software coherence mechanism may be required to deal with the multiple copies of data. In the case of ownership transfer, the overheads associated with transferring ownership to the thief processor need to be managed. In this work, we have a model in which transfer of ownership is considered.

Repartitioning of the data is another strategy to address load imbalance. It is well known that data distribution is fundamental to achieving acceptable levels of load balance. There exists a large body of literature regarding partitioning of distributed data structures [36, 131]. We focus on computing, and enforcing through data migration, high quality partitions of the problem across processing elements.

In general, the type of load balancing technique applied to an imbalanced computation depends on the nature of the computation itself. Repartitioning is well suited for applications in which a good estimate of the computation associated with the data can be easily computed and the total amount and structure of the computation is known *a priori*. In contrast, work stealing is best suited for dynamic applications in which either the execution of the algorithm defines more computation as the algorithm progresses, or the work associated with the task cannot be easily estimated.

Uniform (radial) subdivision is limited in the types of motion planning environments it can handle. It performs well in uniform and homogeneous environments, but not so well in non-uniform and heterogeneous environments. For example, a house or factory floor is typically composed of logically separate parts; open or free space, cluttered space, doorways, etc. Uniform subdivision in this scenario is prone to load imbalance. As an illustration, consider the uniform subdivision of the planning space in Figure 3.10(a); if different processors are assigned to each region, processors assigned to region R_0 are relatively overloaded. This irregularity in planning space leads to workload imbalance, which will have an overall negative affect on scalability.

One important consideration is the granularity in which the problem is partitioned, as the size of the biggest quanta of work establishes a lower bound by which the problem can be balanced using any load balancing strategy. In addition, a more refined problem provides more opportunity to distribute work amongst processing elements. For parallel motion planning, regions represent the quanta of work and thus for the presented load

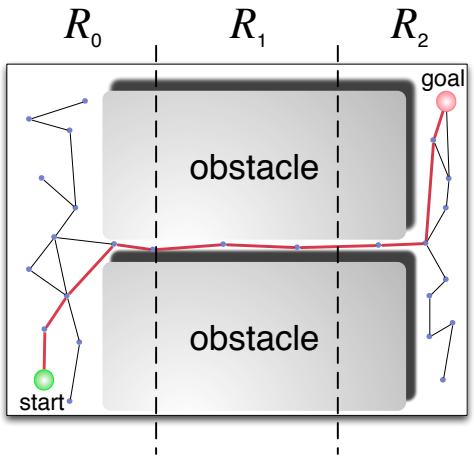


Figure 3.10: An environment where a regular partition would result in load imbalance. Reprinted with permission from [7].

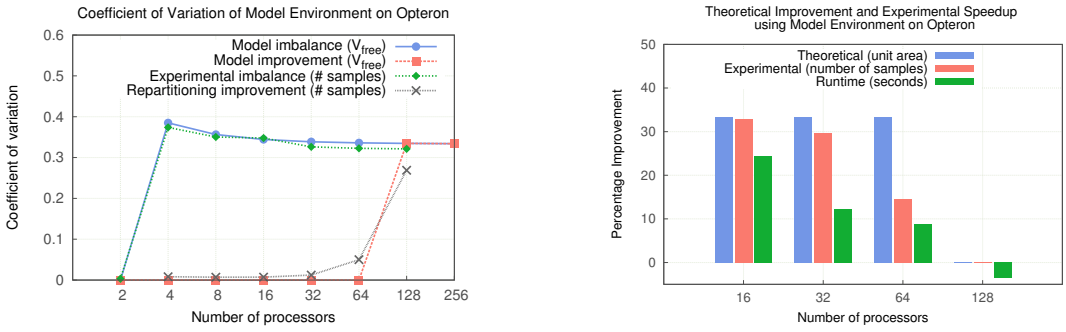
balancing strategies, we consider an over-partitioned region graph.

In general, a load balancing strategy using repartitioning requires a reasonable estimate for the amount of effort that is required to compute a quanta of work. In section 3.4.2, we will discuss weighting techniques for the two discussed parallel motion planning algorithms and the difficulty of estimating work for uniform radial subdivision RRT.

Repartitioning for Parallel Sampling-Based Planning

The effectiveness of repartitioning is highly dependent on the ability to estimate the load of an RRT or PRM region.

PRM. For uniform subdivision PRM, since we can easily and cheaply compute a cost metric of the amount of work to be done, this algorithm is a good candidate to use repartitioning. In parallelizing PRM, the two data structures of interest are the graph representation of the regions and the PRM roadmap. The regions represent a spatial subdivision of the environment in which configurations will be sampled. Connections are attempted between configurations through the use of local planning methods. It is well known in motion planning that the cost of connecting samples in $\mathbb{C}_{space}$ is highly representative of



(a) (b)

Figure 3.11: Experimental validation of (a) measure of load imbalance and (b) potential improvement in model environment. Reprinted with permission from [7].

the amount of time the overall algorithm will take in generating a solution [121]. This in fact is the most time consuming phase of the entire computation. As regions that have a high number of samples will generally incur a large number of local planning calls, a good metric for approximating the amount of work that a region will generate is the number of samples in the roadmap that lie within that region.

Using this information, we can determine that load imbalance in terms of regions corresponds to the number of roadmap samples of the region, and this metric can be used to weight regions. A high quality partition of the region graph will attempt to balance the regions based on this metric. However, as regions are also spatial entities, the spatial geometry of regions should also be preserved in an ideal partition. By partitioning the region graph using these approximations of the amount of work that a region will perform, the algorithm will see a higher level of load balance for subsequent phases of computation.

RRT. Unfortunately, in uniform radial subdivision RRT, the amount of work a region will perform is difficult to estimate beforehand. In our experiments, we show an estimate of work for an RRT branch that uses k random rays originating from the origin of the region, and computes the minimum distance to an obstacle in the direction of these rays.

Intuitively, this should give a reasonable approximation of the amount of *reachable* free space in that region; however, we show that this metric is a poor indicator of work for a given region unless a large number of rays is utilized, making this an expensive operation to calculate.

3.4.3 Experimental Evaluation

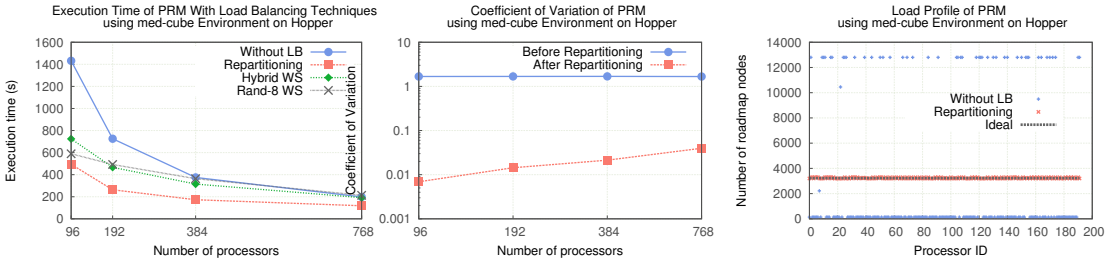
We implement and evaluate standard load balancing techniques for parallel motion planning and show that with an appropriate estimate for the amount of work in a region, geometric repartitioning outperforms work stealing. In the dynamic case where load is unknown *a priori*, repartitioning will be at a disadvantage and can potentially be worse than performing no load balancing at all.

Experimental studies were conducted on two massively parallel machines: a 153,216 core Cray XE6 (HOPPER) and a 2,400 core Opteron cluster (OPTERON-CLUSTER). Unless otherwise noted, all experiments show strong scaling in which the total number of regions is kept constant as the number of processing elements increases.

3.4.4 Implementation in STAPL Graph Library

Load imbalance in parallel computations is dealt with in various ways in STAPL. For repartitioning, this is realized through redistribution of the two `pGraphs` [101] (i.e., the region graph and the roadmap or RRT graph) in the parallel motion planning algorithms. Alternatively, load balancing can be addressed by using a custom work stealing scheduler for parallel motion planning algorithms.

In its most basic form, an application can be instrumented to perform repartitioning by simply providing a view of the container to migrate, and weights of the individual elements of the container. Additionally, a user-defined function can be provided that will define actions that need to be taken upon a migration, such as additional migrations of secondary data structures. Internally, the data structure will be redistributed using various techniques,



(a) (b) (c)

Figure 3.12: Evaluation of (a) execution time and (b) coefficient of variation and (c) load distribution for PRM on HOPPER using med-cube. Reprinted with permission from [7].

including STAPL algorithms that diffusively move work to neighbors and attempt to minimize edge cuts and by extension preserve geometric features of the graph, or those that globally balance weight in blocks. Alternatively, the STAPL Load Balancing Framework can also be used interoperably with external partitioning libraries, such as Zoltan [132].

By invoking repartitioning techniques after node generation to help balance computation for the node connection phase for parallel motion planning, we show experimentally that we achieve good load balance for the most computationally expensive phase of the algorithm.

3.4.4.1 Model PRM Environment

Consider a 2D environment with a single square obstacle that lies equidistant from the bounding box. It is possible to compute the volume of the free space (V_{free}) by using the total volume of the region and the volume of the obstacle within the region. With an estimate of the free space in the environment, we can say that the total load that the region will experience is proportional to V_{free}.

One measure of imbalance among processors is the *coefficient of variation*, defined to be the ratio of the standard deviation σ and mean μ load. A naïve mapping of regions to processors would perform a 1D partitioning of the region mesh and assign a balanced

number of region columns to processors. This naïve region mapping will have a high coefficient of variation for the model environment. We find an estimate of the most balanced partitioning of the region graph statically ignoring edge-cuts using a greedy global partitioning algorithm, as the exact problem is NP-complete.

Figure 3.11(a) shows the model's prediction of the imbalance in terms of the coefficient of variation of samples (lower is better) with the naïve partitioning strategy and the best load balance possible. In addition, we plot the measure of load imbalance experienced during a trial run of the algorithm using repartitioning and show that we closely track the model. As shown, the best possible distribution of regions to processors for higher core counts shows less benefit, as each processor has an increasingly smaller granularity of work as the number of processors increases.

Figure 3.11(b) studies various metrics according to the model and an experimental evaluation. We study the potential improvement according to the model (theoretical), which measures the total reduction in V_{free} for the processor with the highest amount of V_{free}. Next, we plot the reduction in the number of roadmap nodes (experimental) on the highest loaded processor. Finally, we show the overall improvement in execution time (runtime) for the load-balanced phase using repartitioning. In general, we track the model's theoretical estimate of the best load distribution in terms of roadmap nodes, which in turn tracks the improvement in execution time. The discrepancies between the best distribution of V_{free} and the roadmap node distribution can be explained by both the probabilistic nature of the computation and by the geometric restrictions enforced by the repartitioning. The gap between the improvement in roadmap distribution and total time reduction is a result of the number of roadmap nodes per region being an imperfect indicator of the total amount of work generated by that region. At 128 cores, there is no better distribution of load possible, so the experimental result only shows the overhead of attempting to repartition.

3.4.4.2 Experimental Results

PRM. The environments considered in this section are variants of a 3D narrow passage with a rigid-body robot, similar to the theoretical environment that consists of a single cubic obstacle in which roughly 24% (med-cube), 6% (small-cube) and 0% (free) of the environment is blocked. In all environments, we subdivide the problem into 250,000 regions total. Figure 3.12(a) shows raw execution time for computing the final roadmap on the HOPPER platform for this strong scaling PRM experiment in the med-cube environment. We can see that using repartitioning, we are able to achieve a 2.9x improvement over the baseline on 96 cores and a 1.68x improvement on 768 cores. Because of the strong scaling nature of our experiment, there are significantly fewer regions per processor at 768 cores, which allows for less opportunity for moving load across processors. From Figure 3.12(b), we can see that although the coefficient of variation is substantially lower for all processor counts after repartitioning, the difference is not as much for higher processors counts simply because of a reduced opportunity to rebalance. Figure 3.12(c) shows the distribution of load across processors on a 192-core run on HOPPER. We see that without load balancing, there is a wide spread in work and after applying repartitioning, a distribution closer to the ideal is achieved. Figure 3.13 shows the trend shown in the previous analysis holds for higher processor counts on HOPPER.

For the same experiment, we show the breakdown of the various phases of parallel PRM in Figure 3.14(a). As suspected, the portion of the computation connecting roadmap nodes in a region dominates most of the computation at 90% of the total execution time. After load balancing for both methods, the total time decreases, mainly because of the decrease in node connection time. For repartitioning, there is an increase in region connection time, which can be partially attributed to an increase in remote accesses in the region connection phase, as shown in Figure 3.14(b). This is due to an increase in edge

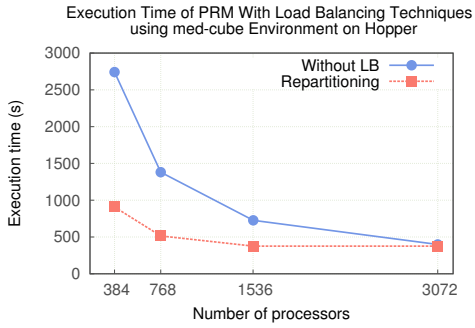


Figure 3.13: Evaluation of computing roadmap in the med-cube environment for a rigid body robot on Hopper. Reprinted with permission from [7].

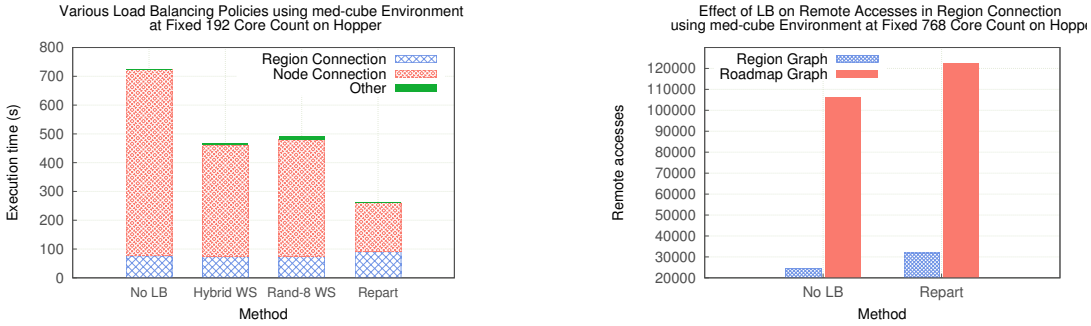
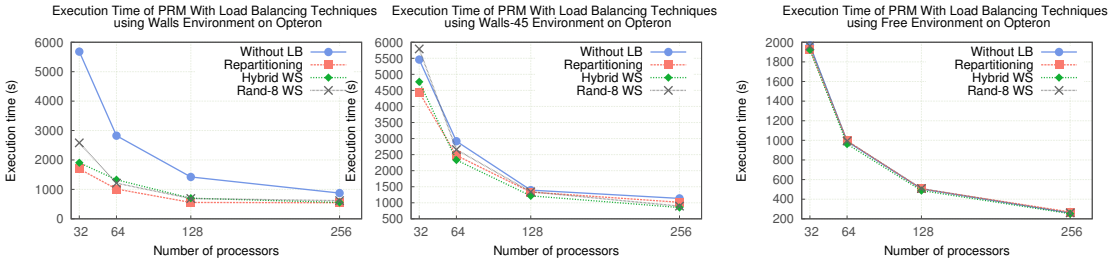


Figure 3.14: Breakdown of (a) the various phases of PRM (b) and the effect of load balancing on remote accesses. Reprinted with permission from [7].

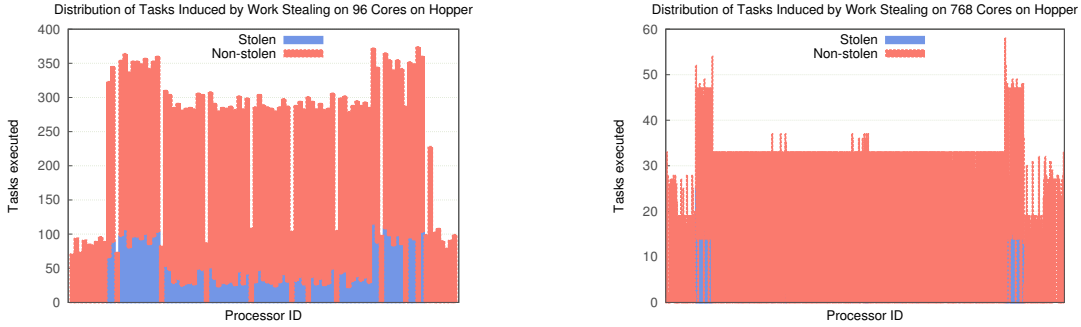
cuts, which was induced by repartitioning.

Figure 3.15 demonstrates load balancing techniques on multiple environments that display different levels of imbalance on OPTERON-CLUSTER. In Figure 3.15(a) and (b), we see up to a 3.4x speedup over the baseline using repartitioning in the med-cube environment, but only a 1.2x speedup in the small-cube environment. This shows that even on workloads that are not imbalanced to such a high degree, load balancing can still provide a large benefit. In addition, we find that in the free environment which exhibits no imbalance, all load balancing techniques do not show any significant overhead over the non-load



(a) (b) (c)

Figure 3.15: Execution time for PRM with various load balancing strategies in (a) med-cube (b) small-cube (c) and free environment. Reprinted with permission from [7].



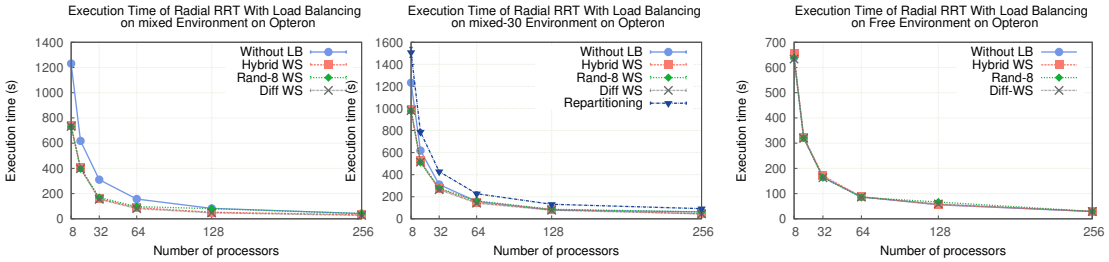
(a) (b)

Figure 3.16: Breakdown of the amount of tasks stolen vs. executed locally for PRM on (a) 96 and (b) 768 cores on HOPPER. Reprinted with permission from [7].

balanced variant, as shown in Figure 3.15(c).

RRT. We also evaluated load balancing techniques on the uniform radial RRT parallel motion planning algorithm. As discussed in Section 3.4.2, it is difficult to estimate the amount of work that a radial branch will compute due to the probabilistic and dynamic nature of the algorithm. Thus, computing an effective partition for load balancing is difficult and may prove to be inaccurate.

In Figure 3.17(b), we computed an estimate of work for an RRT branch by using the k random rays metric discussed in Section 3.4.2. This metric should intuitively estimate



(a) (b) (c)

Figure 3.17: Execution time for RRT with various load balancing strategies in (a) mixed (b) mixed-30 (c) and free environment. Reprinted with permission from [7].

the number of local planning calls in the RRT construction, yet it acts as a poor estimate for the amount of work needed to compute the RRT branch and in most cases, we see a slowdown when using this weight. Indeed, any metric to estimate the amount of work in this random and dynamic algorithm would likely be imprecise and it is for this reason work stealing strategies are better suited for uniform subdivision radial RRT.

RRT. We evaluated work stealing on the radial RRT parallel motion planning algorithm. Figure 3.17 shows the total execution time for computing the final RRT for a rigid body robot in two cluttered environments and one free environment on OPTERON-CLUSTER. We varied the amount of free space in each environment such that the first environment (mixed) is 60% blocked, the second environment (mixed-30) is 30% blocked and the third environment (free) is completely free of obstacles (0% blocked). Using the DIFFUSIVE work stealing strategy allowed the algorithm to achieve a speedup of 2.0x on 32 cores and 1.55x at 256 cores in the mixed environment. A similar pattern of decreasing marginal benefit of work stealing from uniform subdivision is exhibited in this experiment. As with uniform subdivision, the stealable work per processor decreases with the number of processors, while the number of potential victims from which to steal also increases.

We find that all three work stealing strategies show similar improvements in execution

time, as the problem is over-decomposed to such a degree that underloaded processors have a high probability of finding work, regardless of the victim selection. As expected, work stealing shows a larger improvement in the mixed environment vs. the mixed-30 environment, as the reduction in execution time is a function of the amount of imbalance. Similar to the PRM experiments in which we measure load balancing overheads, we find that in the free environment, we do not see significant differences in the load-balanced execution vs. the baseline.

4. APPROXIMATE GRAPH COMPUTING[1]

In this chapter, we introduce a novel approximate parallel breadth-first search algorithm [8] based on the k-level asynchronous [6] (KLA) paradigm. The KLA paradigm bridges level-synchronous processing [3] (based on the bulk-synchronous parallel model [19]) and asynchronous processing [112], allowing for parametric control of the amount of asynchrony from full (asynchronous) to none (level-synchronous). In a level-synchronous execution of breadth-first search, distances are correct at the end of a level, at the cost of expensive global synchronizations. On the other hand, a high amount of asynchrony in breadth-first search may lead to redundant work, as the lack of a global ordering could cause a vertex to receive many updates with smaller distances until the true breadth-first distance is discovered. Each update to the vertex's state will trigger a propagation of its new distance to its neighbors, potentially leading to all reachable vertices being reprocessed many times and negating the benefit of asynchronous processing.

Our novel algorithm controls the amount of redundant work performed by controlling how updates trigger propagation and allowing for vertices to contain some amount of error. In short, by only sending the improved values to neighbors if the change is large enough, we limit the amount of redundant work that occurs during execution. We modify the KLA breadth-first search algorithm by conditionally propagating improved values received from a neighbor update.

The contributions of this work include:

- **Approximate k-level asynchronous breadth-first search algorithm.**

  We present a new algorithm for approximate breadth-first search that trades accuracy

[1]Reprinted with permission from *"Fast Approximate Distance Queries in Unweighted Graphs Using Bounded Asynchrony"* by Fidel A., Sabido F.C., Riedel C., Amato N.M., Rauchwerger L., 2017. Lecture Notes in Computer Science, vol 10136, Copyright 2017 by Springer International Publishing Switzerland.

for performance in a KLA BFS. We prove an upper bound on the error as a function of degree of approximation.

- **Implementation that achieves scalable performance.** Our implementation in the STAPL Graph Library shows an improvement of up to 2.27x over the exact KLA algorithm and 3.8x improvement over the level-synchronous version with minimal error. Results show that our technique is able to scale up to 32,768 cores.

This chapter is reprinted with permission from [8].

4.1 Approximate Breadth-First Search

Our algorithm is implemented in the k-level asynchronous paradigm. In KLA, algorithms are expressed using two operators. The vertex operator is a fine-grained function executed on a single vertex that updates the vertex's state and issues visitations to neighboring vertices. It may spawn visitations through the use of `Visit(u, op)` or `VisitAllNeighbors(v, op)`, where `u` is the ID of a single neighbor and `v` is the ID of the vertex being processed. These visitations are encapsulated in the neighbor operator, which updates a vertex based on values received from a single neighbor.

In the exact KLA breadth-first search, skipping the application of the neighbor operator could lead to an incorrect result, but reduces the performance overhead of redundant work that is often seen in highly asynchronous algorithms. We show that the amount of error can be bounded, while improving the performance of the distance query.

4.1.1 Algorithmic Description

In this section, we show how to express approximate breadth-first search using the KLA paradigm. The goal is to compute, for each vertex, the distance from the vertex and the root in the breadth-first search tree. We denote this distance as $d(v)$.

Initially, all vertices except the source have distance $d_k(v) = \infty$, no parent, and color

```
1 Function VertexOperator(v)
2     if v.color = GREY then
3         v.color = BLACK
4         VisitAllNeighbors(v, NeighborOp, v.dist+1, v.id)
5         return true
6     else
7         return false
8     end
```

Algorithm 3: k-level asynchronous BFS vertex operator.

```
1 Function NeighborOperator(u, dist, parent)
2     if u.dist > dist then
3         u.dist ← dist
4         u.parent ← parent
5         u.color ← GREY
6         return true
7     else
8         return false
9     end
```

Algorithm 4: Original k-level asynchronous BFS neighbor operator.

set to black. The source vertex sets its distance to 0, itself as its parent and marks itself active by setting its color to grey. Algorithm 3 shows the vertex operator that is executed on all vertices in parallel. Each vertex determines if it is active by checking if its color is set to grey. If so, it issues visitations to all of its neighbors, sending its distance plus one. The traversal is completed if all invocations of the vertex operator return false in a superstep (i.e., none of the vertices are active).

Algorithm 4 presents the neighbor operator for the exact breadth-first search algorithm. The distance and parent are updated if the incoming distance is less than the vertex's current distance. In addition, the vertex sets its color to grey, marking it as active, and returns a flag indicating that it should be revisited. In the k-level async model, if the

invocation of the neighbor operator returns true, the vertex operator will be reinvoked on that vertex only if its hop-count is still in bounds of the KLA superstep. That is, if $d(v)$ mod $k = 0$, then the visitation is at the edge of the superstep and thus the vertex operator will not be invoked until the start of the next superstep.

```
1  Function ApproximateNeighborOperatorTolerance(u, dist, parent)
2      if u.dist > dist then
3          u.dist = dist
4          first_time ← u.parent = none
5          better ← (u.prop - dist)/u.prop ≥ τ
6          if first_time ∨ better then
7              u.parent ← parent
8              u.prop ← dist
9              u.color ← GREY
10             return true
11         end
12     else
13         return false
14     end
```

Algorithm 5: Approximate k-level asynchronous BFS with tolerance neighbor operator.

In this work, we introduce a new neighbor operator in Algorithm 5 that allows for the correction of an error and repropagation of the corrected distance under certain conditions. We use tolerance $0 \leq \tau < 1$ to denote the amount of error a vertex will allow until it propagates a smaller distance. For a visit with current distance d and better distance d_{new}, we will propagate the new distance if $(d - d_{new})/d \geq \tau$. We now need to store two distances: one that represents the current smallest distance seen and the distance of the last propagation. The last propagated distance is required as a vertex may continually improve its own distance, but it will only repropagate if a neighbor visitation contains a distance that is τ-better than its last propagated distance. By following a vertex's parent property,

the algorithm also provides a path from every reachable vertex to the source, similar to the traditional version of breadth-first search. However, these vertices may report a larger distance than the length of the discovered path, due to updates that were not propagated.

The parameter τ controls the amount of tolerated error. Note that if $\tau < 1/|V|$, then there is no error in the result and the neighbor operator is equivalent to the exact version in Algorithm 4.

4.1.2 Error Bounds

As the approximate breadth-first search may introduce error, we quantify the error that may be caused due to asynchronous visitations. We denote the breadth-first distance of a vertex v at the end of a KLA traversal using $d_k(v)$, where k is the level of asynchrony. Similarly, $d_0(v)$ is the true breadth-first distance for vertex v. In this section, we will show that the error of the breadth-first distance is bounded by $d_k(v) \leq d_0(v)k$.

Lemma 4.1.1. *At the end of the first KLA superstep, all reached vertices have distance* $d_k(v) \leq k$.

Proof. Assume at the end of the first superstep, there exists a vertex v with distance $d_k(v) > k$. This means that v was reached on a path from the source that has $h > k$ hops. This is not possible, as the traversal will not allow a visitation that is more than k hops away. Therefore $d_k(v) \leq k$. □

Theorem 4.1.2. *At the end of the algorithm, all reachable vertices will have distance* $d_k(v) \leq ks_v$*, where* s_v *is the superstep in which* v *was discovered.*

Proof. Assume that after superstep s all reached vertices will have distance $d_k(v) \leq sk$. Lemma 4.1.1 shows this holds for $s = 1$. All active vertices will issue visitations to their neighbors, traveling up to at most k hops in superstep $s + 1$. Consider a previously unreached vertex u that will be discovered in superstep $s + 1$ from some vertex w that was

discovered in superstep s. Vertex w was on the boundary of superstep s and has distance at most sk from the source. Therefore, $d_k(u) \leq d_k(w) + k$ because u will be discovered from a path that is up to k hops from w.

$$\begin{aligned} d_k(u) &\leq d_k(w) + k \\ &\leq sk + k && \text{(inductive hypothesis)} \\ &\leq (s+1)k && \text{(simplification)} \end{aligned}$$

Through induction, $d_k(u) \leq sk$ for a vertex u discovered in superstep s. □

Lemma 4.1.3. *If there exists a path π from the source to a vertex v, then v must be discovered no later than superstep $|\pi|$.*

Proof. We will show the lemma holds by induction. If the length of path π is 1, vertex v shares an edge with the source. Then in the first superstep, the source will visit all edges and discover v.

Suppose the lemma holds for any path with length i. Let π be a path with length $|\pi| = i + 1$. Then the i^{th} vertex along the path, v_i, will have been discovered in or before the i^{th} superstep. Now, by Algorithm 3, the vertex v_i will traverse all of its outgoing edges in or before the $(i+1)^{th}$ superstep and discover the $(i+1)^{th}$ vertex along the path π. This proves the lemma holds for any path π of length $i + 1$. Therefore, the lemma holds for any path π by induction. □

Lemma 4.1.4. *If there exists a path from the source to a vertex v, then v will be discovered at the latest in superstep $d_0(v)$.*

Proof. If a vertex has distance $d_0(v)$, then the shortest path π^* to v has length $|\pi^*| = d_0(v)$. By Lemma 4.1.3, this path must be discovered at the latest in superstep $d_0(v)$. □

Theorem 4.1.5. *At the end of the algorithm, all reachable vertices will have distance* $d_k(v) \leq d_0(v)k$.

Proof. By Theorem 1, $d_k(v) \leq s_v k$. We know through Lemma 4.1.4 that v will be visited by superstep $d_0(v)k$. Combining these, the approximation of the true breadth-first distance is off by at most a multiplicative factor of k: $d_0(v)k$. □

4.1.3 Bounds with Tolerance

When using the tolerance heuristic, a vertex with distance d will only propagate a new distance d_{new} if the following is true:

$$\frac{d - d_{new}}{d} \geq \tau \tag{4.1}$$

In the exact k-level asynchronous algorithm, all vertices that are distance $d_0(v)$ away from the source will be visited in superstep $\frac{d_0(v)}{k}$. However, since we allow some bounded error, it is possible for a vertex to be visited in the $\frac{d_k(v)}{k}$ superstep, which may be later than its original visitation. In addition, all edges that are traversed through visitations will be visited in the same superstep in which the visit was issued. However, not all visitations trigger a propagation of a better distance to the vertex's neighbors.

We will denote the discovered distance of a vertex using the tolerance heuristic as $d^{\tau}(v)$. In this section, we will prove that by using this heuristic, if a vertex v is reached at the end of the first superstep, then $d^{\tau}(v) \leq \frac{\sum_{j=0}^{d_0(v)-1}(1-\tau)^j}{(1-\tau)^{d_0(v)}}$.

Lemma 4.1.6. *All vertices with a true distance of 1 will propagate a distance that is at most* $\frac{1}{1-\tau}$.

Proof. Because the distance from the source to v is 1, the shortest path $\pi^* =< (src, v) >$ will be processed eventually in the traversal. Consider that vertex v is discovered along a path π from the source and marks itself as distance $|\pi|$. Once the path π^* is processed, v

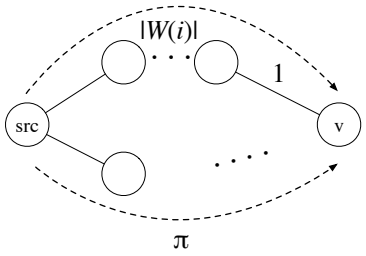


Figure 4.1: Example graph showing two different paths from the source to a vertex v. Reprinted with permission from [8].

will not propagate its distance if $\frac{|\pi|-1}{|\pi|} < \tau$. Simplifying, the length of the path is $|\pi| < \frac{1}{1-\tau}$. Therefore, v will propagate a distance that is at most $\frac{1}{1-\tau}$, otherwise a repropagation will be triggered. □

Theorem 4.1.7. *At the end of the first superstep, all reachable vertices will propagate a distance at most* $\frac{\sum_{j=0}^{d_0(v)-1}(1-\tau)^j}{(1-\tau)^{d_0(v)}}$.

Proof. Let $W(i) = \frac{\sum_{j=0}^{i-1}(1-\tau)^j}{(1-\tau)^i}$ denote the length of the longest path that will be tolerated by a vertex of true distance i without triggering a propagation. Lemma 4.1.6 shows that this holds for vertices with true distance 1. Assume that this property holds for vertices of distance i.

Let v be a vertex with true distance $i+1$ discovered along some path π. By definition, v will not repropagate upon seeing a path π_{new} if the following holds:

$$\frac{|\pi| - |\pi_{new}|}{|\pi|} < \tau \tag{4.2}$$

The shortest path π_{new} that could be discovered without repropagating could have length $|\pi_{new}| = W(i) + 1$. Any path longer than π_{new} would have triggered a repropagation along the path, by definition of $W(i)$. See Figure 4.1 for an example.

The vertex will not propagate the better distance if the threshold is not met:

$$\frac{|\pi| - (\frac{\sum_{j=0}^{i-1}(1-\tau)^j}{(1-\tau)^i} + 1)}{|\pi|} < \tau \tag{4.3}$$

Written in terms of $|\pi|$, this can be simplified:

$$\begin{aligned}
|\pi| &< \frac{\frac{\sum_{j=0}^{i-1}(1-\tau)^j}{(1-\tau)^i} + 1}{1-\tau} \\
&= \frac{\sum_{j=0}^{i-1}(1-\tau)^j}{(1-\tau)^{i+1}} + \frac{(1-\tau)^i}{(1-\tau)^{i+1}} \\
&= \frac{\sum_{j=0}^{i}(1-\tau)^j}{(1-\tau)^{i+1}} \\
&= W(i+1) && \text{(definition of } W(i)\text{)}
\end{aligned}$$

The bound therefore holds for vertices with true distance $i+1$ and thus all vertices by induction. □

As shown in Algorithm 5, a vertex always updates its distance upon seeing a better distance, without necessarily propagating it. This means that a vertex's discovered distance is at most its propagated distance. That is, all vertices discovered in the first superstep will have distance at most $d^\tau(v) \leq \frac{\sum_{j=0}^{d_0(v)-1}(1-\tau)^j}{(1-\tau)^{d_0(v)}}$.

Note that in the case of $\tau = 0$, $d^\tau(v) = \sum_{j=0}^{d_0(v)-1} 1/1 = d_0(v)$. Therefore, $\tau = 0$ is equivalent to the exact algorithm.

4.1.4 Combined Bounds

By the definition of KLA, the maximum distance that any vertex can be assigned in the first superstep is k. Therefore, for a vertex of true distance i, its discovered distance can be at most k. $W(i)$ is the length of the longest path that can be tolerated by a vertex of

true distance i without propagation. However, if this path is longer than k, then it will not be visited and thus the worst case distance will be less than $W(i)$. Now, solving $W(i) = k$ for i only considering $\tau > 0$ because, as shown above, there is no error for $\tau = 0$, we find:

$$
\begin{aligned}
k = W(i) &= \frac{\sum_{j=0}^{i-1}(1-\tau)^j}{(1-\tau)^i} \\
&= \frac{\frac{1-(1-\tau)^i}{1-(1-\tau)}}{(1-\tau)^i} && \text{(Partial geometric sum, where } 1-\tau > 0\text{)} \\
k\tau &= \frac{1-(1-\tau)^i}{(1-\tau)^i} \\
k\tau + 1 &= \frac{1}{(1-\tau)^i} \\
i &= \log(\frac{1}{k\tau+1}) / \log(1-\tau)
\end{aligned}
$$

If a vertex v has at most true distance i, then its discovered distance is bounded by $W(i)$. However, if the true distance is greater than $\log(\frac{1}{k\tau+1}) / \log(1-\tau)$, then the vertex's discovered distance can be no more than k, because the path that causes the bound of $W(i)$ is no longer reachable in k hops.

Therefore, if a vertex v is reached in the first superstep, the maximum distance $d_k^\tau(v)$ that v can have is:

$$
d_k^\tau(v) \leq \begin{cases} d_0(v) & \tau = 0 \\ \frac{\sum_{j=0}^{d_0(v)-1}(1-\tau)^j}{(1-\tau)^{d_0(v)}} & d_0(v) \leq \log(\frac{1}{k\tau+1}) / \log(1-\tau) \\ k & otherwise \end{cases}
$$

Figure 4.2 presents the trend of this function for various values of τ and a fixed value $k = 16$. We see that $W(i)$ can grow very rapidly, but is bounded by at most k. For $\tau = 0$,

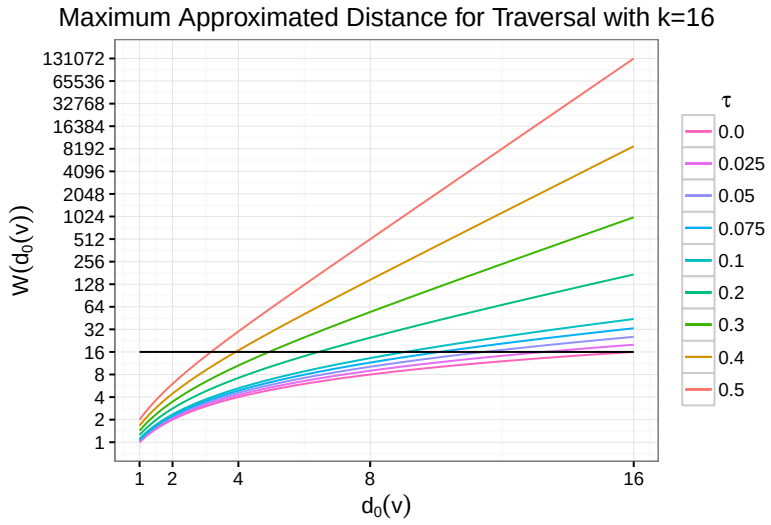


Figure 4.2: Computed distance $d_k^\tau(v)$ vs actual distance $d_0(v)$ for multiple τ and fixed k. Reprinted with permission from [8].

the approximated distance is the same as the exact distance.

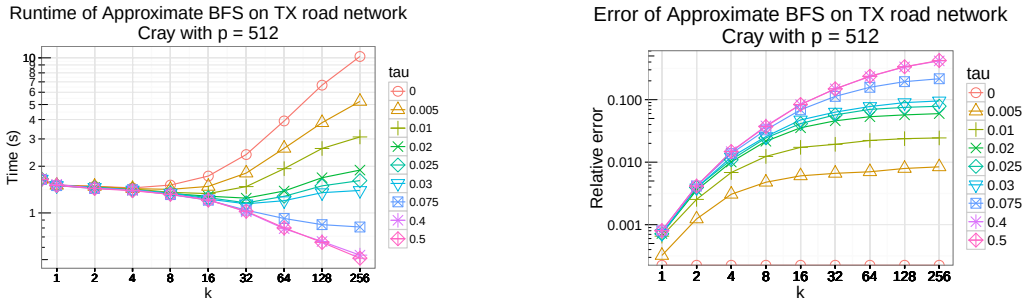
Using the same technique as Theorem 4.1.5, we can show that error will accumulate across supersteps in an additive way. Therefore, the total distance that a vertex at the end of the algorithm will have is $d_k^\tau(v) \leq d_0(v)k$.

4.2 Implementation

We implemented the approximate breadth-first traversal in the STAPL Graph Library (SGL) [101, 102, 6]. We use the k-level-asynchronous paradigm in the STAPL Graph Library to implement the approximate breadth-first search algorithm. Algorithms 1-3 are implemented as function objects in C++ whose function operators return a boolean value indicating if the traversal should continue.

4.3 Experimental Evaluation

We evaluated our technique on two different systems.



(a) (b)

Figure 4.3: Approximate BFS with tolerance heuristic on TX road network with 512 cores on Cray evaluating (a) runtime and (b) error. Reprinted with permission from [8].

CRAY-XK7**.** This is a Cray XK7m-200 system which consists of twenty-four compute nodes with AMD Opteron 6272 Interlagos 16-core processors at 2.1 GHz. Twelve of the nodes are single socket with 32 GB of memory, and the remaining twelve are dual socket nodes with 64 GB of memory.

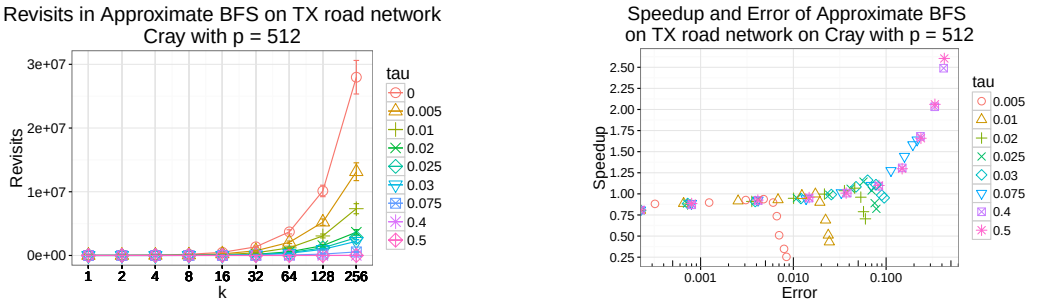
IBM-BG/Q**.** This is an IBM BG/Q system available at Lawrence Livermore National Laboratory. IBM-BG/Q has $24,576$ nodes, each node with a 16-core IBM PowerPC A2 processor clocked at 1.6 GHz and 16 GB of memory. The compiler used was `gcc` 4.8.4.

The code was compiled with maximum optimization levels (`-DNDEBUG -O3`). Each experiment has been repeated 32 times and we present the mean execution time along with a 95% confidence interval using the t-distribution. We also measure the relative error of a vertex's distance, where error is defined as $(d_k^\tau(v) - d_0(v))/d_0(v)$. We show the mean relative error across all vertices.

4.3.1 Breadth-First Search

In this section, we evaluate our algorithm on various graphs in terms of execution time and relative error.

In Figure 4.3, we evaluate both the execution time and error on the Texas road network



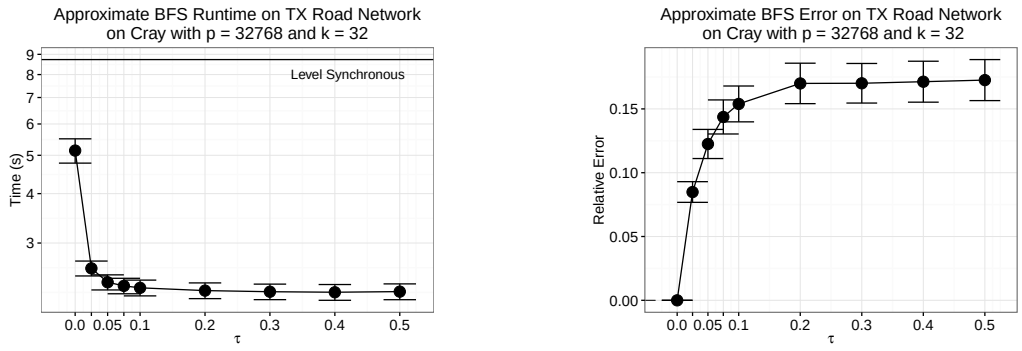
(a) (b)

Figure 4.4: Approximate BFS with tolerance heuristic on TX road network with 512 cores on CRAY-XK7 evaluating (a) number of repropagations that occur during traversal and (b) speedup over the fastest k. Reprinted with permission from [8].

from the SNAP [114] collection on 512 cores on the CRAY-XK7 platform. This graph has 1.3 million vertices and 1.9 million edges. As expected, a lower value of τ results in slower execution time as more repropagations occur with lower tolerance. In the extreme case of $\tau = 0$, every message that contains a better distance is propagated and thus it is the same as the exact version of the algorithm. Figure 4.4(a) shows the number of repropagations that occur as we vary the level of asynchrony and τ. As expected, higher values of k result in many more visitations, while higher τ triggers relatively less visitations. This behavior results in the corresponding time and error tradeoffs we observe in Figure 4.3.

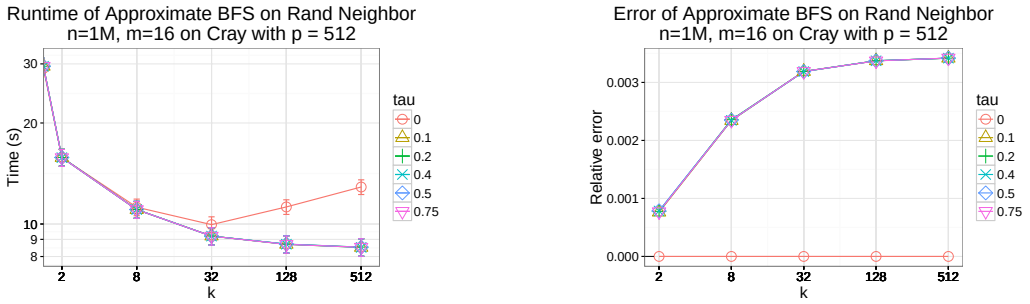
Figure 4.4(b) shows speedup vs error on the Texas road network. Speedup is defined as the ratio of the exact algorithm's execution time with the fastest k and the approximate algorithm's execution time. If an application is willing to tolerate error in the result, we see that we are able to achieve 2.6x speedup for an execution with 42% error.

Figure 4.5 shows that we see similar benefit using the road network graph on the IBM-BG/Q platform for a fixed value of k. We see that the exact version of the KLA breadth-first search ($\tau = 0$) is slower than the level synchronous version, and the approximate version is faster than both. At 32,768 cores, the approximate version is 2.27x faster with



(a) (b)

Figure 4.5: Strong scaling of approximate BFS on IBM-BG/Q platform evaluating sensitivity of (a) runtime and (b) error. Reprinted with permission from [8].



(a) (b)

Figure 4.6: Approximate BFS with tolerance heuristic on random neighborhood network ($n = 1,000,000$ and $m = 16$) with 512 cores on CRAY-XK7. Reprinted with permission from [8].

around 17% mean error.

Random Neighborhood. We next evaluate the algorithm on a deformable graph that allows us to vary the diameter from very large (circular chain) to very small (random graph). This results in graphs with different diameters by allowing any given vertex to randomly select and connect only to its $\pm m$-closest neighboring vertices. This is similar to the approach described by Watts and Strogatz [20] where the rewiring mechanism is limited in terms of distance.

Figure 4.6 shows the performance and error of an execution of this algorithm on a random neighborhood graph on 512 cores on the CRAY-XK7 platform. As shown, we see a benefit for using the approximate version for higher values of k. At a k of 512, the approximate algorithm has a 1.12x speedup over the fastest exact version but only has an error of 0.3%. Because this graph does not have as much opportunity for wasted work as the road network, the benefits of approximation are not as pronounced, but we still see an improvement in performance with negligible error.

5. NESTED PARALLEL GRAPH PROCESSING[1]

Processing real-world graphs on distributed-memory systems is a challenging endeavor due to the scale-free nature of many important graphs of interest. These graphs are known for their presence of *hub vertices* that have extremely high degrees and present challenges for parallel computations for distributed memory. In this chapter, we present a novel mechanism to decouple the distribution of the edges on a per-vertex basis for the standard adjacency-list graph representation. We show several strategies for adjacency-list partitioning and analyze the types of strategies that are most beneficial to specific graph algorithms and machine architectures. Our adjacency-list partitioning transformation is completely agnostic to the algorithm itself, preserving the traditional vertex-centric algorithmic expression for parallel graph computations. Finally, we evaluate our technique on several real-world graphs and show up to 2.25x speedup against the on 4,096 cores.

Processing large-scale graphs has increasingly become a critical component in a variety of fields, from scientific computing to social analytics. An important class of graphs are *scale-free* networks, where the vertex degree distribution follows a power-law. These graphs are known for their presence of *hub vertices* that have extremely high degrees and present challenges for parallel computations on these types of graphs.

In the presence of hub vertices, simple 1D partitioning (i.e., vertices distributed, edges stored sequentially with corresponding vertex) of scale-free networks presents challenges to keeping a balanced number of vertices and edges per processor, as the placement of a hub could overload any one processor. More sophisticated types of partitioning have been proposed, including checkerboard 2D adjacency matrix partitioning [40], edge list parti-

[1]Part of this chapter is reprinted with permission from *"Asynchronous Nested Parallelism for Dynamic Applications in Distributed Memory"* by Ioannis Papadopoulos, Nathan Thomas, Adam Fidel, Dielli Hoxha, Nancy M. Amato, Lawrence Rauchwerger, 2015. Lecture Notes in Computer Science, 9519, 106-121, Copyright 2015 by Springer International Publishing Switzerland.

tioning [41] and specialized techniques for distributing hub vertices [28, 33]. However, these strategies often change both the data representation as well as the algorithm, making it difficult to unify them in a common framework.

In this work, we propose a framework to independently control the distribution of edges on a per-vertex basis, allowing the possibility to express orthogonal strategies for the various kinds of vertices in the graph. First, we represent the graph as a distributed array of vertices, with each vertex having a (possibly) distributed array of edges, partitioned in a fine-tuned manner. Using nested parallel constructs, we can define several strategies for distributing the edges of hub vertices, that can be interchanged without changing the graph algorithm itself.

We introduce several adjacency-list partitioning strategies built on top of this framework, each with its own set of performance characteristics. The first distribution strategy, EVERYWHERE places a hub's adjacency list on all processing elements where the vertices themselves are stored. Another strategy, NEIGHBORS, places the edges only on the locations on which the vertex has a neighbor. This strategy is especially dynamic as the distribution of each hub edge list is dependent on the input data and this kind of irregular behavior is difficult to account for in static partitioning work. The last strategy, STRIPED, distributes the adjacency list on only one processor per shared-memory node in a strided fashion such that each hub is on a potentially different processor on the node than the other hubs.

Even though the distribution strategy of the edges changes dynamically, the expression of the graph algorithm itself remains unchanged. The nested parallel algorithm that executes over the edges is specified at the algorithmic level. We augment the existing k-level-asynchronous vertex-centric programming model to support execution on a graph whose edges may not be colocated with either its source or target vertex. Execution of this model of parallel graph algorithms facilitates one-sided, locality driven nested parallelism.

In addition, it allows one to easily experiment with novel and arbitrary mappings of the edges to locations, without the overhead of rewriting and hand-tuning the nested parallel algorithm to support a change in edge distribution.

Our contributions include:

- A novel mechanism to partition the edges of high-degree vertices in scale-free graphs in a user-controlled manner, while still retaining the traditional vertex-centric programming model to express parallel graph algorithms.
- A suite of pre-defined adjacency list partitioning strategies that show algorithmic-sensitive benefit for various graph algorithms, further reinforcing the need for a more modular approach in edge partitioning specification.
- Scalable performance up to 4,096 cores, showing 2.25x speedup for selected algorithms.

5.1 Adjacency List Partitioning

Our adjacency-list partitioning framework consists of two aspects: gang specification, which outlines the set of processing elements to place the edges of a vertex, and partitioning, which specifies how to partition the set of edges on those processing elements. These are separate but related concepts.

5.1.1 Gang Specification

In our model, *gangs* [133] represent subgroup support. Each gang is a set of N *locations* (a processing element and associated address space) with identifiers in the range $[0, \ldots, N-1]$ in which an SPMD task executes. For partitioning a vertex's edge list, a clear interface to easily specify the gang on which the nested container will be placed should exist. Ideally, this should be decided on a per-vertex basis during the specification of the graph itself.

The edges gang specifier concept defines a type of object that given a vertex descriptor and degree, should return a gang specifier defining on which locations the edge container should be created. It should perform this action through the standard function operator. As an example, the specifier for all locations is shown in Figure 5.1. A more sophisticated specifier to create the nested container across a single location on each shared memory node is illustrated in Figure 5.2.

```
struct all_locations_edges
{
  void operator()(C ctor, size_t id, size_t degree)
  {
    ctor(all_locations);
  }
};
```

Figure 5.1: All locations gang specifier

```
struct stripe_across_nodes
{
  void operator()(C ctor, size_t id, size_t degree)
  {
    ctor(
     location_range,
     std::view::ints(0, nodes-1) | std::view::transform([](auto i) {
       return (i * num_locs_per_node + my_loc) % num_locs;
     })
    );
  }
}
```

Figure 5.2: Striped gang specifier

5.1.2 Edge Container Partitioning

Once the gang is created for the nested edge container, how the edges will be partitioned over the given set of locations needs to be defined next. There exist two general

ways to define the partition: either by using indexed-based partitioning or value-based partitioning.

Indexed-Based Partitioning. This method of partitioning the edges allows one to specify on which location of the edge container's gang an edge should be placed based solely on the index of the edge itself. That is, a function $\pi(i)$ is defined to give the location of edge i in a given vertex's adjacency list. A suite of general partitions are provided including `balanced`, `blocked`, `cyclic` and more.

Value-Based Partitioning. In contrast to the index-based strategy where the only information given to the partitioner is the edge's index in its adjacency list, the value-based partitioning mechanism provides information about the source and target vertices and the data on the graph.

To implement the NEIGHBORS strategy where an edge is place on the same location of the target vertex, the user would define the following value-based partitioning function:

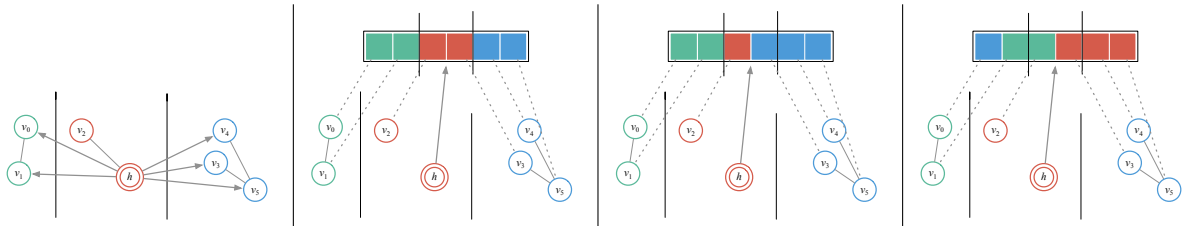
$$\pi(v, u, x) = location\_of(u) \tag{5.1}$$

This function places any given edge (u, v) on the same location of the target vertex v. For static graphs where the number of vertices are known a priori, a lookup for the location of any given vertex is an $O(1)$ operation with zero communication.

More complex partitioning functions can be defined wherein the value on the edge is also taken into account. For example, we can place any edge whose edge property is less than some user-defined threshold δ on the location of the target and place all other edges on a random location.

$$\pi(v, u, x) = \begin{cases} x < \delta & location\_of(u) \\ x \geq \delta & hash(u) \mod N \end{cases} \tag{5.2}$$

This kind of strategy effectively gives priority to edges whose properties meet some predicate. It may prove to be useful for certain classes of algorithms, such as single-source shortest path, where not every edge needs to be visited and thus those edges with higher distances could receive lower priority for being processed.



(a) Input graph (b) Everywhere (c) Neighbors (d) Random

Figure 5.3: Adjacency list partitioning strategies for an (a) input graph.

5.1.3 Distribution Strategies

Combining the gang specification and edge container partitioning results in a unified adjacency list distribution. This work introduces various adjacency list distribution strategies for distributing the edges of hub vertices, that can be interchanged without changing the graph algorithm itself.

The first distribution strategy, EVERYWHERE places a hub's adjacency list on all locations of the graph's gang and balances the edges across the locations. This is an index-based partition and ignores the locality of the source and target vertices of the edges themselves. Using this strategy, all locations have a balanced number of edges on a per-hub basis, increasing the amount of parallelism available when visiting all edges of a hub vertex. Figure 5.3(b) provides an example of the EVERYWHERE strategy for the hub vertex h in Figure 5.3(a). In the example, all three locations have an equal number of edges.

We introduce another distribution strategy, NEIGHBORS, which places the edges only

on the locations on which the hub has a neighbor. In addition to limiting the locations to those of neighbors, we also place each edge (v, u) on the location of the target vertex u. This strategy is especially dynamic as the distribution of each hub edge list is dependent on the input data and thus heavily relies on the arbitrary subgroup support of the STAPL-RTS. Given N processors that contain the neighbors of a vertex v, the neighbor strategy limits the total number of messages generated by v in a single superstep to $\min\{N, d(v)\}$, where $d(v)$ is the out-degree of vertex v. This strategy is very similar to the mirroring technique in Pregel+ [134] and the large adjacency list partitioning technique in [135]. Figure 5.3(c) provides an illustration of the NEIGHBORS strategy for vertex h. All of the edges are stored in the same location color as the target vertex.

Finally, we introduce the STRIPED strategy, wherein a hub's adjacency list is distributed on only one location per shared-memory node in a strided fashion such that each hub is on a potentially different location on the node than the other hubs. Amongst those locations, the adjacency list itself is balanced.

5.2 Algorithmic Expression

The k-level-asynchronous paradigm provides vertex-centric specification for parallel graph algorithms with several operators. The first operator, the *vertex operator*, is applied on all active vertices of the graph and spawns visits to neighbor vertices, wherein the *neighbor operator* will be executed. The neighbor operator returns a status indicating whether or not the vertex operator should be reapplied on the neighbor. Execution is finished when all vertex operators vote to halt by returning false.

Neighbor visitation is exposed through two interfaces: `visit` and `visit_all_neighbors`. In both cases, a neighbor operator for an edge (v, u) is spawned directly from from the source vertex v and an asynchronous visit occurs. There exist various algorithms wherein properties on the edge itself are necessary when visiting a neighbor, such as distances

```
bool sssp_vertex_op(vertex v, visitor vis)
  if (v.color == GREY)          // Active if GREY
    v.color = BLACK;
    vis.VisitAllEdges(sssp_scatter_op(_1, v.dist), sssp_neighbor_op(), v);
    return true;                // vertex was Active
  else  return false;           // vertex was Inactive
```

(a) vertex-operator

```
int sssp_scatter_op(edge e, int dist)
  return dist + e.weight
```

(b) scatter-operator

```
bool sssp_neighbor_op(vertex u, int new_distance)
  if (u.dist > new_distance)
    u.dist = new_distance;  // update distance
    u.color = GREY;         // mark to be processed
    return true;            // vertex was updated
  else  return false;
```

(c) neighbor-operator

Figure 5.4: The vertex, neighbor and scatter operators for single-source shortest path.

in single-source shortest path and capacities in max flow. In this model, there is the assumption that the edge property is accessible from the source vertex v itself, which is not always the case. For example, in the NEIGHBORS strategy, the edge and its property are stored with the target vertex u and thus creation of the neighbor operator would require a synchronous read for the edge property on the location of vertex u.

We introduce an augmented model that supports a third additional *scatter-operator* which is applied on the edge (v, u) itself and computes what information should be available for the neighbor operator to be applied on vertex u. Figure 5.4 shows how to write single-source shortest path (SSSP) in this three-operator model. The vertex-operator and neighbor-operator are similar to the standard KLA expression of the algorithm. The main difference is the vertex operator invokes `visit_all_edges` instead of `visit_all_neighbors` and the edge visitation occurs through the scatter-operator. The scatter-operator computes the sum of the weight of the edge and the source vertex's current distance, passing this value along to the neighbor operator. In this manner, the

computation that occurs on the edge itself is completely decoupled and a lightweight task can be computed to asynchronously visit the edge and subsequently the neighbor vertex.

5.3 Implementation

We implemented our approach in the STAPL Graph Library (SGL) [101] to evaluate performance, due to SGL's ease of use and modification, and scalable performance [101, 6]. In this section, which is partially reprinted from [9], we give an overview of SGL and describe the relevant features and extensions needed to support the hierarchical approach, as well as provide an experimental evaluation. We also show how users can express important graph mining and graph analytics algorithms using our hierarchical paradigm, and thus benefit from our approach.

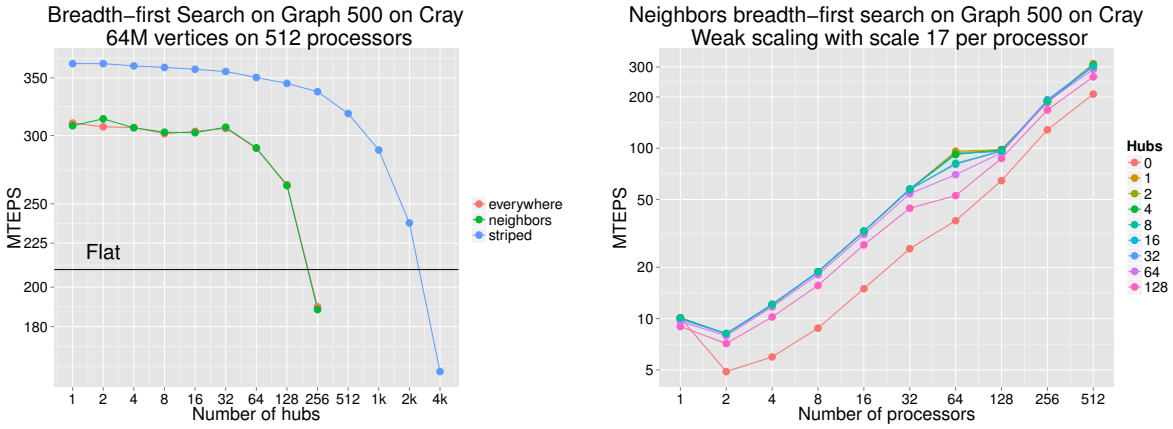
5.3.1 The STAPL Graph Library

SGL is a generic parallel graph library that provides a high-level framework which allows the user to concentrate on parallel graph algorithm development and decouples them from details of the underlying distributed environment. It consists of a parallel graph container (`pGraph`), a collection of parallel graph algorithms to allow users to easily process graphs at scale, and a graph engine that supports level-synchronous and asynchronous execution of algorithms.

The `pGraph` container is a distributed data storage built using the `pContainer` framework (PCF) [103] provided by the Standard Template Adaptive Parallel Library (STAPL) [104]. It provides a shared-object view of graph elements across a distributed-memory machine. The STAPL Runtime System (STAPL-RTS) and its communication library ARMI(Adaptive Remote Method Invocation) is decoupled from the underlying platform, providing portable performance, thus eliminating the need to modify STAPL applications. The STAPL-RTS abstracts the physical parallel processing elements into *location*s, components of a parallel machine where each one has a contiguous memory address space

and associated execution capabilities (e.g threads). ARMI uses the remote method invocation (RMI) abstraction to allow asynchronous communication on shared objects while hiding the underlying communication layer (e.g MPI, OpenMP).

5.4 Experimental Evaluation



(a) (b)

Figure 5.5: Graph 500 breadth-first search on Cray varying (a) the number of hubs on 512 processors and (b) the number of processors for a weak scaling experiment. Reprinted with permission from [9].

We performed our experiments on two different systems. The code was compiled with maximum optimization levels (`-DNDEBUG -O3`).

The first system is a Cray XK7m-200 system which consists of twenty-four compute nodes with AMD Opteron 6272 Interlagos 16-core processors at 2.1 GHz. Twelve of the nodes are single socket with 32 GB of memory, and the remaining twelve are dual socket nodes with 64 GB. Our benchmark code has been compiled with `gcc` 4.9.2 and we configured the STAPL-RTS with the OpenMP-based concurrency back-end with each location mapped to one OpenMP thread, pinned to one core.

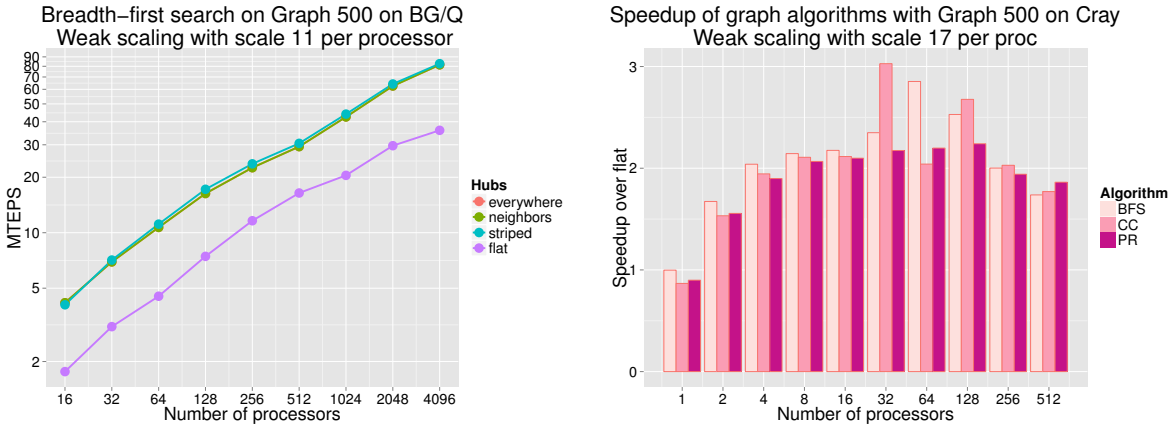
This second machine is an IBM BG/Q system available at Lawrence Livermore National Laboratory. BG/Q has $24,576$ nodes, with each node populated by a 16-core IBM PowerPC A2 processor clocked at $1.6\,\mathrm{GHz}$ and $16\,\mathrm{GB}$ of memory. The compiler used was `gcc` $4.7.2$, and we used the C++11-thread based multithreaded backend.

5.4.1 Graph Algorithms

To validate our technique, we implemented the Graph 500 benchmark [81], which performs a parallel breadth-first search on a scale-free network. Figure 5.5(a) shows the breadth-first search algorithm over the Graph 500 input graph. As shown, all three edge distribution strategies fare well over the baseline of non distributed adjacency lists for modest number of hubs, and then degrade in performance as more and more vertices are distributed. The `EVERYWHERE` and `NEIGHBORS` strategies behave similarly, as the set of locations that contain any neighbor is likely to be all locations for high-degree hub vertices. The `EVERYWHERE` and `NEIGHBORS` strategies are 49% and 51% faster than the baseline, respectively. The `STRIPED` strategy performs up to 75% faster than the baseline, which is a further improvement over the other two strategies. In our Cray machine, the cores exhibit high performance relative to the interconnect, and thus even modest amounts of communication can often bring about large performance degradation. The `STRIPED` strategy reduces the amount of off-node communication to create the parallel section from the source vertex location, bringing the performance of the algorithm above the other two strategies. We are currently investigating this phenomenon to derive a more rigorous model for distribution of edge lists.

Figure 5.5(b) shows a weak scaling study of the neighbor distribution strategy on Cray. As shown, the flat breadth-first search scales poorly from 1 to 2 processors due to an increase in the amount of communication which can stress the system. By distributing the edges for hubs, we are able to reduce this pressure on the communication and provide

better performance than the flat algorithm. The number of hubs to distribute needs to be carefully chosen, as too few hubs will not provide sufficient benefit in disseminating edge traversals, whereas too many hubs could overload the communication subsystem.



(a) (b)

Figure 5.6: Graph 500 (a) breadth-first search with various adjacency distributions on BG/Q and (b) various graph analytics algorithms on Cray. Reprinted with permission from [9].

In order to evaluate our technique at a larger scale, we also evaluated breadth-first search on the Graph 500 graph on BG/Q in Figure 5.6(a). In our evaluation, we found that although faster than the flat version, all three distribution strategies performed comparably with each other. At 4,096 processors, the distributed adjacency list breadth-first search using nested parallelism is 2.25x faster than the flat baseline, regardless of the distribution strategy. The kind of distribution strategy for nesting is machine-dependent and further reinforces the need for a modular and algorithm-agnostic mechanism to easily explore the possible space for nested parallelism for parallel graph algorithms.

Finally, to show the generality of the nested algorithm support in the context of dynamic computations, we implement two other popular graph analytics algorithms: Hash-

Min connected components (CC) [136] and PageRank [109] (PR). In Figure 5.6(b) we present the *oracle speedup* of the nested parallel versions over the flat version, where speedup is measured by computing the ratio between the best configuration and hub count for the nested parallel version and the flat version. All three algorithms show substantial improvement for all processor counts except for 1, where the overhead of creating a nested parallel section is measured. In some cases, the nested parallel version is able to achieve upwards of 3x speedup, such as on connected components at 32 cores.

6. GRAPH PROCESSING PARADIGMS

The STAPL Graph Library provides various strategies to execute parallel graph algorithms that are specified using a single model (the SGL model). These strategies include the KLA Paradigm, the Out-of-Core Paradigm and the Hierarchical Paradigm and its derivatives (hubs/h2). Although it is possible to use the same algorithm specification (i.e., the vertex and neighbor operator) with a different paradigm by passing them into the desired paradigm function, a unified approach would provide for a cleaner external interface.

We present a unification of the various execution strategies into a single interface named the SGL Execution Policy. This policy can be used with a single function as the SGL Execution Engine (sgl::execute). That is, we can specify a graph algorithm using only its two operators and encapsulate all other information related to how that algorithm should be executed into a separate policy object. This will lead to an interface as such the one shown in Figure 6.1.

In this chapter, we focus on a formalization of the semantics for a specific execution-policy: the k-level asynchronous.

6.1 KLA Machine

In this section, we create a formulation of an abstract machine capable of processing a KLA algorithm.

As inputs, we are given a graph $G = (V, E)$ and a partition of that graph across p processors $P : V \to \mathbb{N}$. The level of asynchrony $k \in \mathbb{N}$ defines how many hops a traversal can proceed in a single KLA superstep.

The global state of the machine can be defined as a tuple (D, C, Q^c, Q^n) where D is the vertex data associated with each vertex v, C is the state of the communication channels

```
sgl::execute(sgl::kla_policy{k}, g, vertex_op{}, neighbor_op{})
sgl::execute(sgl::async_policy{}, g, vertex_op{}, neighbor_op{})
sgl::execute(sgl::storage_policy{block_size}, g, vertex_op{}, neighbor_op{})
```

Figure 6.1: The vertex- and neighbor-operators for breadth-first search.

between processors, Q^c and Q^n are the processor-local queues of tasks for the current and next KLA superstep, respectively. The data associated with each vertex is algorithm-specific; for breadth-first search, each vertex has a tuple $(distance, active)$. A task is a tuple $(p, v, k', op\_type, op\_args...)$ representing an operation with certain arguments that will be applied to vertex v on processor p.

$t \leftarrow 0$
while $\neg empty(Q_t^c) \vee \neg empty(Q_t^n) \vee \neg empty(C_t)$ **do**
  $C_{t+1} \leftarrow f(C_t)$
  if $\neg empty(Q_t^c)$ **then**
    $(q, Q_{t+1}^c) \leftarrow \text{pop}(Q_t^c, \pi)$
    $S \leftarrow S^\frown q$
    $(D_{t+1}, q_{new}) \leftarrow \text{apply}(q, G, D_t)$
    foreach $x \in q_{new}$ **do**
      $C_{t+1}[P(q.v)][P(x.v)] \leftarrow C_{t+1}[P(q.v)][P(x.v)] \cup \{x\}$
    end
  else
    $S \leftarrow S^\frown \text{sync}$
    $Q_{t+1}^c \leftarrow Q_t^n$
    $Q_{t+1}^n \leftarrow \{\}$
  end
  $t \leftarrow t + 1$
end

Algorithm 6: Main loop of KLA machine

The machine operates by executing one task at a time and recording the new state of the vertex data and queues. There is a task selection policy π that determines what

```
1  tasks_new ← op(C_t)
2  foreach x ∈ tasks_new do
3  |  if x_op = vertex_operator then
4  |  |  x_k' ← x_k' + 1
5  |  end
6  |  if x_k' > k then
7  |  |  x_k' ← 0
8  |  |  Q^n_{t+1}[P(x_v)] ← Q^n_{t+1}[P(x_v)] ∪ {x}
9  |  else
10 |  |  Q^c_{t+1}[P(x_v)] ← Q^c_{t+1}[P(x_v)] ∪ {x}
11 |  end
12 end
```

Algorithm 7: Function to process communication channels

the next task to select from the queue is at each timestep. For example, a naive task selection policy could choose an arbitrary task from an arbitrary processor's queues. A more advanced policy may select from an arbitrary processor the task with the highest priority. An optimal policy will choose the globally best task to minimize some penalty function Ψ.

Each task may in turn add new tasks to be executed. These tasks are not put in queues directly, but are instead added to communication channels between the processor that executed the task and the processor in which the destination vertex is stored. At the beginning of each timestep, a channel processing function f is invoked to move tasks from channels to queues. This function returns the set of tasks that will be added to the queues from the communication channel. To represent an instantaneous network, f can simply return all of the tasks in the channel every time. However, latency can be simulated by delaying the movement through the channel between certain processors. This policy can be experimented with to simulate different network topologies.

In the end, we will produce a schedule S which is a sequence of tasks and a sequence of states T. There will be a task executed at each timestep that produces a state and

$|S| = |T| = t$.

6.1.1 Processing Loop

In Algorithm 6, we present the main processing loop of the KLA machine model. Initially, the current timestep t is initialized to 0 in line 1. Next, the loop starts and continues until both the current and next queue are empty.

First, we deal with the case where the current queue has tasks in lines 3-10. On line 4, a task q is chosen from the queue based on the selection policy π, and the task will be removed from the queue in the next timestep. On line 5, we append this task to the schedule. Next, we apply this task and record the change to the vertex data. The application of this task may result in a set of new tasks q_{new} that need to be queued. In lines 8-10, we add the new tasks to the appropriate communication channel. That is, if processor i is executing the current task and it generates a task for a vertex on processor j, we place the task in channel $P[i][j]$.

If the current superstep's queue is empty (lines 12-14), we instead add an explicit synchronization marker to the schedule and swap the current and next queues. Finally, we increase the timestep and recheck the queues.

Algorithm 7 illustrates the process for moving tasks from communication channels to queues. On line 1, we execute the user-defined function op which provides a list of messages to be delivered. In lines 2-11, we add the new tasks generated to the appropriate queues. First, we increase the task's effective k value k' if the task represents a vertex operator. If the task's k' value exceeds k, we will add this task to the appropriate processor's queue for the next superstep and reset its k'. This is because the task is at the edge of the superstep and should only be executed after all of the tasks in the current superstep have been executed. Otherwise, in line 10, this task should be executed in the current superstep, so it is added to the correct processor's queue.

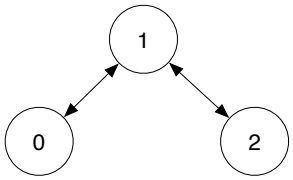
6.1.2 Breadth-First Search

For breadth-first search, the data associated with vertex is a tuple representing whether or not a vertex is active and what its current distance from the source is. The initial state of the machine $I = (D, Q^c, Q^n)$ will be as follows:

- $D_v = (\infty, 0)$ for $v \neq src$ and $D_{src} = (0, 1)$
- $Q^c = (Q^c_0, \ldots, Q^c_{p-1})$ where $Q^c_i = \{(i, v, 0, vop) | P(v) = i\}$
- $Q^n = (\{\}, \ldots, \{\})$

6.1.2.1 Example

Consider the following input graph:



This graph has the following partition: $P = \{(0, 0), (1, 0), (2, 1)\}$. We will be using a value of $k = 1$.

If we set our source to be vertex 0, our initial state $I = (D, Q^c, Q^n)$ will be as follows:

- $D_v = ((0, 1), (\infty, 0), (\infty, 0))$
- $Q^c = (\{(0, 0, 0, vop), (0, 1, 0, vop)\}, \{(1, 2, 0, vop)\})$
- $Q^n = (\{\}, \ldots, \{\})$

Say we choose an arbitrary task to execute first: $S_0 = (\{(0, 1, 0, vop)\}$. This will induce the following state:

- $D_v = ((0, T), (\infty, F), (\infty, F))$
- $Q^c = (\{(0, 0, 0, vop)\}, \{(1, 2, 0, vop)\})$
- $Q^n = (\{\}, \ldots, \{\})$

Notice the only change is the that the task was removed from processor 0's queue. Since vertex 1 was not active, there was no change to the data or queues. Next, suppose we pick the task for vertex 0, which happens to be active: $S_1 = (\{(0, 0, 0, vop)\}$. This will induce the state:

- $D_v = ((0, F), (\infty, F), (\infty, F))$
- $Q^c = (\{(0, 1, 1, nop, 1, 0)\}, \{(1, 2, 0, vop)\})$
- $Q^n = (\{\}, \ldots, \{\})$

This new state changed the active flag for vertex 0 and introduced a neighbor operator task for vertex 1 in processor 0's queues. Looking at one more step, we can choose the task $S_3 = (0, 1, 1, nop, 1, 0)$. The state that will be induced is:

- $D_v = ((0, F), (1, T), (\infty, F))$
- $Q^c = (\{(0, 1, 1, vop)\}, \{(1, 2, 0, vop)\})$
- $Q^n = (\{\}, \ldots, \{\})$

As shown, the data for vertex 1 has changed to set its distance to 1 and mark it as active. In addition, the neighbor operator added a task to processor 0's queue to invoke the vertex operator on vertex 1. Notice that the k' value increased by one; when this value

reaches k, the task is instead added to the appropriate queue in the next superstep Q^n with the k' field reset to 0.

In the end, we will have a schedule S and a final state T_{t-1} where the queues are empty and the vertex data is at its final value.

6.2 Using the KLA Model

The KLA machine model can be used as a general framework to prove properties of KLA algorithms.

6.2.1 Proof Through Witness

One method of proof that is enabled by the KLA machine is the ability to prove that a predicate regarding an algorithm's behavior could happen through the ability to present a valid scheduling in which the predicate is true.

For example, if we consider the approximate breadth-first search algorithm from Chapter 4, we can prove that the error bounds of the distance is at least as large as a certain function. Concretely, if we want to prove that the error bound has to be at least $d_v^* + k$, where d_v^* is the exact distance of vertex v from the source, our predicate would be $\exists v \mid d_v = d_v^* + k$. In order to prove this statement, it is sufficient to present a valid scheduling of an execution where the predicate is true on the vertex data of the final state. This schedule is therefore a witness that the predicate is true.

As we know that the error bounds for the approximate breadth-first search algorithm with the naive scheduling policy is $d_v \leq d_v^* \times k$, we can use this same technique to prove that the same algorithm with a different scheduling policy would have the same bounds by attempting to find a witness to prove the predicate $d_v = d_v^* \times k$ is true for some input.

6.2.2 Proof Through Impossible Schedule

Another technique that can be used to prove properties of a given KLA algorithm is through contradiction. The intuition is as follows. Say we wish to prove that certain predicate pr is true. We will first assume that it is not true. Then we will argue that the only way in which the predicate could be false is through a schedule that is impossible due to the either the selection policy π or because it would violate some aspect of the KLA machine model. Therefore, the original predicate pr must be true.

6.3 Results

Using the KLA machine model, we were able to use the proof through witness technique to discover an important property with the KLA model in general, which is the KLA guarantees are only enforced if tasks are processed in a processor-local FIFO order. That is, if we execute a KLA machine with a selection policy π wherein tasks from the same processor are not executed in FIFO order, it is possible for a traversal to extend outside of the bounds of the KLA superstep.

6.3.1 Priority Scheduler Bounds

In this section, we will prove that using a scheduling policy π_{min} that uses the priority of tasks in local queues to choose the next task does not improve the worst-case bounds for the approximate breadth-first search algorithm.

Specifically, we will show a special graph for which a worst-case scheduling can result in the following bounds for the distance:

$$d_k(v) = \frac{d_0(v)}{2}k \tag{6.1}$$

This bound is within the same class as the previously shown bounds with no restrictions on scheduling, which is $O(d_0(v) \times k)$.

6.3.1.1 *Problem Description.*

We will be using the KLA machine with a priority scheduler to execute the KLA approximate breadth-first algorithm. Each task will receive a priority in the following way:

- Neighbor operator: the priority is the distance that the operator is storing
- Vertex operator: lowest priority (maximum integer)

When π_{min} chooses a task, it chooses the task with the smallest priority. That is, it chooses the neighbor operator with the lowest distance, or a vertex operator if there are no neighbor operators available. As we will see, no queue will have more than one task in it, so any processing function will have the same result.

For our channel processing function f, we will choose the standard FIFO processing function. That is, messages sent between the same pair of processors p_0 and p_1 are ordered and there is no ordering guarantee amongst messages destined to some other processor $q \neq p_0 \neq p_1$. This is the same behavior guaranteed through the RMI causal consistency model of the STAPL runtime system.

Finally, we will assume that $\tau = 1$, meaning no revisitation will trigger a repropagation. This is consistent with the original bounds proof, as the τ factor grows significantly faster than the k factor, and thus the total bounds is limited by k.

6.3.1.2 *Worst-case Schedule*

Figures 6.2 illustrate a worst case schedule that exhibits distances that are equal to those shown in Equation 6.1 for $k = 5$. Consider why this schedule is valid. The traversal starting at vertex 0 will spawn two neighbor operators to its two neighbors on processor P_1. The operator for the lower vertex could be delayed and the longer upper path could be taken, as illustrated. When the lower neighbor finally receives a visitation, it will update

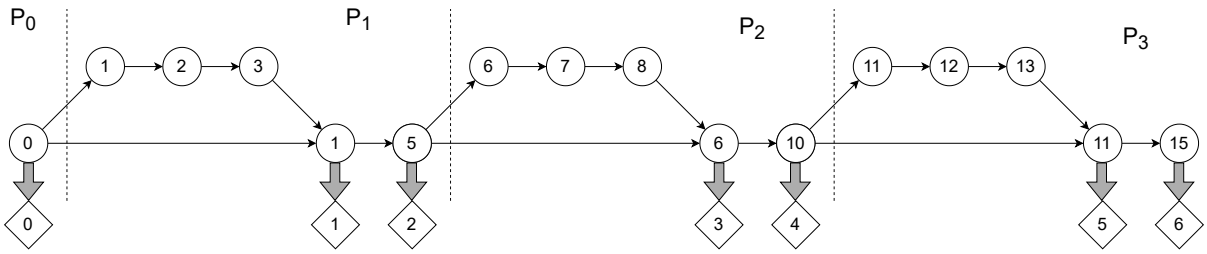


Figure 6.2: Example graph that shows the worst-case bounds even in the presence of priority scheduling. The vertices in the graph contain the computed distances and the associated diamonds are the true distances for vertices of interest.

its true distance but it already propagated the longer incorrect path. The vertex that is labeled distance 5 will then propagate that distance to its neighbors on processor P_2. In the end, the vertex that is labeled 15 distance has a true distance of 6, which matches with Equation 6.1. Note that at no point in this schedule was a lower priority operator processed before a higher priority one, which satisfies the constraint of the π_{min} policy. Due to arbitrary network delays, the presented schedule follows the error function defined in Equation 6.1. Thus, even with this priority scheduler, the worst-case bounds of the error does not improve from the normal scheduler presented in Chapter 4.

6.4 Conclusion

This chapter introduced a synthesis of the various execution strategies presented in STAPL Graph Libraryas well as a formalization of the computational engine for the k-level-asynchronous model. Using this formalization, we are able to prove various properties of graph algorithms in that model, such as the lower bounds of the approximate breadth-first search algorithm with a different priority scheduling.

7. IMPLEMENTATION FOR PRACTICAL PERFORMANCE

Creating a high-performance graph framework is challenging, particularly due to the irregular access of graph elements (edges and vertices) where the access pattern is unknown *a priori* and is difficult to predict. In this chapter, we discuss practical issues that arise when implementing a graph framework and the tradeoffs that are made when balancing generic capability with high performance.

Our contribution in this chapter is the exploration of algorithm-driven implementations of crucial data structures, such as the graph storage type and active vertex set. We introduce a technique for algorithm-driven redundancy elimination through the choice of the active vertex set representation. Finally, we discuss utilizing appropriate implementations for the core graph processing engine and opting out of expensive features such as data-flow graphs and instead using point-to-point communication.

7.1 Frontier Representation

When implementing a parallel graph algorithm, the access to graph elements is often expressed explicitly in a data structure that stores the graph vertices that will be accessed next. For example, in level-synchronous breadth-first search [5], this data structure stores the source vertex of the traversal for the first superstep, after which it is populated with that vertex's neighbors which represent the vertices to process in the next superstep. This active set of vertices $F \subset V$ is often referred to as the *frontier* of the traversal.

This frontier can be represented in many different kinds of data structures, each of which possess their own performance characteristics and conditions for which the structure is valid. For example, if the frontier contains a large percentage of all vertices in the graph, it may be beneficial to represent this set as a condensed bitmap, where there is a single bit associated with every vertex in the graph that is set to 1 if the vertex resides

in the frontier. On the other hand, if there are only a few active vertices during a given superstep–e.g., the first superstep of breadth-first search–then an enumerated list of vertex IDs stored as an array would result in better storage and time complexity. Further, there are some algorithms, such as PageRank [109], for which all vertices are in the frontier for every superstep. In this scenario, it would be ideal to forgo an explicit frontier and instead simply use the graph data structure itself as the set of active vertices.

Often in graph processing libraries, it is common that there is only a single type of frontier provided by the framework and the user has no choice but to use a potentially inefficient representation for their workload. In other libraries, it is left as a choice for the user to explicitly select the frontier representation. In this case, it is often difficult for an end-user to accurately pick a frontier representation, as they may be unaware of the potential ramifications of any given data structure.

In this work, we introduce a methodology that abstracts the choice of frontier representation and provides tools that algorithm writers can use to drive the frontier data structure selection process. Our goal is to eliminate the end user's need to manually select a frontier representation and to provide an interface for the algorithm developer to describe their algorithm's behavior which will in turn influence the choice of frontier representation.

Our contribution in this work is as follows:

- Identify and define characteristics of graph algorithms that developers can provide that describe their algorithm's behavior without explicitly dictating a frontier representation.
- Introduce a frontier selection framework that automatically chooses an efficient frontier data structure based on the aforementioned characteristics and graph workload features.
- Implement efficient graph algorithms using appropriate frontier data structures with-

out direct user intervention.

7.1.1 Algorithm Characterization

The choice of frontier representation is driven by many factors, including the precise ratio of active vertices during an instantiation of an algorithm, an algorithmically specified estimate of superstep occupancy, and whether or not it is algorithmically valid for there to exist duplicate entries in the frontier.

7.1.1.1 Active Vertex Ratio

The active vertex ratio is a rough estimate of how many vertices will be active in a given superstep during the traversal. For example, if it is expected that roughly 90% of the vertices will be active in a superstep, then the user can pass a value of 0.9.

Internally, this number is used to select different implementations of the frontier data structure used to track which vertices are active. If a high number of vertices are anticipated to be active (more than 75%), we will use a bitmap structure to efficient keep track of which vertices are in the frontier.

7.1.1.2 Superstep Occupancy

The superstep occupancy options allows the algorithm writer to describe the number of vertices for specific supersteps – the first superstep, a middle superstep and the last superstep. For example, in breadth-first search, the first superstep has a single vertex, a middle superstep has a subset of vertices the last superstep has a subset of vertices. Table 7.1 provides a list of other example algorithms, along with their superstep occupancy values.

7.1.1.3 Ordering

For some algorithms, the ordering of the vertex operator and neighbor operator are important and invoking a vertex operator on the same vertex twice in a row would result in

Algorithm	First superstep	Intermediate superstep	Last superstep
BFS	single	subset	subset
CC	all	subset	subset
PageRank	all	all	all

Table 7.1: Example algorithm superstep occupancy

```
1  Function LabelPropVertexOperator(v, level)
2      if level is odd then
3          VisitAllNeighbors(v, LabelPropNeighborOp, v.label)
4      else
5          frequent = MostFrequentLabels(v.labels)
6          v.labels = ∅
7          if v.label = frequent then
8              return false
9          else
10             v.label = frequent
11             return true
12         end
13     end
```

Algorithm 8: Label propagation community detection vertex operator.

incorrect behavior. For example, consider the label-propagation community detection algorithm [137] depicted in Algorithms 8 and 9. This algorithm labels each vertex based on the most commonly occurring label amongst its neighbors by storing each label received by the neighbor operator into temporary storage and allowing the vertex operator to compute the most frequent label in its storage and clearing the storage. During the execution of the algorithm, it would be incorrect for a vertex operator to be applied to some vertex v twice in a row without an invocation of some intermediate neighbor operator along an edge (u, v). If such a scenario were to arise, then the vertex operator would send its label multiple times to its neighbors, thus diluting the quality of the communities discovered.

To this end, our framework asks the algorithm writer to provide a list of operator

```
1 Function LabelPropNeighborOperator(u, label)
2 |   u.labels = u.labels ∪{ label }
3 |   return true
```

Algorithm 9: Label propagation community detection neighbor operator.

orderings that result in a valid execution:

1. (VertexOperator(v), NeighborOperator(v, u), VertexOperator(v))

2. (VertexOperator(v), VertexOperator(v))

Ordering 1, which we denote as an *interleaved ordering* is the most natural ordering restriction, wherein a neighbor operator is called on some vertex v before a vertex operator can be invoked on that same vertex again. Ordering 2, which we denote as a *successive ordering* allows for a vertex operator to be invoked twice in succession for the same vertex v.

Based on these two ordering restrictions, there are four cases that can arise:

- Only the interleaved ordering is allowed. This is the case for the label-propagation community detection algorithm.

- Both the interleaved and successive ordering are allowed. This case is found in the majority of studied graph algorithms in this dissertation.

- Only the successive ordering is allowed. It is unknown if any useful graph algorithm has this restriction.

- Neither ordering is allowed. It is unknown if any useful graph algorithm has this restriction.

The ordering restrictions on the algorithm operators guide the implementation of the frontier traversal both in terms of correctness and potential performance optimizations. An algorithm which only allows for interleaved ordering of operators must maintain a frontier that does not allow duplicates, as the presence of duplicate vertices in the frontier creates the potential for the vertex operator to be invoked on the same duplicate vertex successively.

Maintaining a frontier data structure that does not allow duplicates comes at a cost. For a frontier with relatively few entries, the most natural implementation for a non-unique set would be a dynamic array, which allows for amortized $O(1)$ insertion and cache-friendly traversal. However, if the algorithm requires the use of a unique frontier due to its interleaved ordering requirement, then a data structure such as an ordered-tree or hash-table must be used, resulting in either worse time complexity, higher constants or worse cache behavior.

7.1.2 Frontier Selection Framework

Algorithmic information about the behavior of the frontier for key supersteps is can be effectively utilized to select among several candidate frontier data structures on a per-superstep basis, as well as provide for some other optimizations dealing with populating and enumerating vertices in the frontier. Figure 7.1 illustrates the design of the frontier selection framework and Table 7.2 provides a small set of examples for selecting frontier implementations based on the properties discussed in previous sections.

Internally, the occupancy information is used to select appropriate implementations of our frontier data structure. For example, if an algorithm states that the occupancy is "all" for every superstep, then we do not need to have an explicit frontier. Figure 7.3 provides an experiment where an explicit bitmap frontier is used for PageRank on a Kronecker graph instead of an implicit frontier that simply traverses the graph's vertices for the active set.

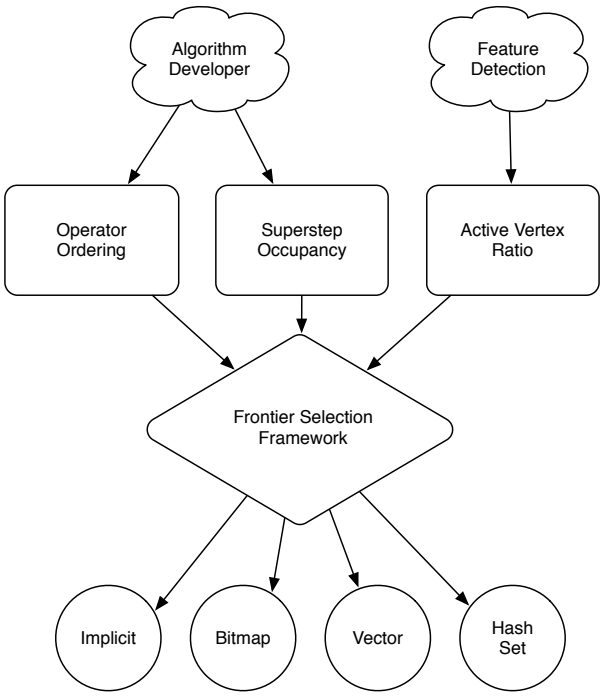


Figure 7.1: Architecture of frontier selection framework

The experiment shows that the implicit frontier implementation has a modest but measurable improvement of the explicit bitmap. Note that the version of the explicit frontier that used an array-backed storage instead of a bitmap storage ran out of memory for the same experiment.

Another small optimization that can be employed is to not use an explicit frontier for the first superstep when the algorithm is known to include the entire set of vertices for the first superstep. A small experiment with the label-propagation connected components algorithm shows a modest 2% improvement in sequential performance with this optimization on the CRAY-XK7 platform for a scale 22 Kronecker graph.

Example	Superstep Occupancy	Ordering	Active Vertex Ratio	Frontier Data Structure
Breadth-first search on scale-free graph	(single, subset, subset)	No restrictions	90%	Bitmap
Breadth-first search on road network	(single, subset, subset)	No restrictions	10%	Dynamic Array
PageRank	(all, all, all)	No restrictions	-	Implicit
LP community detection on road-network	(all, subset, subset)	Interleaved only	10%	Hash Set
LP community detection on scale-free graph	(all, subset, subset)	Interleaved only	90%	Bitmap

Figure 7.2: Preferable frontier data structure based on workload features.

7.1.3 Experimental Evaluation

In Figure 7.4, we demonstrate the effect of an interleaved ordering restriction for breadth-first search with respect to the same algorithm with no restriction. Note that the breadth-first search algorithm does not have an ordering restriction, and thus providing that information to the SGL interface allows for the selection of an optimized data structure for the frontier. For the sparse case of a low active vertex ratio, the data structure that is used for the frontier is a dynamic array for the unrestricted version and a hash-table for the interleaved ordering version. For the dense case of a high active vertex ratio, a bitmap is used for both versions, as a bitmap by its nature does not allow duplicates. In Figure 7.4, the sparse cases see a 2.2x speedup for the Texas road network and a 1.5x speedup for the scale 20 Kronecker graph. These speedups stem exclusively from the choice of a dynamic array for the frontier data structure instead of the hash-table. Thus, by allowing for the user to provide ordering information about their algorithm, we enable the choice of an optimized frontier data structure which ultimately leads to improved performance.

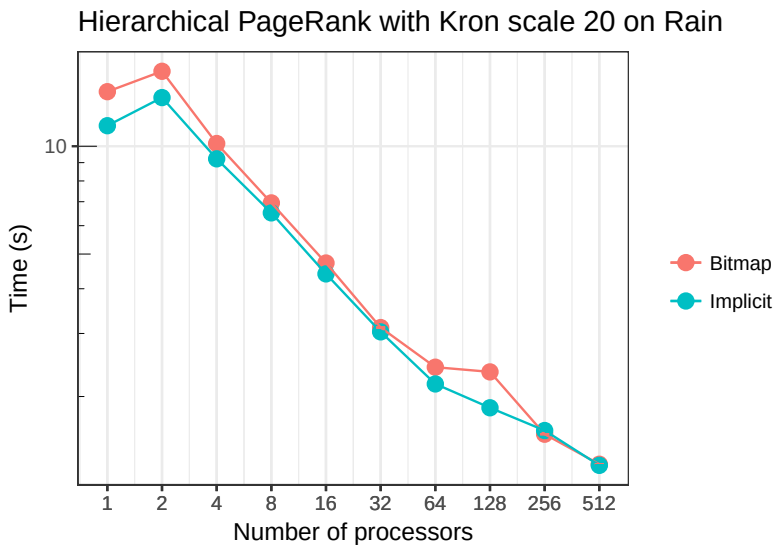


Figure 7.3: Implicit frontier optimization vs using an explicit frontier with bitmap storage with the PageRank algorithm

7.2 Graph Runtime Implementation

When implementing a general graph processing framework, it is important to consider the underlying graph processing engine as well as the data structure implementation.

7.2.1 Graph Processing Engine

In the STAPL Library, all computation is expressed as a composition of task-dependency graphs using the PARAGRAPH component. PARAGRAPHs allow the user to create complex data flow graphs that can be executed in a correct, scalable and efficient manner for all general kinds of data flow graphs. However, the generic capability of the task-dependency graph execution engine could lead to a less efficient implementation than a hand-crafted solution tailored for the specific problem.

Our work shows that using underlying simple communication primitives instead of high-level dependency patterns (such as map and reduce) can lead to large performance improvements. For example, we utilize STAPL Runtime System's low-level communica-

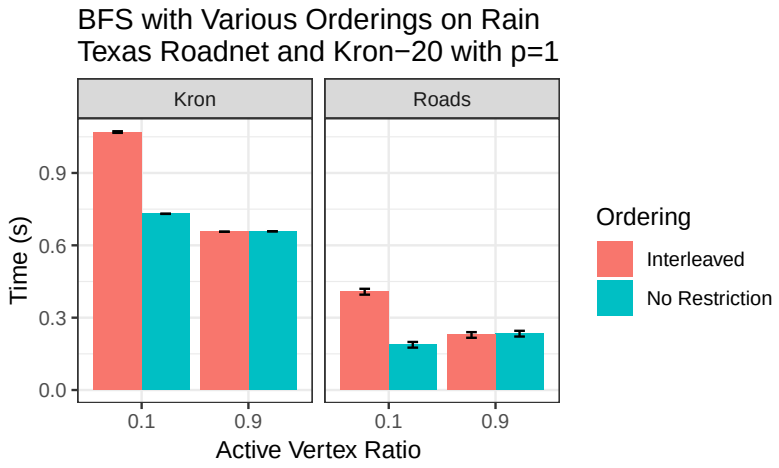


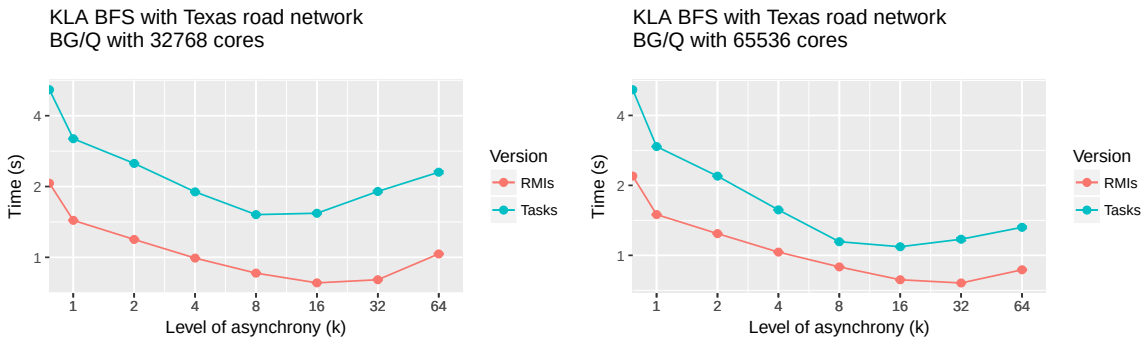
Figure 7.4: Effect of interleaved ordering restriction vs no ordering restriction for breadth-first search on a single core of CRAY-XK7.

tion methods `async_rmi` and `rmi_fence` as the main methods of sending messages and synchronizing between processes.

Figure 7.5 illustrates the performance benefit of using raw RMI messages to communicate instead of using the high-level and often heavyweight PARAGRAPH component. With 32,768 cores on BG/Q, the RMI-based method is almost 2x as fast as the task-based method for various levels of asynchrony. At 65,536 cores, the performance improvements still exist, but are less pronounced.

7.2.2 Graph Data Structure

Equally important for a high-performance graph framework is the graph data structure implementation. Many general purpose graph frameworks employ an adjacency-list data structure to represent graphs. Although adjacency-lists are flexible in terms of insertion and deletion of vertices and edges, they suffer from numerous performance implications. Poor cache locality is often a symptom of adjacency-list implementations. Although this can be partially alleviated with carefully tuned memory allocators for the individual lists,



(a) (b)

Figure 7.5: Runtime of a breadth-first search using task-based vs raw message based interface on BG/Q.

such a data structure will not be as performant as one with more contiguity in its edge allocations. Additionally, adjacency list structures are often heavyweight, as they need to store multiple pointers per vertex which represent the beginning and end of its adjacency list.

In SGL, we balance the performance and flexibility constraints of graph representations by providing both adjacency lists and compressed sparse row data structures. Additionally, we allow the user to specify the width of vertex IDs (32-bit and 64-bit).

8. CONCLUSION

Throughout this dissertation, we introduced several techniques for speeding up parallel graph data structures, parallel graph algorithms and general graph traversals. Our novel bounded asynchrony technique avoids many of the pitfalls of expensive global synchronizations while also balancing performance penalties of unbounded asynchronous execution. Using this methodology, we introduce a family of approximate parallel graph algorithms which balance the tradeoffs of performance and accuracy.

We also introduce a mechanism to express graph algorithms in a nested parallel manner which improves parallel performance by matching the algorithmic specification to the underlying machine architecture.

Finally, we provide broad guidelines for implementing a generic and performant graph framework for modern parallel systems by exploring various frontier storage and processing techniques, data structure representations and graph processing engine runtimes.

REFERENCES

[1] D. Gregor and A. Lumsdaine, "The parallel BGL: A generic library for distributed graph computations," in *Parallel Object-Oriented Scientific Computing (POOSC)*, July 2005.

[2] J. W. Berry, B. Hendrickson, S. Kahan, and P. Konecny, "Software and algorithms for graph queries on multithreaded architectures," *Parallel and Distributed Processing Symposium, International*, vol. 0, p. 495, 2007.

[3] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, (New York, NY, USA), pp. 135–146, ACM, 2010.

[4] J. H. Reif, ed., *Synthesis of Parallel Algorithms*. San Mateo, CA: Morgan Kaufmann, 1993.

[5] A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, (New York, NY, USA), pp. 65:1–65:12, ACM, 2011.

[6] Harshvardhan, A. Fidel, N. M. Amato, and L. Rauchwerger, "KLA: A new algorithmic paradigm for parallel graph computations," in *Proc. Intern. Conf. Parallel Architecture and Compilation Techniques (PACT)*, PACT '14, (New York, NY, USA), pp. 27–38, ACM, 2014. Conference Best Paper Award.

[7] A. Fidel, S. A. Jacobs, S. Sharma, N. M. Amato, and L. Rauchwerger, "Using load balancing to scalably parallelize sampling-based motion planning algorithms,"

in *Proc. International Parallel and Distributed Processing Symposium (IPDPS)*, (Phoenix, Arizona, USA), May 2014.

[8] A. Fidel, F. C. Sabido, C. Riedel, N. M. Amato, and L. Rauchwerger, "Fast approximate distance queries in unweighted graphs using bounded asynchrony," in *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2016.

[9] I. Papadopoulos, N. Thomas, A. Fidel, D. Hoxha, N. M. Amato, and L. Rauchwerger, "Asynchronous nested parallelism for dynamic applications in distributed memory," in *Int. Workshop on Languages and Compilers for Parallel Computing (LCPC), in Lecture Notes in Computer Science (LNCS)*, (Raleigh, NC, USA), September 2015.

[10] P. Compeau, P. Pevzner, and G. Tesler, "How to apply de bruijn graphs to genome assembly," *Nature biotechnology*, vol. 29, no. 11, pp. 987–991, 2011.

[11] A. Fidel, S. A. Jacobs, S. Sharma, N. M. Amato, and L. Rauchwerger, "Using load balancing to scalably parallelize sampling-based motion planning algorithms," in *Proc. IEEE Int. Parallel and Distributed Processing Symp. (IPDPS)*, (Phoenix, Arizona, USA), May 2014.

[12] S. A. Jacobs, K. Manavi, J. Burgos, J. Denny, S. L. Thomas, and N. M. Amato, "A scalable method for parallelizing sampling-based motion planning algorithms," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, pp. 2529–2536, 2012.

[13] M. Zeyao and F. Lianxiang, "Parallel flux sweep algorithm for neutron transport on unstructured grid," *The Journal of Supercomputing*, vol. 30, pp. 5–17, Oct 2004.

[14] J. Barnes and P. Hut, "A hierarchical O(N log N) force-calculation algorithm," *Nature*, vol. 324, pp. 446–449, Dec. 1986.

[15] W. H. Butt, M. U. Akram, S. A. Khan, and M. Y. Javed, "Covert network analysis for key player detection and event prediction using a hybrid classifier," *The Scientific World Journal*, vol. 2014, 2014.

[16] U. Brandes, "A faster algorithm for betweenness centrality," *Journal of Mathematical Sociology*, pp. 163–177, 2001.

[17] G. Sabidussi, "The centrality index of a graph," *Psychometrika*, vol. 31, pp. 581–603, Dec 1966.

[18] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, (Berkeley, CA, USA), pp. 29–42, USENIX Association, 2008.

[19] L. Valiant, "Bridging model for parallel computation," *Comm. ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[20] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks," in *Nature*, pp. 440–442, 1998.

[21] F. McSherry, M. Isard, and D. G. Murray, "Scalability! but at what cost?," in *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS'15, (Berkeley, CA, USA), pp. 14–14, USENIX Association, 2015.

[22] L. Zhao, S. Sakr, A. Liu, and A. Bouguettaya, *Big Data Processing Systems*, pp. 135–176. Cham: Springer International Publishing, 2014.

[23] J. Murty, *Programming Amazon Web Services*. O'Reilly, first ed., 2008.

[24] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, no. 01, pp. 5–20, 2007.

[25] J. S. Firoz, T. A. Kanewala, M. Zalewski, M. Barnas, and A. Lumsdaine, "Context matters: Distributed graph algorithms and runtime systems: A case study of distributed graph traversals," in *Proceedings of the Platform for Advanced Scientific Computing Conference*, PASC '16, (New York, NY, USA), pp. 12:1–12:10, ACM, 2016.

[26] M. J. Quinn and N. Deo, "Parallel graph algorithms.," *ACM Comput. Surv.*, pp. 319–348, 1984.

[27] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "Graphx: A resilient distributed graph system on spark," in *Workshop on Graph Data-management Experiences and Systems*, pp. 1–6, 2013.

[28] R. Pearce, M. Gokhale, and N. M. Amato, "Faster parallel traversal of scale free graphs at extreme scale with vertex delegates," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, (Piscataway, NJ, USA), pp. 549–559, IEEE Press, 2014.

[29] E. Fleury, S. Lattanzi, V. Mirrokni, and B. Perozzi, "ASYMP: Fault-tolerant Mining of Massive Graphs," *arXiv e-prints*, p. arXiv:1712.09731, Dec. 2017.

[30] J. Shun and G. E. Blelloch, "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.

[31] S. Grossman, H. Litz, and C. Kozyrakis, "Making pull-based graph processing performant," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, (New York, NY, USA), pp. 246–260, ACM, 2018.

[32] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Graphlab: A new framework for parallel machine learning," *CoRR*, vol. abs/1006.4990, 2010.

[33] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, (Berkeley, CA, USA), pp. 17–30, USENIX Association, 2012.

[34] M. Garey, D. Johnson, and L. Stockmeyer, "Some simplified np-complete graph problems," *Theoretical Computer Science*, vol. 1, no. 3, pp. 237 – 267, 1976.

[35] G. Karypis and V. Kumar, "METIS, unstructured graph partitioning and sparse matrix ordering system. version 2.0," tech. rep., University of Minnesota, Department of Computer Science, Minneapolis, MN, Aug. 1995.

[36] U. Catalyurek, E. Boman, K. Devine, D. Bozdag, R. Heaphy, and L. Riesen, "Hypergraph-based dynamic load balancing for adaptive scientific computations," in *Proc. International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 1 –11, March 2007.

[37] G. Karypis and V. Kumar, "Multilevel k-way partitioning scheme for irregular graphs," *Journal of Parallel and Distributed Computing*, vol. 48, no. 1, pp. 96–129, 1998.

[38] A. Abou-Rjeili and G. Karypis, "Multilevel algorithms for partitioning power-law graphs," in *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, (Washington, DC, USA), pp. 124–124, IEEE Computer Society, 2006.

[39] G. M. Slota, S. Rajamanickam, K. Devine, and K. Madduri, "Partitioning trillion-edge graphs in minutes," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 646–655, May 2017.

[40] A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, (New York, NY, USA), pp. 65:1–65:12, ACM, 2011.

[41] R. Pearce, M. Gokhale, and N. M. Amato, "Scaling techniques for massive scale-free graphs in distributed (external) memory," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS '13, (Washington, DC, USA), pp. 825–836, IEEE Computer Society, 2013.

[42] J. Yan, G. Tan, and N. Sun, "Study on partitioning real-world directed graphs of skewed degree distribution," in *2015 44th International Conference on Parallel Processing*, pp. 699–708, Sept 2015.

[43] Y. Zhang, V. Kiriansky, C. Mendis, M. Zaharia, and S. Amarasinghe, "Making caches work for graph analytics," *IEEE BigData 2017*, Dec 2017.

[44] C. Avery, "Giraph: Large-scale graph processing infrastructure on hadoop," Hadoop Summit, 2011.

[45] R. R. McCune, T. Weninger, and G. Madey, "Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing," *ACM Comput. Surv.*, vol. 48, pp. 25:1–25:39, Oct. 2015.

[46] T. White, *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st ed., 2009.

[47] M. Chen, S. Mao, and Y. Liu, "Big data: A survey," *Mobile Networks and Applications*, vol. 19, pp. 171–209, Apr 2014.

[48] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2010.

[49] Y. Zhang, H.-S. Ko, and Z. Hu, "Palgol: A high-level dsl for vertex-centric graph processing with remote data access," in *Programming Languages and Systems* (B.-Y. E. Chang, ed.), (Cham), pp. 301–320, Springer International Publishing, 2017.

[50] M. Han and K. Daudjee, "Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems," *Proc. VLDB Endow.*, vol. 8, pp. 950–961, May 2015.

[51] Y. Liu, C. Zhou, J. Gao, and Z. Fan, "Giraphasync: Supporting online and offline graph processing via adaptive asynchronous message processing," in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, CIKM '16, (New York, NY, USA), pp. 479–488, ACM, 2016.

[52] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis, "Mizan: A system for dynamic load balancing in large-scale graph processing," in *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, (New York, NY, USA), pp. 169–182, ACM, 2013.

[53] Q. Chen, S. Bai, Z. Li, Z. Gou, B. Suo, and W. Pan, "GraphHP: A Hybrid Platform for Iterative Graph Processing," *arXiv e-prints*, p. arXiv:1706.07221, June 2017.

[54] F. Yang, J. Li, and J. Cheng, "Husky: Towards a more efficient and expressive distributed computing framework," *Proc. VLDB Endow.*, vol. 9, pp. 420–431, Jan. 2016.

[55] H. Qu, O. Mashayekhi, D. Terei, and P. Levis, "Canary: A Scheduling Architecture for High Performance Cloud Computing," *arXiv e-prints*, p. arXiv:1602.01412, Feb. 2016.

[56] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, pp. 2091–2100, Aug 2014.

[57] P. Sun, Y. Wen, T. Nguyen Binh Duong, and X. Xiao, "GraphH: High Performance Big Graph Analytics in Small Clusters," *arXiv e-prints*, p. arXiv:1705.05595, May 2017.

[58] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a pc," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, (Berkeley, CA, USA), pp. 31–46, USENIX Association, 2012.

[59] X. Zhu, W. Han, and W. Chen, "Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning," in *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, (Berkeley, CA, USA), pp. 375–386, USENIX Association, 2015.

[60] P. Sun, Y. Wen, T. Nguyen Binh Duong, and X. Xiao, "GraphMP: I/O-Efficient Big Graph Analytics on a Single Commodity Machine," *arXiv e-prints*, p. arXiv:1810.04334, Oct. 2018.

[61] J. Webber, "A programmatic introduction to neo4j," in *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, (New York, NY, USA), pp. 217–218, ACM, 2012.

[62] "Apache tinkerpop." http://tinkerpop.apache.org/, 2017. Accessed: 2018-01-26.

[63] G. M. Kurtzer, V. Sochat, and M. W. Bauer, "Singularity: Scientific containers for mobility of compute," *PLOS ONE*, vol. 12, pp. 1–20, 05 2017.

[64] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, Mar. 2014.

[65] A. Chu, "Magpie." https://github.com/LLNL/magpie, 2018.

[66] J. Nelson, B. Holt, B. Myers, P. Briggs, S. Kahan, L. Ceze, and M. Oskin, "Grappa: A latency-tolerant runtime for large-scale irregular application," WRSC'14, April 2014.

[67] M. Méndez-Lojo, D. Nguyen, D. Prountzos, X. Sui, M. A. Hassaan, M. Kulkarni, M. Burtscher, and K. Pingali, "Structure-driven optimizations for amorphous data-parallel programs," in *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '10, (New York, NY, USA), pp. 3–14, ACM, 2010.

[68] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, (New York, NY, USA), pp. 456–471, ACM, 2013.

[69] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun, "Green-marl: a dsl for easy and efficient graph analysis," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, (New York, NY, USA), pp. 349–362, ACM, 2012.

[70] J. S. Firoz, M. Barnas, M. Zalewski, and A. Lumsdaine, "The value of variance," in *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, ICPE '16, (New York, NY, USA), pp. 287–295, ACM, 2016.

[71] S. Hong, S. Depner, T. Manhardt, J. Van Der Lugt, M. Verstraaten, and H. Chafi, "Pgx.d: A fast distributed graph processing engine," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, (New York, NY, USA), pp. 58:1–58:12, ACM, 2015.

[72] K. Zhang, R. Chen, and H. Chen, "Numa-aware graph-structured analytics," *SIGPLAN Not.*, vol. 50, pp. 183–193, Jan. 2015.

[73] J. Sun, H. Vandierendonck, and D. S. Nikolopoulos, "Graphgrind: Addressing load imbalance of graph partitioning," in *Proceedings of the International Conference on Supercomputing*, ICS '17, (New York, NY, USA), pp. 16:1–16:10, ACM, 2017.

[74] Y. Simmhan, A. Kumbhare, C. Wickramaarachchi, S. Nagarkar, S. Ravi, C. Raghavendra, and V. Prasanna, "Goffish: A sub-graph centric framework for large-scale graph analytics," in *Euro-Par 2014 Parallel Processing* (F. Silva, I. Dutra, and V. Santos Costa, eds.), vol. 8632 of *Lecture Notes in Computer Science*, pp. 451–462, Springer International Publishing, 2014.

[75] P. Yuan, W. Zhang, C. Xie, H. Jin, L. Liu, and K. Lee, "Fast iterative graph computation: A path centric approach," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, (Piscataway, NJ, USA), pp. 401–412, IEEE Press, 2014.

[76] A. Buluç and J. R. Gilbert, "The combinatorial blas: Design, implementation, and applications," *Int. J. High Perform. Comput. Appl.*, vol. 25, pp. 496–509, Nov. 2011.

[77] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "Graphmat: High performance graph analytics made productive," *Proc. VLDB Endow.*, vol. 8, pp. 1214–1225, July 2015.

[78] J. Kepner, P. Aaltonen, D. Bader, A. Buluc, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, S. McMillan, J. Moreira, J. D. Owens, C. Yang, M. Zalewski, and T. Mattson, "Mathematical Foundations of the GraphBLAS," *ArXiv e-prints*, June 2016.

[79] M. J. Anderson, N. Sundaram, N. Satish, M. M. A. Patwary, T. L. Willke, and P. Dubey, "Graphpad: Optimized graph primitives for parallel and distributed platforms," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 313–322, May 2016.

[80] X. Wen, S. Zhang, and H. You, "DRONE: a Distributed gRaph cOmputiNg Engine," *arXiv e-prints*, p. arXiv:1812.04380, Dec. 2018.

[81] "The graph 500 list." http://www.graph500.org, 2011.

[82] A. Buluç, S. Beamer, K. Madduri, K. Asanović, and D. Patterson., "Distributed-memory breadth-first search on massive graphs.," in *Parallel Graph Algorithms* (D. Bader, ed.), CRC Press, 2015.

[83] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, (Los Alamitos, CA, USA), pp. 12:1–12:10, IEEE Computer Society Press, 2012.

[84] F. Checconi and F. Petrini, "Traversing trillions of edges in real time: Graph exploration on large-scale parallel machines," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pp. 425–434, May 2014.

[85] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the gpu," in *Proceedings of the 21st*

ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '16, (New York, NY, USA), pp. 11:1–11:12, ACM, 2016.

[86] T. Ben-Nun, M. Sutton, S. Pai, and K. Pingali, "Groute: An asynchronous multi-gpu programming model for irregular computations," in *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, (New York, NY, USA), pp. 235–248, ACM, 2017.

[87] A. Gharaibeh, L. Beltrão Costa, E. Santos-Neto, and M. Ripeanu, "A yoke of oxen and a thousand chickens for heavy lifting graph processing," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, (New York, NY, USA), pp. 345–354, ACM, 2012.

[88] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on gpus," in *Proceedings of the 2012 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '12, (Washington, DC, USA), pp. 141–151, IEEE Computer Society, 2012.

[89] H.-N. Tran and E. Cambria, "A survey of graph processing on graphics processing units," *J. Supercomput.*, vol. 74, pp. 2086–2115, May 2018.

[90] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient parallel graph exploration on multi-core cpu and gpu," in *2011 International Conference on Parallel Architectures and Compilation Techniques*, pp. 78–88, Oct 2011.

[91] D. Merrill, M. Garland, and A. Grimshaw, "Scalable gpu graph traversal," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, (New York, NY, USA), pp. 117–128, ACM, 2012.

[92] Y. You, S. L. Song, and D. Kerbyson, "An adaptive cross-architecture combination method for graph traversal," in *Proceedings of the 28th ACM International Con-*

ference on Supercomputing, ICS '14, (New York, NY, USA), pp. 169–169, ACM, 2014.

[93] M. Verstraaten, A. L. Varbanescu, and C. de Laat, "Using Graph Properties to Speed-up GPU-based Graph Traversal: A Model-driven Approach," *ArXiv e-prints*, Aug. 2017.

[94] K. A. Hawick, A. Leist, and D. P. Playne, "Parallel graph component labelling with gpus and cuda," *Parallel Comput.*, vol. 36, pp. 655–678, Dec. 2010.

[95] A. V. P. Grosset, P. Zhu, S. Liu, S. Venkatasubramanian, and M. Hall, "Evaluating graph coloring on gpus," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, (New York, NY, USA), pp. 297–298, ACM, 2011.

[96] A. Rungsawang and B. Manaskasemsak, "Fast pagerank computation on a gpu cluster," in *2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pp. 450–456, Feb 2012.

[97] R. Dathathri, G. Gill, L. Hoang, H.-V. Dang, A. Brooks, N. Dryden, M. Snir, and K. Pingali, "Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, (New York, NY, USA), pp. 752–768, ACM, 2018.

[98] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–13, Oct 2016.

[99] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, (New York, NY, USA), pp. 105–117, ACM, 2015.

[100] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "GraphR: Accelerating Graph Processing Using ReRAM," *ArXiv e-prints*, Aug. 2017.

[101] Harshvardhan, A. Fidel, N. M. Amato, and L. Rauchwerger, "The STAPL Parallel Graph Library," in *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pp. 46–60, Springer Berlin Heidelberg, 2012.

[102] Harshvardhan, A. Fidel, N. M. Amato, and L. Rauchwerger, "An algorithmic approach to communication reduction in parallel graph algorithms," in *Proc. Intern. Conf. Parallel Architecture and Compilation Techniques (PACT)*, PACT '15, (San Francisco, CA, USA), pp. 201–212, IEEE, 2015. Finalist for Conference Best Paper Award.

[103] G. Tanase, A. A. Buss, A. Fidel, Harshvardhan, I. Papadopoulos, O. Pearce, T. G. Smith, N. Thomas, X. Xu, N. Mourad, J. Vu, M. Bianco, N. M. Amato, and L. Rauchwerger, "The STAPL parallel container framework," in *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011*, pp. 235–246, 2011.

[104] A. A. Buss, Harshvardhan, I. Papadopoulos, O. Pearce, T. G. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger, "STAPL: standard template adaptive parallel library," in *Proceedings of of SYSTOR 2010: The 3rd Annual Haifa Experimental Systems Conference, Haifa, Israel, May 24-26, 2010*, (New York, NY, USA), pp. 1–10, ACM, 2010.

[105] I. Papadopoulos, N. Thomas, A. Fidel, N. M. Amato, and L. Rauchwerger, "STAPL-RTS: An Application Driven Runtime System," in *Proc. ACM Int. Conf. Supercomputing (ICS)*, ICS '15, (New York, NY, USA), pp. 425–434, ACM, 2015.

[106] S. Saunders and L. Rauchwerger, "ARMI: an adaptive, platform independent communication library," in *Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPoPP)*, (San Diego, California, USA), pp. 230–241, ACM, 2003.

[107] A. A. Buss, A. Fidel, Harshvardhan, T. G. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger, "The STAPL pview," in *Languages and Compilers for Parallel Computing - 23rd International Workshop, LCPC 2010, Houston, TX, USA, October 7-9, 2010. Revised Selected Papers*, pp. 261–275, 2010.

[108] D. Musser, G. Derge, and A. Saini, *STL Tutorial and Reference Guide, Second Edition*. Addison-Wesley, 2001.

[109] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web," 1998.

[110] Harshvardhan, B. West, A. Fidel, N. M. Amato, and L. Rauchwerger, "A hybrid approach to processing big data graphs on memory-restricted systems," in *Proc. International Parallel and Distributed Processing Symposium (IPDPS)*, (Hyderabad, India), pp. 799–808, IEEE, May 2015.

[111] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, Jan. 2008.

[112] R. A. Pearce, M. Gokhale, and N. M. Amato, "Multithreaded asynchronous graph traversal for in-memory and semi-external memory," in *Conference on High Performance Computing Networking, Storage and Analysis, SC 2010, New Orleans, LA, USA, November 13-19, 2010*, pp. 1–11, 2010.

[113] U. Meyer and P. Sanders, "Delta-stepping : A parallel single source shortest path algorithm," in *Proceedings of the European Symposium on Algorithms*, pp. 393–404, 1998.

[114] "Stanford large network dataset collection.." http://snap.stanford.edu/data/index.html, 2013.

[115] J. JàJà, *An Introduction Parallel Algorithms*. Reading, Massachusetts: Addison–Wesley, 1992.

[116] G. Song and N. M. Amato, "Using motion planning to study protein folding pathways," in *Proc. Int. Conf. Comput. Molecular Biology (RECOMB)*, pp. 287–296, 2001.

[117] A. P. Singh, J.-C. Latombe, and D. L. Brutlag, "A motion planning approach to flexible ligand binding," in *Int. Conf. on Intelligent Systems for Molecular Biology (ISMB)*, pp. 252–261, 1999.

[118] H. Chang and T. Y. Li, "Assembly maintainability study with motion planning," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, pp. 1012–1019, 1995.

[119] H. Choset, K. M. Lynch, S. Hutchinson, G. A. Kantor, W. Burgard, L. E. Kavraki, and S. Thrun, *Principles of Robot Motion: Theory, Algorithms, and Implementations*. Cambridge, MA: MIT Press, June 2005.

[120] L. E. Kavraki, P. Švestka, J. C. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Trans. Robot. Automat.*, vol. 12, pp. 566–580, August 1996.

[121] S. Thomas, G. Tanase, L. K. Dale, J. M. Moreira, L. Rauchwerger, and N. M. Amato, "Parallel protein folding with STAPL," *Concurrency and Computation: Practice and Experience*, vol. 17, no. 14, pp. 1643–1656, 2005.

[122] S. A. Jacobs, N. Stradford, C. Rodriguez, S. Thomas, and N. M. Amato, "A scalable distributed RRT for motion planning.," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, (Karlsruhe, Germany), pp. 5088–5095, May 2013.

[123] J. Bialkowski, S. Karaman, and E. Frazzoli, "Massively parallelizing the RRT and the RRT*," in *Proc. IEEE Int. Conf. Intel. Rob. Syst. (IROS)*, 2011.

[124] D. Devaurs, T. Simeon, and J. Cortes, "Parallelizing RRT on distributed-memory architectures," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, 2011.

[125] V. Garcia, E. Debreuve, and M. Barlaud, "Fast k nearest neighbor search using gpu," in *CVPR Workshop on Computer Vision on GPU, Anchorage, Alaska, USA*, 2008.

[126] E. Plaku and L. Kavraki, "Distributed computation of the knn graph for large high-dimensional point sets," *Journal of Parallel and Distributed Computing*, vol. 67, no. 3, 2007.

[127] J. H. Reif, "Complexity of the mover's problem and generalizations," in *Proc. IEEE Symp. Foundations of Computer Science (FOCS)*, (San Juan, Puerto Rico), pp. 421–427, October 1979.

[128] S. M. LaValle and J. J. Kuffner, "Randomized kinodynamic planning," *Int. J. Robot. Res.*, vol. 20, pp. 378–400, May 2001.

[129] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An Efficient Multithreaded Runtime System," in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, vol. 30, (New York, NY, USA), pp. 207–216, ACM, July 1995.

[130] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, pp. 720–748, Sept. 1999.

[131] G. Karypis, K. Schloegel, and V. Kumar, *ParMeTis: Parallel Graph Partitioning and Sparse Matrix Ordering Library, Version 2.0*. University of Minnesota, Dept. of Computer Science, Sept. 1999.

[132] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan, "Zoltan data management services for parallel dynamic applications," *Computing in Science and Engineering*, vol. 4, no. 2, pp. 90–97, 2002.

[133] I. Papadopoulos, *STAPL-RTS: A Runtime System for Massive Parallelism*. PhD thesis, Texas A&M University, 2016.

[134] D. Yan, J. Cheng, Y. Lu, and W. Ng, "Effective techniques for message reduction and load balancing in distributed graph computation," in *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, (Republic and Canton of Geneva, Switzerland), pp. 1307–1317, International World Wide Web Conferences Steering Committee, 2015.

[135] S. Salihoglu and J. Widom, "Gps: A graph processing system," in *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, SSDBM, (New York, NY, USA), pp. 22:1–22:12, ACM, 2013.

[136] L. Chitnis, A. Das Sarma, A. Machanavajjhala, and V. Rastogi, "Finding connected components in map-reduce in logarithmic rounds," in *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, (Washington, DC, USA), pp. 50–61, IEEE Computer Society, 2013.

[137] U. N. Raghavan, R. Albert, and S. Kumara, "Near linear time algorithm to detect community structures in large-scale networks," vol. 76, p. 036106, Sept. 2007.

[138] R. D. Peng, "Reproducible research in computational science.," *Science (New York, N.Y.)*, vol. 334, pp. 1226–7, Dec. 2011.

[139] "Artifact review and badging." https://www.acm.org/publications/policies/artifact-review-badging, 2016. Accessed: 2017-12-12.

[140] G. K. Sandve, A. Nekrutenko, J. Taylor, and E. Hovig, "Ten simple rules for reproducible computational research," *PLOS Computational Biology*, vol. 9, pp. 1–4, 10 2013.

[141] E. Dolstra, M. de Jonge, and E. Visser, "Nix: A safe and policy-free system for software deployment," in *Proceedings of the 18th USENIX Conference on System Administration*, LISA '04, (Berkeley, CA, USA), pp. 79–92, USENIX Association, 2004.

[142] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral, "The spack package manager: Bringing order to hpc software chaos," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, (New York, NY, USA), pp. 40:1–40:12, ACM, 2015.

[143] F. Chirigati, R. Rampin, D. Shasha, and J. Freire, *ReproZip: Computational reproducibility with ease*, vol. 26-June-2016, pp. 2085–2088. Association for Computing Machinery, 6 2016.

[144] I. Jimenez, M. Sevilla, N. Watkins, C. Maltzahn, J. F. Lofstead, K. Mohror, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "The popper convention: Making reproducible systems evaluation practical," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2017, Orlando / Buena Vista, FL, USA, May 29 - June 2, 2017*, pp. 1561–1570, 2017.

[145] H. Wickham, "Tidy data," *Journal of Statistical Software, Articles*, vol. 59, no. 10, pp. 1–23, 2014.

[146] "Top500." https://www.top500.org/. Accessed: 2021-10-08.

[147] D. R. Morrison, S. H. Jacobson, J. J. Sauppe, and E. C. Sewell, "Branch-and-bound algorithms," *Discret. Optim.*, vol. 19, p. 79102, Feb. 2016.

[148] A. J. Kunen, T. S. Bailey, and P. N. Brown, "Kripke - a massively parallel transport mini-app," 6 2015.

APPENDIX A

A FRAMEWORK FOR PARALLEL SOFTWARE EXPERIMENTS

One of the cornerstones of experimental science is the ability to replicate and reproduce an experiment. In computational sciences, full replication of an experiment is often prohibitively expensive and reproducibility is regarded as an "attainable minimal standard" for verification of scientific claims [138]. Reproducibility entails capturing the details of an experiment in order to allow scientists to interpret the resulting experimental artifacts, as well as for others to verify the conditions from which experimental results are produced. The Association for Computing Machinery ha proposed processes for artifact review [139] for many of its conferences and journals. Sandve et. al [140] introduce a set of common sense guidelines toward reproducible general computational research, including tracking provenance of individual results, archiving external programs used and version controlling custom scripts. Although useful as a general set of principles, there are many specific points to address when evaluating parallel software. For example, it is important to capture various aspects of the experimental evaluation such as batch submission job files, version control information for the code used to produce the executable and the resulting program output which are tracked exactly from the job output.

In this section, we introduce a framework for generating, maintaining and describing parallel software experimental artifacts, such as batch job files, raw and structured program output and provenance information.

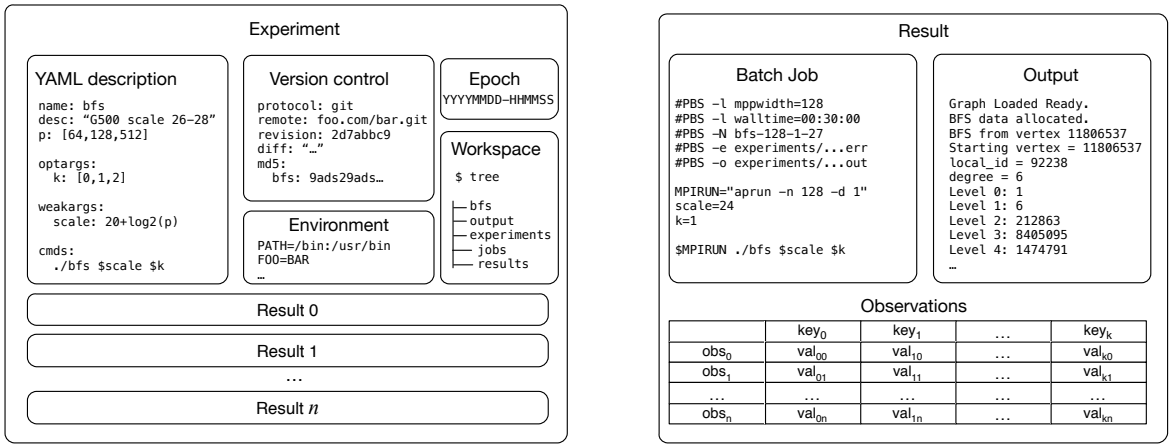
A.1 Motivation and Related Work

Much work has been devoted to recreating the environment in which a particular workload is run. Recently, containerized solutions such as Docker [64] have become the de

factor standard for solving issues with reproducing an exact environment for cloud computing scenarios. Singularity [63] attempts to bring a similar containerized solution to the high-performance computing area, where there is inherently more coupling between the container environment and the host environment, typically a supercomputer. Also in the realm of environment recreation are modern package managers such as Nix [141] and Spack [142], which provide a functional approach to describing a package's dependencies in order to precisely recreate artifacts related to binaries that are ultimately to be run. Various tools [143] exist to trace an experiment at runtime and capture its dependencies to pack and share with others. Recently, the Popper [144] effort provides a systematic and DevOps approach to recreating the environment and experiments for research papers on systems software.

Although these projects are useful for recreating the binaries that are executed for a particular experiment, they do not provide facilities to describe the parallel software workload itself, such as the set of batch job files. For example, if one wished to create a weak scaling experiment of a particular benchmark such as the Graph 500 that evaluated its performance for processor counts up to 32,768 processors in powers of two and varied input parameters in a particular pattern, it is up to the researcher themselves to create the dozens of job files that represent this experiment. Often, this workflow is achieved through the use of ad-hoc scripts, which are rarely reusable and often not included when presenting the experiment's result. Worse, version control information is often not captured at job creation time, making it difficult to associate artifacts with code. In addition, associating the output results with the job that produced the result, along with its runtime parameters, becomes error-prone with this approach.

To this end, we introduce Dimebox, a framework for the creation and submission of batch processing jobs with the following goals:



(a) (b)

Figure A.1: Organization of an (a) experiment and (b) result in Dimebox.

1. Provide a structured mechanism that improves the reproducibility of experiments conducted for parallel software research.

2. Make provenance of the results for an experiment clear by capturing crucial information about how an experiment was conducted, including version control metadata, environment variables, runtime parameters and job scripts.

3. Provide tools that facilitate common workflows for parallel software experiments (e.g., creating and running batch jobs) on supercomputers, while capturing information to help reproduce the experiment.

A.2 Experiment Organization

Parallel software experiments are organized in a structured way using Dimebox. Figure A.1 illustrates the main components that make up experiments in our framework.

```
name: bfs
desc: Breadth-first search w/ different sizes, values of k
p: [8, 16]
env:
  OMP_PROC_BIND=TRUE
optargs:
  size: [10, 20]
  k: [0, 1]
cmds:
  variant1: ./bfs1 $size $k
  variant2: ./bfs2 $size $k
```

Figure A.2: Example Dimebox experiment file describing a graph workload.

A.2.1 Experiment

The heart of Dimebox is the concept of an *experiment*. An experiment represents a single cohesive set of jobs to run and is described using a YAML file.

As an example, Listing A.2 shows an example experiment file in YAML format for a parallel graph workload that runs breadth-first search for 8 and 16 processors, while varying the size of the graph and tunable parameter `k`. Experiment files are only a description of what to run and not necessarily how to provision the resources to run it. When combined with the information of how to generate and submit jobs for a given machine (through Dimebox machine files), an experiment can generate multiple batch job files that will be submitted to be executed to the machine's batch job scheduler. Typically, there is one job created for every permutation of parameters. For Listing A.2, there will be a total of 16 jobs created, one for every processor count, executable, input size and value of `k`.

In addition, an experiment captures information about the environment and the code used to compile the executables. Figure A.1(a) provides an example of the information stored about an experiment. Each experiment is uniquely identified by an epoch, which

is a datetime formatted in `YYMMDD-HHMMSS`. Version control information is recorded, such as Git revision, branch, remote, status and changes to the working copy that are not committed. Cryptographic hashes of the executables are also included to determine if multiple experiments use the same executable. Environment variables at the time of job creation are recorded and jobs have the ability to be executed in an isolated directory (named a workspace).

A.2.2 Result

An experiment is associated with one or more results, which represent a single piece of work that is to be run, such as an executable with fixed parameters on a single processor count. In the example from Listing A.2, a job to run the `bfs1` executable on 8 cores with size 20 and $k = 1$ would be a single result.

For each result file, there is the output of the execution of the command with that particular configuration, as well as the job file that was created and submitted to obtain the output. Figure A.1(b) provides an example of a single result, which contains the original batch job and the job's output.

A.2.3 Observation

A result will contain one or more observations. An observation is used in the same sense as in the Tidy Data [145] model.

During execution, the program can emit information about observations by printing dbx.kv key: value in its output. In this example, key will form a column in the parsed output and value will be one cell of a row.

Consider the same experiment as above. Imagine that the contents of one of the results is as follows:

```
dbx.kv size: 1024
```

```
dbx.kv time: 0.214469
dbx.kv error: 0.00979
```

The entire contents of this file therefore represents a single observation. If we were to parse this experiment, we would see the following in the output:

```
cmd p k j size  time  error
foo 16  0 10  1024  0.214496  0.00979
...
```

A program may want to record multiple observations per run. In this case, the dbx.obs keyword can be used to distinguish between multiple observations. For example, consider that the updated result file contains the following content:

```
dbx.kv size: 1024
dbx.obs {"algo": "exact", "time": 0.65, "error": 0}
dbx.obs {"algo": "approx", "time": 0.21, "error": 0.00979}
```

This program ran two versions of an algorithm: an approximate version and an exact version. The exact version executed in more time with no error while the approximate version finished in less time with some error. If we parse this experiment now, we may expect to see the observations collected as follows:

```
cmd p k j size  algo  time  error
foo 16  0 10  1024  approx  0.214496  0.00979
foo 16  0 10  1024  exact 0.650415  0
...
```

Note that any freestanding key-value pair is simply appended to all observations in the result.

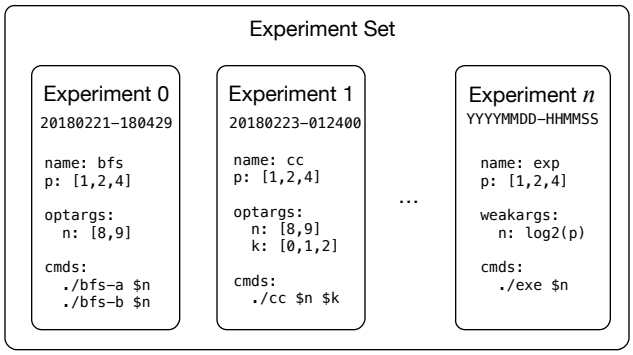


Figure A.3: Organization of a set of experiments in Dimebox.

In addition to the provided mechanisms to parse observations from results, Dimebox also includes a plugin system for parsing that can be used to scrape observations from files in any custom format. This feature is useful for running benchmarks that are already written and whose output format is fixed.

A.2.4 Experiment Set

The collection of all experiments for a project is called an experiment set. In Dimebox, an `experiments` directory directly corresponds with an experiment set. Figure A.3 provides an illustrative example of a set of n experiments. Each experiment has associated provenance metadata, including version control information, binary hashes and experimental setup configurations. In this manner, one can easily identify the exact configuration for every experiment in in Dimebox.

A.3 Job Submission Workflow

A researcher tasked with running experimental results on a supercomputer will often create job files specific to that supercomputer's batch processing system. Without Dimebox, these job files will be created either by hand or the researcher will create an ad-hoc script that generates a single job file for a configuration of the experiment. For example,

```
$ vim my_experiment.yml
$ dimebox generate -m vulcan my_experiment.yml
20190212-182122
$ dimebox submit 20190212-182122
$ dimebox parse 20190212-182122
```

Figure A.4: An example Dimebox workflow to run an experiment.

there will be one job for each processor count.

This process is prone to many types of errors. The researcher could forget to include jobs for particular processor counts. Output for the program can be directed to files with misleading names, obscuring the provenance of results. The researcher could forget to update where results are stored when a new version of the program is to be tested. With Dimebox, many of these types of errors can be mitigated.

Instead, users of Dimebox will first create a single YAML file that describes *what* the experiment is in a declarative manner, rather then describing *how* to run the experiment in the common procedural way. This YAML file will then be used to generate job files for the particular supercomputer that the researcher is using. This process is less error-prone than the manual method, as there is a clear transformation from the user's YAML file to a set of job files that is based on machine-specific specifications of job files.

Once these job files are created, jobs can be submitted to the batch system. Figure A.4 provides a typical workflow for the tool. The generation and submission procedure is illustrated in lines 1-4 of the figure through the `generate` and `submit` subcommands of Dimebox. Once the jobs have completed, their results can be parsed using the `parse` subcommand that applies either a generic regular-expression based parser to the job output or a custom parsing script. Once parsed, the results can be analyzed or visualized in any typical data processing pipeline (e.g., R, Python, Excel, etc.).

A.4 Conclusion

The framework for parallel software experiments described in this appendix, as well as the accompanying software tool Dimebox that realizes this framework, helps with experiment reproduction and organization, provides provenance of raw data and allows users to easily perform common tasks related to running experiments. Dimebox as a software tool has been used by over a dozen developers and researchers in the Parasol Lab at Texas A&M University. We provide Dimebox as an open source tool hosted on GitHub[1] under an MIT License.

[1] Dimebox codebase: `https://github.com/ledif/dimebox`

APPENDIX B

NESTED PARALLELISM FOR HIERARCHICAL GRAPH ALGORITHMS

Many modern architectures for parallel machines are designed in a hierarchical manner. For example, the most powerful supercomputer at the time of this writing is Fugaku according to the Top 500 [146]. This system is composed of racks, where each rack contains shelves, each shelf contains a *Bunch of Blades* (BoB) and each BoB consists of 16 CPUs. This deep hierarchy introduces strongly non-uniform latency between CPUs and any application running on this machine would be losing out on performance if work and communication is scheduled in a manner that is agnostic to the machine's hierarchy.

In this appendix, we introduce a technique for expressing graph algorithms that are naturally hierarchical in such a way that nested parallelism can be applied to the hierarchy to achieve performance that is in line with the machine's natural hierarchical composition. By expressing a graph algorithm in this manner, we would achieve the following benefits. Spatial locality would increase due to subgraphs representing elements in the same level of the machine. Additionally, every level of the hierarchy can employ a different algorithm that is tailored specifically for the type of parallelism at that the level. For example, if a subgraph contains vertices that are all on the same shared-memory node, we could seamlessly employ an efficient thread-parallel algorithm. This is possible if the hierarchical algorithm is expressed in a recursive manner. One additional benefit of this technique is that it allows for algorithm-driven aggregation of messages across node. Instead of sending point-to-point messages through edges that cross different levels of the hierarchy, we can instead aggregate messages in algorithm-specific ways and send the aggregated message across. Note that this optimization not only saves on point-to-point latency but also reduces the total amount of traffic that crosses the interconnect. Finally, there is a class

of hierarchical graph algorithms designed around pruning entire computations by ignoring certain subgraphs (e.g., branch-and-bound [147] style algorithms).

We will be exploring this proposed technique using a case study of a parallel wavefront computation outlined in the Kripke benchmark [148].

B.1 Parallel Wavefront Computation

Kripke is 3D deterministic particle transport code that solves the linear Boltzmann equation specifically for 3D radiation transport. The goal of the Kripke benchmark is to study how different parallelization schemes and data layouts affect the performance of a large-scale parallel wavefront computation called *sweep*. The benchmark uses a structured decomposition of the domain into cells and the computation can be seen as simultaneous traversals from the corners of the spatial domain. For non-regular decompositions of the domain, it is necessary to express the core algorithm in graph-theoretic terms where the decomposed cells are vertices and spatially adjacent cells are connected through edges in a graph.

First, the original spatial domain is decomposed into an arbitrary graph representation. Each vertex of this graph is associated with the following properties: its 3D coordinate in space, a collection indexed by sweep direction of floats used to store physical properties of the sweep, and another indexed collection of floats representing incoming values from neighbors. Each edge in the graph has the 3D coordinate of that edge in space.

Initially, each vertex stores the initial physical properties of the domain. Then, simultaneous graph traversals in different directions are started from the source vertices that represent "corners" of the spatial domain. These source vertices are direction-specific and are defined as a vertex that has no incident edges along a given direction. The graph traversals behave similarly to a traditional parallel breadth-first search algorithm; vertices are visited in breadth-first order and each vertex receives values from its incoming neighbors. These

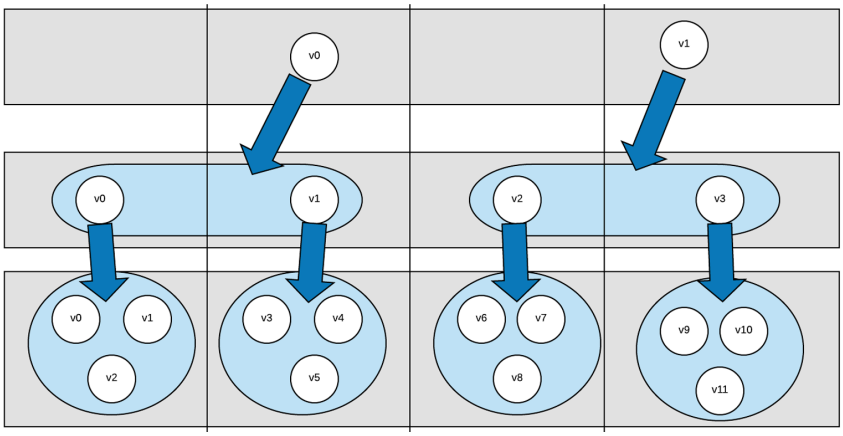


Figure B.1: Example decomposition of original spatial domain into a multi-level hierarchical graph

values are then aggregated in a physically meaningful way and then propagated further to the vertex's out edges.

Because this sweep algorithm is very similar to breadth-first search, many of the techniques outlined in the previous chapters of this dissertation can apply here as well, including k-level-asynchronous processing and nested parallel visitation.

B.1.1 Technique

The arbitrary sweep algorithm described above can benefit from a nested hierarchical representation. The main idea is to subdivide the original graph, which represents a mesh of the spatial domain, into recursive and hierarchical subgraphs. Each subgraph represents a disjoint partition of the original graph. Vertices in the same subgraph share the same locality (e.g., on the same shared-memory node). Edges within a subgraph represent intra-locality communication whereas edges that connect subgraphs will induce inter-locality communication. The original graph can be partitioned into subgraphs recursively up to n levels.

For example, in Figure B.1, the original graph is represented in the bottom-most level.

The second level contains subgraphs which group together vertices in the bottom level that are on the same processor. Finally, the vertices on the top level represent subgraphs of vertices on the second level. For instance, this particular partitioning of the graph may represent subgraphs that are not on the same processor, but on the same shared-memory node.

The nested hierarchical sweep algorithm starts from the top-most graph, which is the most coarse graph. It performs a sweep on this graph from each source. When encountering a vertex, it behaves differently based on whether the vertex represents a subgraph or if it is scalar. If it is scalar, it computes the fine-grained diamond-difference function representing the main mathematical computation of the benchmark. If instead the vertex represents a subgraph, the algorithm will perform a nested sweep on that subgraph and aggregate the resulting values.

B.2 Preliminary Implementation

We provide a sample implementation of the nested sweep algorithm in the STAPL Graph Library. This implementation uses the k-level-asynchronous paradigm to implement the traversal of the subgraphs in a parametrically asynchronous manner.

The current implementation has a limitation that the partitioned subgraphs are all contained on the same processor. In this manner, it is possible to swap the nested parallel algorithm for an efficient serial algorithm to compute the inner sweep.