SCALABLE PARALLEL ALGORITHMS FOR MASSIVE SCALE-FREE GRAPHS

A Dissertation

by

ROGER ALLAN PEARCE

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

DOCTOR OF PHILOSOPHY

Chair of Committee,	Nancy M. Amato
Committee Members,	Yoonsuck Choe
	Lawrence Rauchwerger
	Marvin L. Adams
	Maya Gokhale
Head of Department,	Nancy M. Amato

December 2013

Major Subject: Computer Science

ABSTRACT

Efficiently storing and processing massive graph data sets is a challenging problem as researchers seek to leverage "Big Data" to answer next-generation scientific questions. New techniques are required to process large scale-free graphs in shared, distributed, and external memory. This dissertation develops new techniques to parallelize the storage, computation, and communication for scale-free graphs with high-degree vertices. Our work facilitates the processing of large real-world graph datasets through the development of parallel algorithms and tools that scale to large computational and memory resources, overcoming challenges not addressed by existing techniques. Our aim is to scale to trillions of edges, and our research is targeted at leadership class supercomputers, clusters with local non-volatile memory, and shared memory systems.

We present three novel techniques to address scaling challenges in processing large scale-free graphs. We apply an asynchronous graph traversal technique using prioritized visitor queues that is capable of tolerating data latencies to the external graph storage media and message passing communication. To accommodate large high-degree vertices, we present an edge list partitioning technique that evenly partitions graphs containing high-degree vertices. Finally, we propose a technique we call distributed delegates that distributes and parallelizes the storage, computation, and communication when processing high-degree vertices. The edges of high-degree vertices are distributed, providing additional opportunities for parallelism not present in existing methods.

We apply our techniques to multiple graph algorithms: Breadth-First Search, Single Source Shortest Path, Connected Components, K-Core decomposition, Trian-

gle Counting, and Page Rank. Our experimental study of these algorithms demonstrates excellent scalability on supercomputers, clusters with non-volatile memory, and shared memory systems. Our study includes multiple synthetic scale-free graph models, the largest of which has trillion edges, and real-world input graphs. On a supercomputer, we demonstrate scalability up to 131K processors, and improve the best known Graph500 results for IBM BG/P Intrepid by 15%.

DEDICATION

To my parents, for instilling in me the value of education.

To Olga, for your constant support.

To Zhanna, for showing me the joys of a child's curiosity.

ACKNOWLEDGEMENTS

I feel fortunate to have many supportive people who have helped me throughout this work. I would like to thank my advisor, Dr. Nancy M. Amato, for her continual guidance during my graduate and undergraduate studies. She provided an environment where I could explore many different research interests.

I would like to thank my committee members, Dr. Yoonsuck Choe, Dr. Lawrence Rauchwerger, Dr. Marvin Adams, and Dr. Maya Gokhale, for their guidance and suggestions throughout this work.

I would like to thank many people at Lawrence Livermore National Laboratory, where I was a student intern and Lawrence Scholar. Maya Gokhale provided significant guidance on my research and future career paths. I would also like to thank some of my other collaborators, in particular, Dr. Sasha Ames, Dr. Brian Van Essen, Dr. Scott Lloyd, Dr. Craig Ulmer, Dr. Andy Yoo, and Dr. John May.

I would also like to thank the members of the Parasol Lab. Dr. Jinsuck Kim, Dr. Marco Morales, and Dr. Jyh-Ming Lien, were my graduate student mentors as I was starting my research as an undergrad and graduate student. I would also like to thank some of my other collaborators, in particular, Dr. Sam Rodriquez, Olga Pearce, Dr. Shawna Thomas, Dr. Lydia Tapia, Dr. Nathan Thomas, Dr. Timmie Smith, Aimee Vargas, Dr. Xinyu Tang, Sam Jacobs. These students formed the backbone of a strong research environment, and encouraged open collaboration.

Finally, I would like to thank my family for their constant support.

This work was partially performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-TH-645698).

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

A *graph* is a powerful tool that can represent a set of objects and their relationships. Graphs are used in a wide range of fields including Computer Science, Biology, Chemistry, and the Social Sciences. These graphs, sometimes known as *networks*, may represent complex relationships between individuals, proteins, chemical compounds, etc. In a graph, relationships are stored using vertices and edges; a vertex may represent an object or concept, and the relationships between them are represented by edges. The power of the graph data structure lies in the ability to encode complex relationships between data and provide a framework to analyze the impact of the relationships.

Efficiently storing and processing large amounts of graph data is a challenging and growing problem as researchers seek to leverage "Big Data" to answer next-generation scientific questions. Many real-world graphs can be classified as *scale-free*, where the distribution of vertex degrees follows a scale-free power-law [6]. The degree of a vertex is the count of the number of edges connecting to the vertex. A power-law vertex degree distribution means that the majority of vertices have small degree, while a select few have a very large degree, with the distribution of the degrees following a power-law distribution. These high-degree vertices are called *hub* vertices. Hubs have the potential to create scaling issues for parallel and distributed algorithms, such as load imbalance and communication bottlenecks, because the processing requirements for a hub are significantly larger than for an average vertex.

This research develops new techniques to distribute and parallelize the storage, computation, and communication of high-degree vertices in scale-free graphs. Our work facilitates the processing of large real-world graph datasets through the devel-

opment of parallel algorithms and tools that scale to large computational and memory resources, overcoming challenges not addressed by existing techniques. Towards this goal, we begin by identifying key challenges to storing and processing massive scale-free graphs. Many important graph datasets have unstructured and irregular topologies that perform poorly using multi-level memory hierarchies, including external memory. Irregular topologies and high-degree vertices often produce excessive processor to processor, approaching all-to-all, communication when algorithms are parallelized, leading to poor overall performance. These challenges are discussed in depth in Chapter 2.

Many parallel graph algorithms operate on graphs that are partitioned amongst a set of processors, and each processor is assigned a subset of the graph. The graph partitioning problem is to subdivide the vertices and edges of a graph into roughly equal sized groups, while minimizing the number of edges connecting vertices of different groups. The groups or partitions should be of roughly equal size to balance the computation for each processor. Additionally, minimizing the number of edges connecting vertices of different groups reduces the amount of inter-processor communication and coordination required by graph algorithms. Graph partitioning is challenging for many graphs, and is known to be NP-Complete [14]. Without good graph separators, parallel algorithms will require significant communication.

Partitioning many scale-free graphs is difficult, and often not feasible, due to their irregular topology and high-degree vertices. The simplest partitioning is called 1D or row-wise, in which the vertices of the graph are partitioned and all edges adjacent to a vertex, including imbalanced hubs, are assigned to a single partition. For scale-free graphs, the partitions to which high-degree vertices are assigned may contain significantly more edges than the average partition. This edge partition imbalance effects the data storage, computation, and communication costs, because

the processors will store and process an uneven number of edges. Current state-of-the-art partitioning for sparse scale-free graphs into p partitions uses a 2D strategy that partitions high-degree vertices across $O(\sqrt{(}p))$ partitions. 2D partitioning also suffers from storage, computation, and communication imbalances.

We address these challenges by providing three novel techniques for processing large scale-free graphs. First, we develop an asynchronous graph traversal technique using visitor queues that is capable of expressing fine-grained parallelism at the individual vertex level. Data latencies associated with the external graph storage media and message passing communication are mitigated by the asynchrony of the computation.

Second, we introduce a new partitioning technique that guarantees balanced partitions containing high-degree vertices. Previous partitioning strategies using 1D and 2D partitioning may produce an imbalanced number of edges per partition for scale-free graphs. Our edge list partitioning approach partitions the graph's edges such that each partition contains an equal number of edges, overcoming the storage balance issues created by high-degree vertices.

Finally, we develop a technique we call distributed delegates to parallelize and distribute the storage, computation, and communication of high-degree vertices. We make a distinction between low and high degree vertices, and distribute the high-degree vertices. The number of edges per partition is balanced, and the large amount of computation and communication for the high-degree vertices is distributed over the processors, leading to significantly improved performance.

Our techniques provide new tools to analyze large scale-free graph datasets on a wide rage of data-intensive computational resources. Our research is targeted at leadership class supercomputers containing significant distributed memory resources, clusters with node-local non-volatile random access memory (NVRAM), and small

shared-memory systems containing large NVRAM storage devices. Our work breaks new ground for using NVRAM in the high-performance computing (HPC) environment for data intensive applications. We show that by leveraging distributed NVRAM, significantly larger datasets can be processed with only moderate performance degradation. We show that by exploiting both distributed memory processing and node-local NVRAM, significantly larger datasets can be processed than with either approach in isolation. Further, we demonstrate that our asynchronous approach mitigates the effects of both distributed and external memory latency. The architecture and configuration of NVRAM in supercomputing clusters is an active research topic. To our knowledge, our work is the first to integrate node-local NVRAM with distributed memory at extreme scale for important data intensive problems, helping to inform the design of future architectures.

1.1 Research Objective and Contributions

The research contributions of this dissertation can be summarized as:

- Novel algorithmic techniques to process large scale-free graphs:
  - An asynchronous computation model using prioritized visitor queues that tolerates latencies associated with external memory and distributed message passing;
  - An edge list partitioning technique that guarantees balanced partitions for scale-free graphs containing high-degree vertices;
  - A technique we call distributed delegates to parallelize and distribute the storage, computation, and communication of high-degree vertices;
- The application of these techniques to a variety of parallel graph algorithms

including: Single Source Shortest Path, Connected Components, K-Core decomposition, Triangle Counting, and PageRank;

- Experimental results demonstrate the scalability of algorithms using our techniques on leadership class supercomputers on up to 131K processors;

- Results that show that by leveraging node-local NAND Flash, algorithms using our techniques can process larger datasets with only modest performance degradation over a DRAM-only solution.

Portions of our research were previously published and are currently under review. The asynchronous visitor computation model and an initial evaluation in shared and external memory was published at the *International Conference for High Performance Computing, Networking, Storage and Analysis* (SC) 2010 [59], presented here in Chapter 3. This work led to two external memory experiments featured on the Graph500, including a 7^{th} place ranking on the June 2011 list, and was also used as a data-intensive benchmark by Van Essen, et al. [25]. Our technique for partitioning graphs containing high-degree vertices was published at the *International Parallel and Distributed Processing Symposium* (IPDPS) 2013 [60], presented here in Chapter 4. This work led to two distributed-external memory experiments featured on the Graph500 on the June 2012 list. Finally, our approach for parallelizing the storage, computation, and communication of high-degree vertices is under review [58], presented here in Chapter 5.

1.2 Outline

This dissertation describes our asynchronous framework for traversing massive scale-free graphs in shared, distributed and semi-external memories. Chapter 2 describes many of the fundamental properties of real-world graphs, along with previous

work that is related to our asynchronous algorithms. Section 2.6 discusses the challenges of processing large scale-free graphs. Chapter 3 introduces our asynchronous visitor computation model and an experimental study using shared-memory and semi-external memory systems. Chapter 4 discusses partitioning graphs containing high-degree hub vertices and an experimental study using distributed-memory systems. Finally, Chapter 5 discusses distributing the storage, computation, and communication of high-degree hub vertices and an experimental study using distributed-memory systems.

2. PRELIMINARIES AND RELATED WORK

In this chapter, we cover background topics and related work that will be referred to throughout the remainder of this dissertation. An introduction to graph terminology and representations is discussed in Section 2.1. An overview of graph partitioning is discussed in Section 2.6.1. Scale-free graphs and models of real-world graphs are discussed in Section 2.3. An overview of related work on the parallel processing of graphs is discussed in Section 2.5. The challenges associated with processing scale-free graphs is discussed in Section 2.6. Finally, an introduction to the graph algorithms and analytics that we investigated using our scaling techniques is discussed in Section 2.7.

2.1 Terminology and Graph Representations

A graph $G(V, E)$ is composed of a set of vertices V and a set of edges E, where each edge $e = (u, v), e \in E$ connects a pair of vertices, $u, v \in V$. Vertices and edges may contain weights, or other forms of meta-data. The degree of a vertex is the count of the number of edges connecting to the vertex.

Three common data structures used to represent a graph are an adjacency list, an adjacency matrix, and a compressed sparse row. The features of these structures are shown in Table 2.1.

2.2 Graph Partitioning

Many parallel graph algorithms operate on graphs that have been partitioned amongst a set of processors, and each processor is assigned a subset of the graph. The graph partitioning problem is to subdivide the vertices and edges of a graph into roughly equal sized groups, while minimizing the number of edges connecting

Name	Description	Storage Cost
Adjacency List	A list of edge targets stored for each source vertex.	$O(\|V\| + \|E\|)$
Adjacency Matrix	A $\|V\|x\|V\|$ matrix where entry $A[i, j] = 1$ iff edge (i, j) exists. Entries may also contain edge weights.	$O(\|V\|^2)$
Compressed Sparse Row (CSR)	A concatenated array of adjacency lists, with a source vertex look index.	$O(\|V\| + \|E\|)$

Table 2.1: Graph Data Structures

vertices of different groups. The groups or partitions should be of roughly equal size, to balance the cost of computation for each processor. Minimizing the number of edges connecting vertices of different groups reduces the amount of communication and coordination required by the processors. Edges connecting vertices of different partitions are commonly called *cut edges*.

Graph partitioning is challenging for many graphs, and is known to be NP-Complete [14]. Numerous heuristic techniques and libraries have been developed to partitioning graphs approximately. Some of the most successful heuristics are based on hierarchical multilevel techniques, which have been included in libraries such as *Chaco* [34], *Metis* [36], *Party* [52], *Scotch* [62], *KaFFPa* [65], and *Jostle* [76]. Parallel and distributed versions of many of these libraries have also been developed [37, 42, 61, 66, 77].

Currently, there are two approaches to partitioning a graph with the goal of evenly distributing the graph, 1D and 2D. The techniques do not attempt to minimize edge cuts, the number of edges between vertices in different partitions.

2.2.1 1D Partitioning

A simple way to partition a graph among p processors is to evenly partition the vertices and their associated adjacency list into p partitions. This style of partitioning

is called 1D or *row-wise*, and is illustrated using an adjacency matrix in Figure 2.1. In the figure, high-degree vertices in the graph form dense rows in the adjacency matrix. Because the adjacency lists or rows of high-degree vertices are fully contained by a single partition, the partitions may become significantly imbalanced. In this example, the highlighted partitions p_2 and p_5 contain more edges (non zeros) that the average due to the high-degree vertices contained.

When a graph is 1D partitioned into a set of P partitions, and $\max_{p_i \in P}(\sum_{v \in V_{p_i}} degree(v)) > \frac{|E|}{|P|}$, then at least one processor will process more than its fair share of edges.

2.2.2 2D Partitioning

Recent work using 2D graph partitioning has shown the best results for traditional large scale HPC systems [16, 83, 82]. In 2D partitioning, the graph is partitioned according to a checkerboard pattern of the graph's adjacency matrix, as illustrated in Figure 2.2. The adjacency lists of high-degree vertices are split over $O(\sqrt{(p)})$ partitions, which greatly improves the partition balance.

Unfortunately, 2D partitioning has serious disadvantages at scale and with external memory. First, when processing sparse graphs, each 2D block may become hypersparse, i.e., fewer edges than vertices per partition [15]. Specifically, partitions become hypersparse when $O(\sqrt{p}) > degree(g)$, where p is the number of distributed partitions and g is the graph. In Figure 2.2, the highlighted partitions (p_{13}, p_{14}, p_{15}) illustrate hypersparse partitions, where there are fewer edges than vertices. For the sparse Graph500 datasets with average degree of 16, this may occur for as low as 256 partitions and is independent of graph size. Second, under weak scaling, the amount of algorithm state (e.g., Breadth-First Search level) stored per partition scales with $O(\frac{V}{\sqrt{p}})$, where V is the total number of vertices. This can ultimately hit a scaling

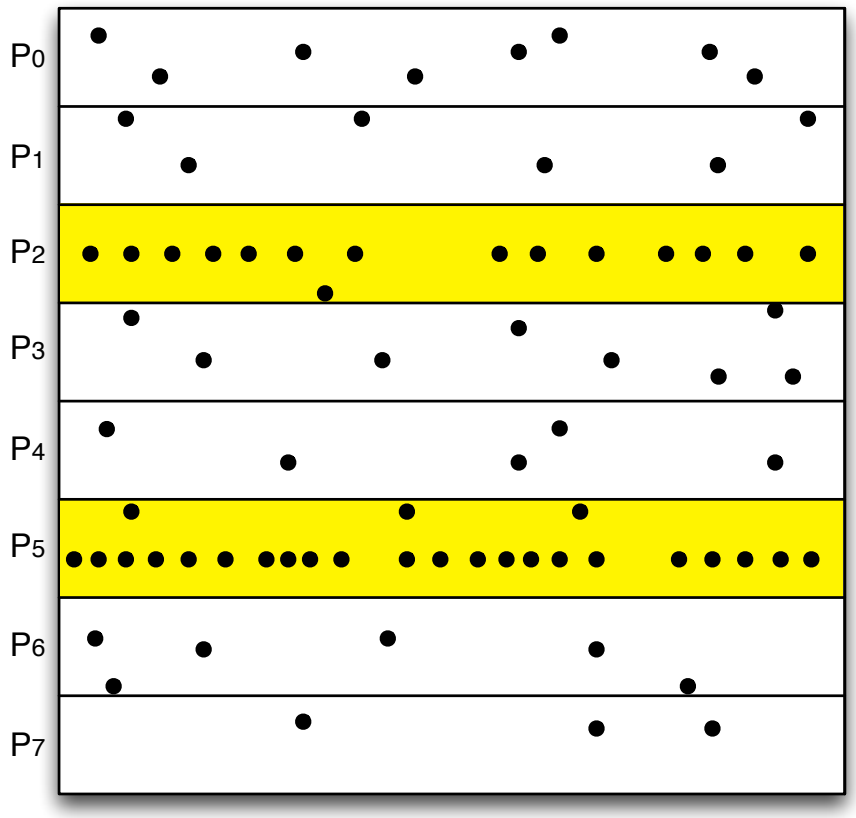


Figure 2.1: Illustration of 1D partitioning a graph's adjacency matrix. High-degree vertices form dense rows that when 1D partitioned row-wise create imbalance. The highlighted partitions p_2 and p_5 contain more edges (non zeros) than average due to the high-degree vertices contained.

wall where the amount of local algorithm state per partition exceeds the capacity of the compute node.

Finally, with respect to our desire to use semi-external memory where the vertex set is stored in in-memory and the edge set is stored in external memory, hypersparse partitions are poor candidates to apply semi-external memory techniques, because the in-memory requirements (proportional to the number of vertices) are larger than the external memory requirements (proportional to the number of edges).

Our partitioning techniques, discussed in Chapters 4 and 5, address the shortcomings of 1D and 2D partitioning for scale-free graphs containing high-degree vertices by creating balanced partitions.

2.3 Scale-free Graphs

Many real-world graphs can be classified as *scale-free*, where the distribution of vertex degrees follows a scale-free power-law [6]. A power-law vertex degree distribution means that the majority of vertices have small degree, while a select few have a very large degree, with the distribution of degrees following a power-law distribution. These high-degree vertices are called *hub* vertices, and create multiple scaling issues for parallel algorithms, discussed further in chapter 2.6.

2.3.1 Properties

Power Law. A common property of many real world graphs is a power law distribution of vertex degree. As an example, Figure 2.3 shows power law distributions of vertex degree for a web graph [43] and the *Epinions* graph [64]. An effect of the power law degree distribution is that while the vast majority of vertices have a low degree, a select few vertices will have a very high degree. These high degree nodes are often referred to as *hub* vertices, and can lead to significant load imbalance for

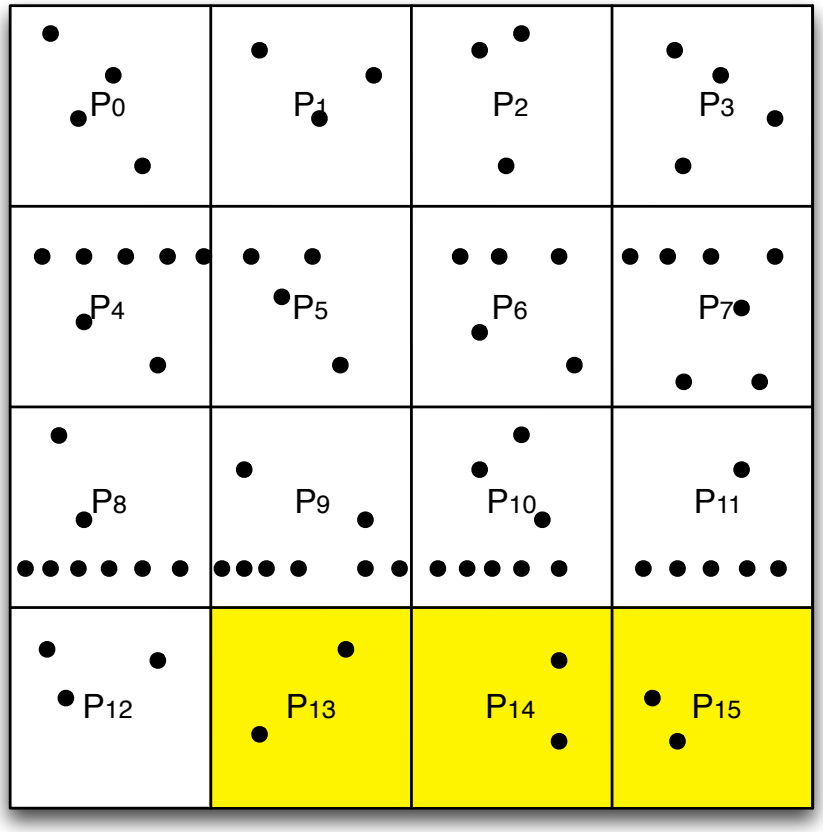
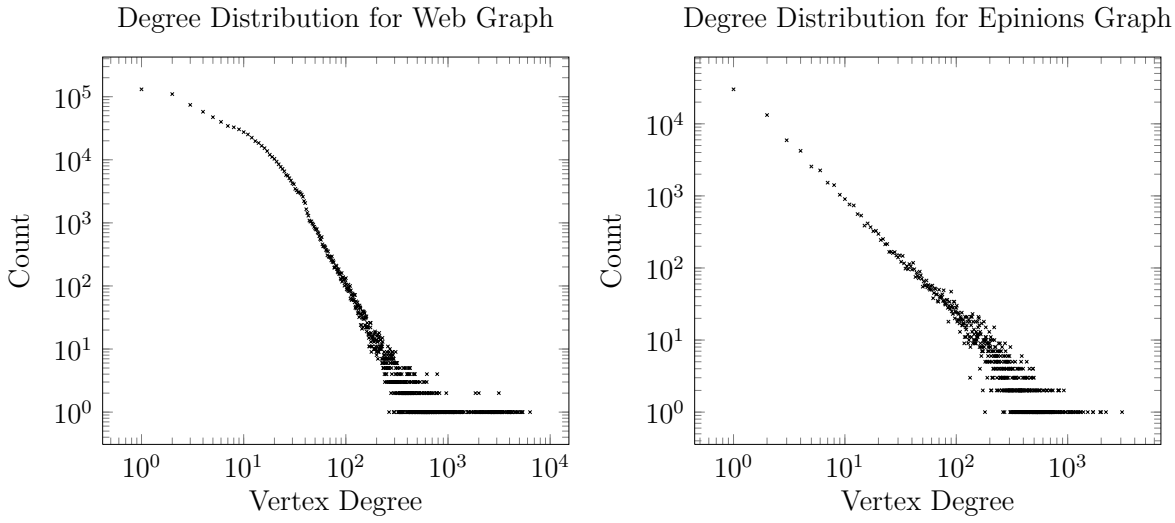


Figure 2.2: Illustration of 2D partitioning a graph's adjacency matrix. The adjacency lists of high-degree vertices are split over $O(\sqrt{(}p))$ partitions, which greatly improves the partition balance. Highlighted partitions (p_{13}, p_{14}, p_{15}) illustrate hypersparse partitions, where there are fewer edges than vertices.



(a) (b)

Figure 2.3: Vertex degree distributions for a web graph [43] (a) and the Epinions graph [64] (b).

parallel processing, discussed in Section 2.6.1.

Small Diameter. Although sparse, many graphs are connected into connected components with small diameters. The diameter of a graph is the longest shortest-path between two vertices. This property leads to a high-level of interconnectedness that has been popularized by the phrase "six degrees of Kevin Bacon."

2.3.2 Examples

Examples of scale-free graphs come from many important domains, including the world wide web, social networks and intelligence networks.

World Wide Web (WWW) graph. Graphs that model the structure of the web often consist of vertices representing webpages and directed edges representing the hyperlinks between the webpages. The web graph has been the focus of numerous

studies aimed at detecting community structures and improving applications such as web search [39, 13, 43]. The vertex degree distribution of a web graph is shown in Figure 2.3 (a).

Social Networks. Graphs naturally model the relationships established by social interactions. These interactions could be on-line friendships, call-networks, etc. Zachary [84] performed a now famous social network study in the 1970s, where he constructed a network of friendships among members of a karate club. The network was constructed by direct observation of 34 members' interactions. Modern social networking tools like *Facebook* and *Twitter* have dramatically increased the scale of social network data to hundreds of millions of individuals. These tools and their accompanying social networks have garnered the attention of many social and network scientists [40, 23, 63, 73]. The vertex degree distribution of a social network is shown in Figure 2.3 (b).

Homeland Security. It has been estimated that graphs of interest to the Department of Homeland Security will reach 10^{15} entities [38], providing a significant challenge to analysts who wish to search them. The sheer size of this data dwarfs the main-memory capacities of modern supercomputers, necessitating the use of external memory devices.

2.4 Synthetic Graph Models

Throughout this work, we used three synthetic graph generators for our studies. Two of the generators, R-MAT and Preferential Attachment, generate scale-free graphs, while the Watts-Strogatz model generates small world graphs with small diameters. This section describes the models and their parameters used in our ex-

perimental studies.

2.4.1 Scale-Free Models

Chakrabarti et al. [18] introduced the R-MAT model that generates power law vertex degree distributions. It is based on a recursive matrix model, and uses four parameters, $\{a, b, c, d\}$ where $a+b+c+d = 1$, to control how the matrix is recursively subdivided. We follow the Graph500 V1.2 specification for generator parameters [20]. R-MAT has been studied analytically and experimentally [68, 30], and has become the de facto standard scale-free graph generator model in large part due to its scalability to large graph scales. We used the open source RMAT implementation provided by the Boost Graph Library [71].

The Barabási-Albert model generates scale-free graphs based on preferential attachment [7]. The model simulates the growth of networks where the probability of a new vertex attaching to an existing vertex is proportional to the vertex's degree. We used a generalized PA model by Móri [54], where the probability of connecting to a vertex of degree d is proportional to $d + \beta$, where $\beta > -16$. By varying the value of β, we can control the rate in which hubs grow. For our studies, we chose β values of *-12, -13,* and *-14*. The β value of *-12* was used to roughly match Graph500 RMAT's hub growth, and the β values of *-13* and *-14* where chosen to increase hub growth and stress the delegate approach. We parallelize the generation of large PA graphs using similar techniques developed by Machta [46].

The growth of the largest hub vertex for the R-MAT and PA graph models is shown in Figure 2.4.

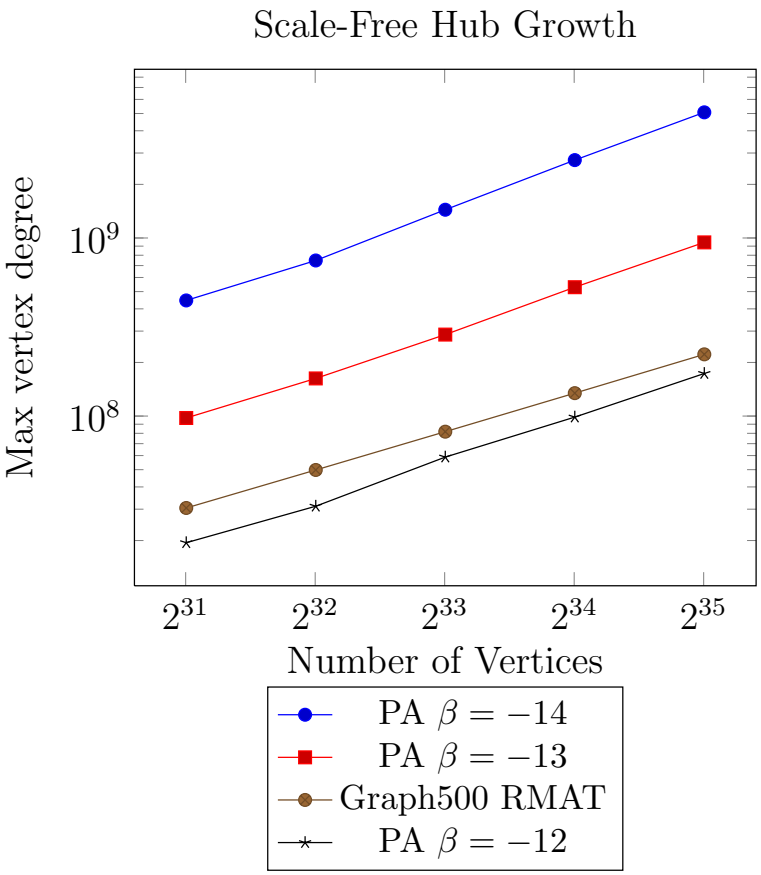


Figure 2.4: Hub growth for scale-free RMAT and preferential attachment graphs.

2.4.2 Small World Models

The Watts-Strogatz [78] model does not generate scale-free graphs, however it models the small world effect of small diameter. The model has a control parameter, β, that interpolates between a ring lattice and an Erdös-Rényi [24] random graph. The interpolating parameter β allows the number of triangles and the graph's diameter to be controlled during experiments.

2.5 Processing Large Graphs

Processing of large graphs is receiving increasing attention by the HPC community as datasets quickly grow past the capacity of commodity workstations. Significant challenges arise for traditional HPC solutions because of the nature of these datasets. These challenges can be categorized into unstructured memory access and poor data locality [33, 45].

While graph algorithms have received tremendous attention for the RAM computational model, many realistic datasets are too large to fit in the memory of a single computer. To address this, researchers have explored using Distributed Memory and External Memory. Key challenges in processing large graphs come from the non-contiguous access to the data structure. Distributed Memory approaches suffer from poor load balancing due to the intrinsic nature of power-law distributions, discussed in Section 2.6.1. External Memory approaches suffer the same issues as distributed memory, and in addition, poor data locality and unstructured memory accesses lead to poor performance for which techniques such as prefetching, blocking, and pipelining generally provide little improvement. Some experiments have shown that BFS designed for the RAM computation model runs orders of magnitude slower when forced to use external memory [2].

2.5.1 Distributed Memory

A popular approach to graph processing in HPC has been to use Distributed Memory computer clusters. Such clusters distribute the graph data amongst its processors and memory and process the graph by exchanging messages during computation phases. This approach works well when the graph exhibits nice load balancing properties (regular or uniformly random) [83] but suffers from significant load imbalance when processing power-law graphs [29].

The most common approach for implementing graph algorithms in distributed memory is with the Bulk-Synchronous Parallel (BSP) model [72]. In BSP, processors iteratively work on their local data, and then participate in collective communication operations. This type of approach is susceptible to load imbalance, because each BSP step waits for the slowest processor with the largest load. We avoid BSP in our work, and use an asynchronous approach that can exploit fine-grained parallelism.

Many distributed memory graph libraries have been developed, including the Parallel Boost Graph Library (PBGL), the STAPL Graph Library, and Pregel. PBGL [29] applies the paradigm of generic programming to the domain of graph computations. It supports distributed memory through a bulk-synchronous message passing communication. The STAPL Graph Library [32] provides a framework that abstracts the user from data-distribution and parallelism and supports the expression of asynchronous algorithms. The Pregel graph library [47] provides a vertex-centric visitor model for implementing graph algorithms. The library provides a bulk-synchronous computation model for the vertex visitors.

2.5.2 Multithreaded Shared Memory

Massive Multithreaded machines address the challenges of unstructured memory accesses and poor data locality by using little or no memory hierarchy. The

Cray XMT has been successful at processing large graph datasets; these specialized supercomputers rely on massive multithreading to mask memory latency without using complex memory caches. The development of the Multithreaded Graph Library (MTGL) for this specialized computing platform has been shown to address many of the issues related to memory latency [9]. Our approach addresses the memory latency issues using commodity hardware and storage devices (NAND Flash) that are relatively slow compared with main memory. Small-world Network Analysis and Partitioning (SNAP) [5] is another parallel graph library for shared memory which utilizes OpenMP for parallelism.

2.5.3 External Memory

Many real world graphs are too large to fit into main memory of modern computers, necessitating the use of external storage devices such as disk. Due to the significant difference in access times between main memory and disk, many efficient in-memory algorithms become impractical when using external storage. To analyze the I/O complexity of algorithms using external storage, the Parallel Disk Model (PDM) [75] has been developed. PDM's main parameters are N (problem size), M (size of internal memory), B (block transfer size), D (number of independent disks), and P (number of CPUs). When designing I/O efficient algorithms, the key principles are *locality of reference* and *parallel disk access*. For an in-depth survey of EM Algorithms, see [74].

In this dissertation, we are interested in graphs in a Semi-External Memory (SEM) scenario. A graph is semi-external if there is enough main memory to store algorithmic information about the vertices but not edges. In our SEM work, the full graph structure is stored on the persistent storage device, and the visitor queues and the output of the algorithms are stored in main memory.

Emerging technologies in persistent data storage are changing the way External Memory algorithms are designed. Flash memory is a form of non-volatile random access memory (NVRAM) that has become a commodity product through widespread use in digital cameras, music players, phones, USB drives, etc. An overview of the characteristics and performance of flash memory (namely NAND Flash) with respect to algorithmic research is given in [1, 3]. The key differences from traditional rotating media can be summarized as follows.

- Significantly faster random access time than disk (microseconds instead of 10's of milliseconds).
- Asymmetric read/write performance (writes are more costly than reads).

An important characteristic of NAND Flash devices not covered by [1, 3] is the ability to service multiple concurrent I/O requests. To achieve maximum random I/O performance, multiple threads must queue I/O requests. This requires External Memory algorithms to be multithreaded to achieve maximum I/O performance. Figure 2.5 shows the multithreaded random read performance of the three NAND Flash configurations that we test in chapter 3. For all configurations tested, significant improvements in I/O per second (IOPS) are seen as an increasing number of threads issue read requests. The Flash configuration details are discussed in Section 3.4.

In our previous work, we addressed the challenge of achieving DRAM-like performance when some portion of the program state resides in I/O bus-connected NVRAM capable of low latency random access [25]. We identified that high levels of concurrent I/O are required to achieve optimal performance from NVRAM devices (e.g., NAND Flash); this is the underlying motivation for designing highly concurrent asynchronous graph traversals.

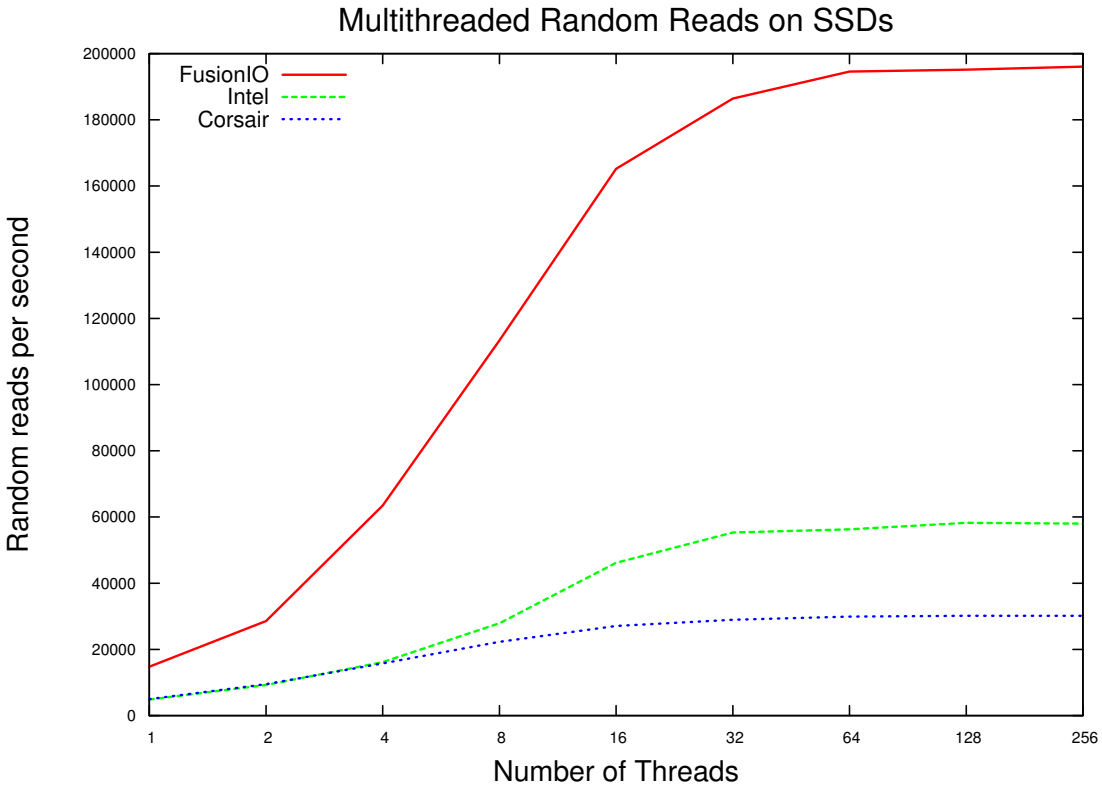


Figure 2.5: Multithreaded random read I/O performance for three NAND Flash configurations. Configuration details discussed in Section 3.4.

We also identified that the Linux I/O system software introduces many bottlenecks. This has led many application developers to use the `O_DIRECT` I/O flag to bypass Linux's default page cache system. For this work, we implemented a custom *page cache* that resides in user space and provides a POSIX I/O interface. Our custom page cache was designed to support a high level of current I/O requests, both for cache hits and misses, and interfaces with NVRAM using direct I/O. The design of our page cache is not the focus of this work, but was required to optimize performance from the NAND Flash devices used in our studies.

NVRAM in the HPC environment. Node-local or node-near NVRAM is gaining traction in the HPC environment, often motivated by improving the performance of checkpointing in traditional HPC applications [53]. Our work leverages the NVRAM for data-intensive applications. Examples of HPC systems with NVRAM include:

- Lawrence Livermore Nat. Lab.: Hyperion, Coastal;
- San Diego Supercomp. Center: Trestles, Flash Gordon;
- Tokyo Institute of Technology: TSUBAME2.

The architecture and configuration of NVRAM in supercomputing clusters is an active research topic. To our knowledge, our work is the first to integrate node-local NVRAM with distributed memory at extreme scale for important data intensive problems, helping to inform the design of future architectures.

2.6 Challenges for Processing Large Scale-Free Graphs

In this section, we identify key challenges to storing and processing massive *scale-free* graphs. Many important graph datasets have unstructured and irregular topologies which thrash multi-level memory hierarchies, including external memory. In Section 2.6.1 we describe how irregular topologies can produce dense processor-processor, approaching all-to-all, communication for parallel algorithms, leading to poor overall performance. In Section 2.6.2 we describe how the growth of high-degree vertices provides significant challenges for balancing storage, computation, and communication.

2.6.1 Dense Processor-Processor Communication

Partitioning scale-free graphs into equal sized partitions with minimal edge cuts is difficult, and often not feasible. Many scale-free graphs lack good graph separators,

resulting in many cut edges when partitioned. When a graph is partitioned with a large number of cut edges, parallel algorithms will require significant communication. Recent work on partitioning scale-free graphs has developed techniques to partition based on community structure [27, 44, 56]. However, these techniques often do not attempt to create equal sized partitions when attempting to uncover the underlying community structure. As such, these techniques are not well suited for partitioning graphs for the purposes of parallel processing.

For scale-free graphs without good graph separators, parallel algorithms will require significant communication. Specifically, when the parallel partitioned graph contains $\Omega(|E|^{\alpha})$, where $0 < \alpha \leq 1$, cut edges, a polynomial number of graph edges will require communication between processors if an algorithm requires communication along the edges. Additionally, *dense communication* occurs when $\Omega(p^{\alpha+1})$ pairs of processors share cut edges, in the worst case creating all-to-all communication.

To mitigate the dense processor-processor communication, we apply communication routing and aggregation through a synthetic network. For dense communication patterns, where every process needs to send messages to all p other processes, we route the messages through a topology that partitions the communication. We have experimented with 2D and 3D routing topologies. Figure 2.6 illustrates a 2D routing topology that reduces the number of communicating channels a process requires to $O(\sqrt{p})$. This reduction in the number of communicating pairs comes at the expense of message latency because messages require two hops to reach their destination. In addition to reducing the number of communicating pairs, 2D routing increases the amount of message aggregation possible by $O(\sqrt{p})$.

Scaling to hundreds of thousands of cores requires additional reductions in communication channels. Our experiments on the IBM BlueGene/P supercomputer use

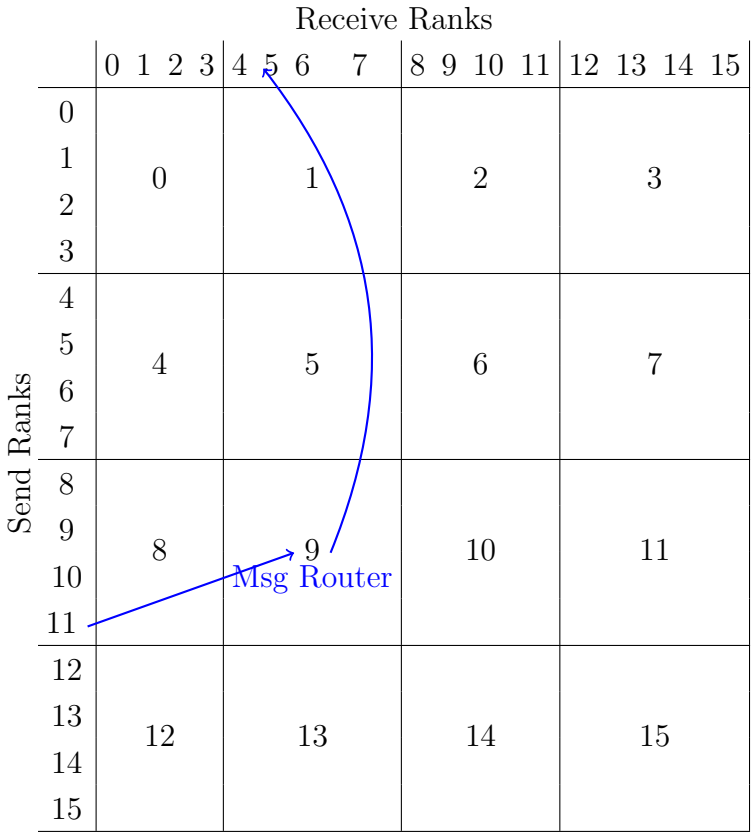


Figure 2.6: Illustration of 2D communicator routing of 16 ranks. As an example, when *Rank 11* sends to *Rank 5*, the message is first aggregated and routed through *Rank 9*.

a 3D routing topology, that is very similar to the 2D illustrated in Figure 2.6, and is designed to mirror the BG/P 3D torus interconnect topology.

Recent work related to our routed communication has been explored by Willcock [80], where active messages are routed through a synthetic hypercube network to improve dense communication scalability. A key difference to our work is that their approach has been designed for the Bulk Synchronous Parallel (BSP) model and is not suitable for asynchronous graph traversals.

2.6.2 Power-law Degree Distribution

Underlying many of the scaling challenges is the growth of high-degree vertices in the graph as the size of the graph increases. Hub vertices have degrees significantly above average, and lead to imbalances in parallel computation and communication. The hub growth for Graph500 R-MAT and Preferential Attachment scale-free graphs is shown in Figure 2.4. While the average degree is held constant at 16, the number of edges belonging to hubs continues to grow as graph size increases.

For scale-free graphs, 1D partitioning suffers from significant partition imbalance in terms of the number of edges per partition, due to the high-degree vertices in scale-free graphs. 2D partitioning is significantly better than 1D; however, 2D may still create imbalanced partitions. The weak scaling of Graph500 partition imbalance for 1D and 2D block partitioning is shown in Figure 2.7. 1D partition becomes over ten times imbalanced, while 2D is only 20% imbalanced.

Our *edge list partitioning*, discussed in chapter 4, does not suffer from imbalances due to high-degree vertices; it guarantees a balanced number of edges per partition.

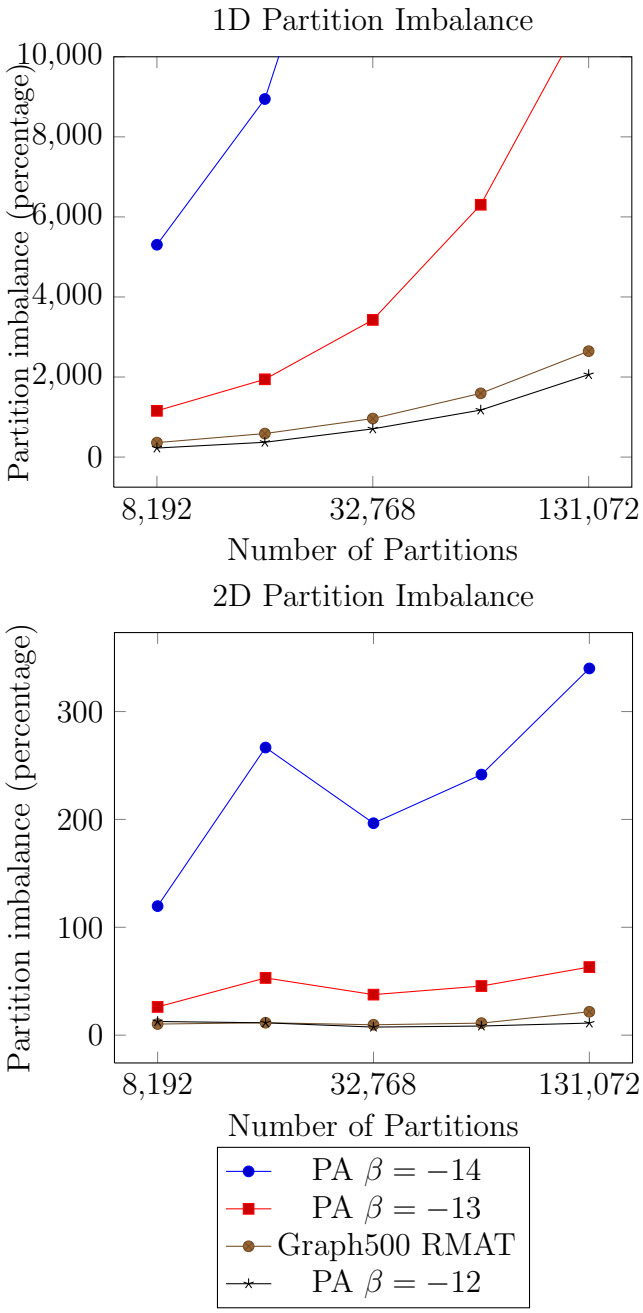


Figure 2.7: Weak scaling of partition imbalance for 1D and 2D partitioning; imbalance computed for the distribution of edges per partition. Weak scaled using 262,144 vertices per partition. The number of vertices per partition matches the experiments on BG/P Intrepid shown in Sections 5.5.2 and 5.5.5. Both 1D and 2D partitioning produce imbalanced partitioning, with the increased imbalance when the graph has greater hub size (e.g. PA $\beta = -14$). 2D partitioning is significantly better than 1D for all graphs in our studies, however our distributed delegates partitioning produces perfectly balanced partitions for these weak scaled graphs.

2.7 Graph Algorithms

In this section, we provide an overview of the graph algorithms we explore in this thesis. We focus on six important graph computations that are fundamental to many other areas of graph analysis: Breadth-First Search, Single Source Shortest Path, Connected Components, K-Core decomposition, Triangle Counting, and Page Rank.

2.7.1 Breadth-First Search (BFS)

Breadth-First Search (BFS) is a simple traversal that begins from a starting vertex and explores all neighboring vertices in a level-by-level manner. Taking the starting vertex as belonging to level 0, level 1 is filled with all unvisited neighbors of level 0. Level $i+1$ is filled with all previously unvisited neighbors of level i; this continues until all neighbors of level i have been visited. BFS runs in $O(|V| + |E|)$ time.

Many parallel versions of BFS are *level synchronous* [8, 5]. This means that all threads of execution working on level i must finish and synchronize before starting to work on level $i+1$. In some cases, additional work between the level synchronizations is needed to merge the level sets.

BFS is an algorithm that is efficient when computing in-memory, but becomes impractical in external memory. In-memory BFS incurs $\Omega(n+m)$ I/Os when using external memory, and it has been reported that the in-memory BFS performs orders of magnitude slower when forced to use external memory [2].

For general undirected graphs, Munagala and Ranade [55] improve the worst-case I/O of BFS to $O(n + sort(m))$ by exploiting the fact that a node in BFS level i can only have edges to nodes in level $i-1$ or $i+1$, removing the need to check all previous BFS levels. The $O(n)$ term in Munagala and Ranade's algorithm is due

to non-contiguous access to the adjacency lists, requiring separate access. Mehlhorn and Meyer [50] improved the adjacency list access by pre-processing the graph into subgraphs of low diameter and storing their adjacency lists contiguously, leading to sub-linear I/O complexity.

For general directed graphs, improvements over in-memory BFS and DFS have not been made, their I/O complexity is $O((n + m/B)lg(n/B) + sort(m))$ [4]. This is considered impractical for general sparse directed graphs. For an in-depth survey of EM graph traversal algorithms, see [4].

2.7.2 Single Source Shortest Path (SSSP)

A Single Source Shortest Path (SSSP) algorithm computes the shortest paths in a weighted graph from a single source vertex to every other vertex. In this work, we only address non-negatively weighted graphs. Our approach to Single Source Shortest Path (SSSP) can be viewed as a hybrid between Bellman-Ford [21] and Dijkstra's [22] SSSP. Bellman-Ford *label-correcting* computes SSSP by making $|V| - 1$ loops over all vertices, *relaxing* the path length of each vertex. Dijkstra's SSSP algorithm also iteratively relaxes vertices, but proceeds in a greedy manner, *relaxing* only the shortest-path vertex at each iteration. Dijkstra's SSSP runs in $O(|E| + |V|log(|V|))$, and Bellman-Ford runs in $O(|V| * |E|)$ time.

We show comparisons to a distributed implementation of the Δ-stepping SSSP algorithm [51] provided by PBGL [29]. This algorithm proceeds in *bulk-synchronous* steps, where vertices within a *delta* of the shortest path are relaxed together.

Asynchronous algorithms for computing shortest paths in parallel have been previously studied [10, 31]. Our work builds on these techniques to create an asynchronous approach that can overcome load imbalance and data latencies.

2.7.3 Connected Components

A connected component of an undirected graph is a subgraph in which all vertices can be connected to all other vertices through pathways in the graph. In other words, if two vertices are in the same connected component, then there exists a pathway between them in the graph. The connected components of a graph can be computed in $O(|V| + |E|)$ time.

The Shiloach-Vishkin parallel connectivity algorithm [70] is a well known PRAM algorithm for computing connected components. We show comparisons to MTGL's [8] parallel implementation, which is an iterative algorithm covered in detail by JáJá [35].

2.7.4 Triangle Counting

A triangle is a set of vertices A, B, C such that there are edges between $A - B$, $B - C$, and $A - C$. Triangle counting is a primitive for calculating important metrics such as *clustering coefficient* [78]. All triangles can be counted in $O(d_{max}^2|V|)$ time, where d_{max} is the maximum vertex degree in G.

2.7.5 K-Core Decomposition

The k-core of a graph is the largest subgraph where every vertex is connected to at least k other vertices in the subgraph. The k-core subgraph can be found by recursively removing vertices with less than degree k. K-Core has been used in a variety of fields including the social sciences [67]. K-Core can be computed in $O(|V| + |E|)$ time [49].

2.7.6 PageRank

PageRank is a network analysis tool that ranks vertices by their relative importance [57]. Designed to rank pages on the Web, PageRank models a random

web surfer that randomly follows links with random restart. It is often iteratively computed as a stochastic random walk with restart, where the starting distribution is a uniform distribution across all vertices. Each iteration of PageRank runs in $O(|V| + |E|)$ time.

3. ASYNCHRONOUS GRAPH TRAVERSAL *

In this chapter, we introduce an asynchronous approach for expressing graph algorithms that can exploit fine grained parallelism. The motivation of our work is to overcome costly parallel synchronizations that inhibit performance and scalability. As clock speeds flatten and massive parallelism becomes mainstream, asynchronous approaches will become necessary to overcome the increasing cost of synchronization.

Using currently accepted synchronous techniques, load imbalance may occur between the synchronization points, leading to performance loss. Many algorithms, including those studied in our work, require multiple synchronizations with existing techniques. We introduce the use of prioritized visitor queues to asynchronously compute Breadth First Search (BFS), Single Source Shortest Paths (SSSP), and Connected Components (CC). We show that our approach allows the computation to proceed in an asynchronous manner, reducing the number of costly synchronizations.

Our asynchronous visitor queue is described in Section 3.1. Examples of asynchronous traversal algorithms are discussed in Section 3.2, with a discussion of their algorithmic complexity in Section 3.3. Finally, an experimental study follows in Section 3.5.

3.1 Asynchronous Visitor Queue

The *vertex visitor* abstraction is a common way to abstract the process in which a graph traversal visits the vertices of a graph. The visitor pattern is used by many

*Part of the data reported in this chapter is reprinted with the kind permission of IEEE from "Multithreaded Asynchronous Graph Traversal for In-Memory and Semi-External Memory" in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, by R. Pearce, M. Gokhale, and N. M. Amato, 2010. Copyright 2010 by IEEE. [59]

graph libraries, including PBGL [29]. A visitor is a simple procedure that contains the algorithmic operations performed for an individual vertex. Our approach allows the visitors to traverse the graph asynchronously in parallel. We use the *vertex visitor* pattern with prioritized work queues to form an *asynchronous visitor queue*. Each processor is assigned a prioritized visitor queue, where pending visitors wait to be processed.

An algorithm is started with an initial set of vertex visitors that evaluate and potentially modify the state of the vertices. As the graph traversal proceeds, vertices are visited and adjacent vertices to be visited (if needed) are dynamically queued into the visitor queue. The traversal is complete when the visitor queues are empty, and all visitors have completed.

In a multithreaded environment, the *visitor queue* can be implemented as a set of priority queues with a hash function controlling the selection of an individual queue. Using multiple queues with a hash function reduces lock contention when multiple threads are inserting or removing from the queues. In our implementation and experiments, each thread 'owns' a queue and the queue is selected based on a hash of the vertex identifier. This adds an additional guarantee that a visitor has exclusive access to a vertex when executing, removing the need for additional vertex-level locking when visiting a vertex. Additionally, a near-uniform hash function may improve load balance amongst the visitor queues as high-cost vertices will be uniformly distributed across the queues.

3.2 Algorithms

We applied our asynchronous traversal techniques to Breadth First Search, Single Source Shortest Path, and Connected Components. This section describes the design of these algorithms.

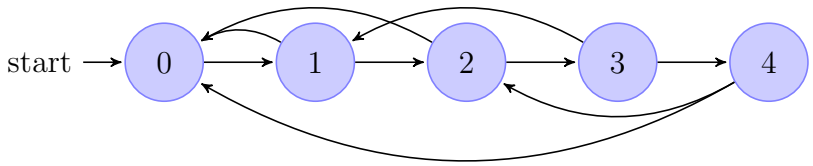


Figure 3.1: An example directed graph with poor parallelism for BFS and SSSP.

3.2.1 Breadth-First Search (BFS) and Single Source Shortest Path (SSSP)

Like Dijkstra's SSSP [22], our approach traverses paths in a prioritized manner, *visiting* the shortest path possible at each visit. Our approach does not introduce synchronizations between steps; therefore, we cannot guarantee that the absolute shortest-path vertex is visited at each step, possibly requiring multiple visits per vertex. Like Bellman-Ford [21], our approach relies on *label-correcting* to compute the traversal, and completes when all corrections are complete. In this work, we compute a Breadth First Search (BFS) by applying our asynchronous SSSP algorithm with all edge weights equal to 1.

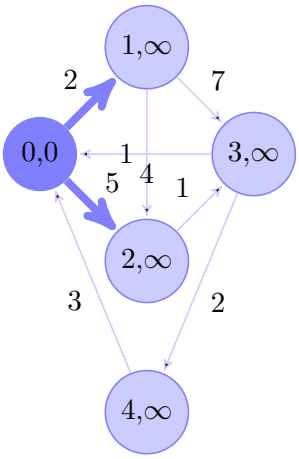
Algorithms 1 and 2 outline an asynchronous SSSP traversal of a graph $G = (V, E)$. Each vertex is assigned a label representing the path length to that vertex, initialized to ∞. The traversal starts at the source in Algorithm 1. For each vertex visited, an instance of Algorithm 2 decides if the current path length needs to be corrected, and queues the adjacent vertices if the path has been updated. A visitor corrects the path length if it represents a shorter pathway than is currently assigned to the vertex. The visitor queue is prioritized based on the visitors' path lengths. After starting the traversal, Algorithm 1 waits for all queued visitors to complete.

Algorithm 1 Single Source Shortest Path – Main

```
1: INPUT: g ← input weighted graph G(V,E)
2: INPUT: dist_array ← an array holding the path length to each vertex, initialized to ∞
3: INPUT: parent_array ← an array holding the path parent to each vertex, initialized to ∞
4: INPUT: start ← starting vertex for SSSP
5: pq_visit ← a multithreaded visitor queue, prioritized by SSSPVertexVisitor's cur_dist
6: pq_visit.push( SSSPVertexVisitor(g, pq_visit, dist_array, parent_array, start, 0, start) )
7: pq_visit.wait() //wait for queued work to finish
```

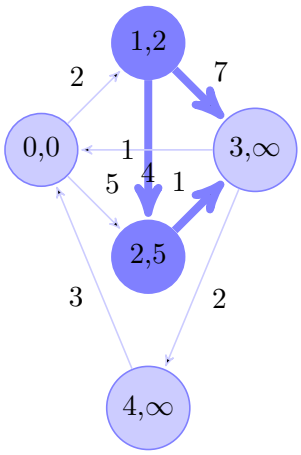
Algorithm 2 Single Source Shortest Path – SSSPVertexVisitor

```
1: INPUT: g ← input weighted graph G(V,E)
2: INPUT: pq_visit ← a multithreaded visitor queue
3: INPUT: dist_array ← an array holding the path length
4: INPUT: parent_array ← an array holding the path parent
5: INPUT: v ← vertex to visit
6: INPUT: cur_dist ← tentative bound on min path length
7: INPUT: cur_parent ← tentative shortest path parent
8: if cur_dist < dist_array[v] then
9:     dist_array[v] = cur_dist //relax vertex information
10:    parent_array[v] = cur_parent
11:    adj_list ← g.adj_list( v )
12:    for all vj ∈ adj_list do
13:        edge_weight = g.edge_weight(v,vj)
14:        pq_visit.push( SSSPVertexVisitor(g, pq_visit, dist_array, parent_array, vj, edge_weight, v ) )
15:    end for
16: end if
```



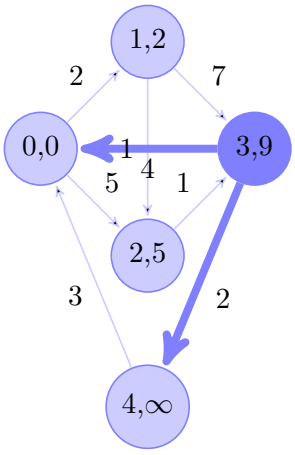
Visitor Queues				
0	1	2	3	4
0	**2**	**5**		

(a)



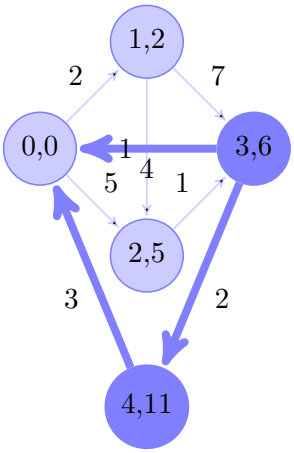
Visitor Queues				
0	1	2	3	4
0	**2**	**5**	**9**	
		6	**6**	

(b)



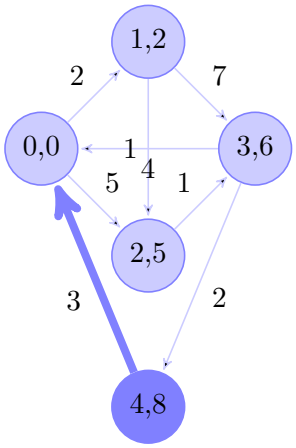
Visitor Queues				
0	1	2	3	4
0	2	5	**9**	**11**
10		**6**	6	

(c)



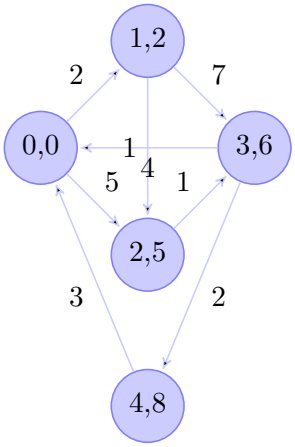
Visitor Queues				
0	1	2	3	4
0	2	5	9	**11**
10		6	**6**	**8**
7				
14				

(d)



Visitor Queues				
0	1	2	3	4
0	2	5	9	11
10		6	6	**8**
7				
11				
14				

(e)



Visitor Queues				
0	1	2	3	4
0	2	5	9	11
10		6	6	8
7				
11				
14				

(f)

Figure 3.2: An example of an asynchronous Single Source Shortest Path (SSSP) traversal of a simple weighted directed graph. Section 3.2.2 discusses the details of this example.

3.2.2 SSSP Traversal Example

To illustrate how the asynchronous computation proceeds we describe SSSP as seen in Algorithms 1 and 2. The path length for all vertices is initially set to ∞ and the visitor queues are empty. The computation starts by queuing a visitor at the *source* with $cur\_dist = 0$. Upon visiting a vertex, each visitor evaluates if the current path length needs to be corrected. If the visitor updates the path, the visitor queues new visitors for the vertex's adjacent vertices.

Figure 3.2 gives a pictorial example for a simple weighted directed graph. In this example, the weights were purposely selected to require multiple visits per vertex. In a real-world context, the weights may represent distances between locations, strength of association between agents, or any other domain-specific relationship information. For simplicity, the computation is represented by 6 steps; however it is important to note that no synchronization is introduced between the steps and the order of the visitor queues is not guaranteed. For clarity, a visitor queue is shown for each vertex; however in practice, a queue may represent a large subset of vertices. The computation in Figure 3.2 proceeds as follows:

(a) All vertices initialize their path length to ∞. *Vertex 0* initializes to a path length of 0 and queues a visitor to vertex 1 with length 2, and a visitor to vertex 2 with length 5.

(b) *Vertex 1* is visited with length 2, updates its path length to 2, and queues a visitor to vertex 2 with length 6, and a visitor to vertex 3 with length 9. *Vertex 2* is visited with length 5, updates its path length to 5, and queues a visitor to vertex 3 with length 6.

(c) *Vertex 2* is visited with length 6; because length 6 is longer than its current

length 5, it does not update its path length; no new visitors queued. *Vertex 3* has 2 visitors queued, however order is not guaranteed and it processes length 9 first, updates its path length to 9, and queues a visitor to vertex 0 with length 10 and visitor to vertex 4 with length 11.

(d) *Vertex 0* is visited with length 10, and does not update its path length; no new visitors queued. *Vertex 3* is visited with length 6, updates its path length to 6, and queues a visitor to vertex 4 with length 8, and a visitor to vertex 0 with length 7. *Vertex 4* is visited with length 11, updates its path length to 11, and queues a visitor to vertex 0 with length 14.

(e) *Vertex 0* is visited with length 7, and does not update its path length; no new visitors queued. *Vertex 4* is visited with length 8, updates its path length to 8, and queues a visitor to vertex 0 with length 11.

(f) *Vertex 0* is visited with length 11, and does not update its path length; no new visitors queued. For the subsequent time step, vertex 0 is visited with length 14, and does not update its path length; no new visitors queued. *End*: The computation terminates when all visitor queues are empty.

3.2.3 Undirected Connected Components

The asynchronous computation of the Connected Components (CC) of an undirected graph is similar to that of SSSP. To compute the CCs, each vertex is labeled by the smallest vertex descriptor that is reachable by a path in the graph, where the vertex descriptor is an integer that labels the vertex. The computation is outlined in Algorithms 3 and 4; in Algorithm 3, a visitor for each vertex is queued in parallel with the vertex's descriptor as the starting component id. When a vertex is visited, if its component id can be updated to a smaller id, then it is updated and visitors for

all adjacent vertices are queued with the updated component id. The computation is finished when all queued visitors complete.

Algorithm 3 Undirected Connected Components – Main

```
1: INPUT: g ← input graph
2: INPUT: ccid_array ← an array holding the cc id for each vertex, initialized to
   ∞
3: pq_visit ← a multithreaded visitor queue, prioritized by UCCVertexVisitor's
   cur_ccid
4: for all v ∈ g.vertex_list() parallel do
5: |  pq_visit.push( UCCVertexVisitor(g, pq_visit, ccid_array, v, v) )
6: end for
7: pq_visit.wait() //wait for queued work to finish
```

Algorithm 4 Undirected Components – UCCVertexVisitor

```
1: INPUT: g ← input graph
2: INPUT: pq_visit ← a multithreaded visitor queue
3: INPUT: ccid_array ← an array holding the cc id
4: INPUT: v ← vertex to visit
5: INPUT: cur_ccid ← tentative bound on min cc id
6: if cur_ccid < ccid_array[v] then
7: |  ccid_array[v] = cur_ccid //relax vertex information
8: |  adj_list ← g.adj_list( v )
9: |  for all vj ∈ adj_list  do
10: |  |  pq_visit.push( UCCVertexVisitor(g, pq_visit, ccid_array, vj, cur_ccid) )
11: |  end for
12: end if
```

3.3 Algorithmic Analysis

The performance of Algorithms 1 and 2 is highly dependent on the structure of the graph traversed. If the graph has multiple shortest-path pathways that can be independently traversed, then the algorithm will have the opportunity to proceed in parallel. However, without the independent pathways, the algorithm will traverse the graph in a serialized manner. Figure 3.1 is an example of a directed graph with poor parallelism when traversed starting from vertex 0.

More formally, the algorithm's complexity can be represented by $O((|E|/p)log(|V|/p))$, where V and E are the number of vertices and edges in the graph, respectively, and p is the degree of parallelism the traversal can exploit. In the worst case, where the traversal is serialized ($p = 1$), the algorithm reduces to Dijkstra's SSSP with a performance of $O(|E|log|V|)$. From our experiments with scale-free graphs and web-graphs, presented in Section 3.5, a significant amount of path parallelism exists in these real-world graphs, giving rise to performance approaching the best case.

As with SSSP, the parallel performance of Algorithms 3 and 4 is highly dependent on the underlying graph structure. A worst case graph with poor parallelism is similar to that of SSSP in Figure 3.1, only undirected.

Our approach to CC can be viewed as performing parallel BFS starting from every vertex. When two BFSs that started from different vertices merge, the BFS that started from the lowest vertex identifier takes over the remainder of both traversals. The end result is that all vertices in the graph are labeled with the smallest vertex identifier connectable to them.

3.4 Implementation Details

We have created two similar implementations for In-Memory and Semi-External Memory graphs.

Thread Oversubscription. Our implementation can benefit from using more threads than cores. Because there is a prioritized queue per thread, with an associated lock, having more threads/queues than cores reduces queue lock contention. From our experiments, using as many as 512 threads on 16 cores offers substantial benefit.

In-Memory Implementation. For In-Memory graph storage, we used Boost's Compressed Sparse Row C++ library [71]. For POSIX thread support, we used the Boost Thread library [81]. All code was compiled with g++ version 4.1.2 with -O3.

Semi-External Implementation. For Semi-External graph storage, we used a custom file-based storage implementing a *compressed sparse row* using explicit POSIX standard I/O access. In the semi-external graph scenario, the algorithmic has enough memory to store algorithmic information about the vertices but not edges. The entire graph structure is stored on the persistent storage device, and the visitor queues and the output of the algorithm are stored in main memory.

For POSIX thread support, we used the Boost Thread library [81]. All code was compiled with g++ version 4.1.2 with -O3.

The only difference from the in-memory algorithm implementation is that the prioritized visitor queues have an additional secondary sorting parameter, the vertex identifier. This increases access locality to the storage devices by sorting the accesses. Because we use a compressed sparse row graph format, the adjacency set for vertex i will be close in terms of data locality to the adjacency set of vertex $i + 1$. When the visitor queues are able to visit vertices with close identifiers, aggregate page-level locality can be exploited. Using Breadth-First Search as an example, not only will level 1's vertices be processed before level 2's, the vertices in level 1 will be visited

in a semi-sorted order to increase locality.

3.5 Experimental Study

We present an experimental study applying our technique to both In-Memory (IM) and Semi-External Memory (SEM) graphs utilizing multi-core processors and solid-state memory devices (SSDs). We provide a quantitative study comparing our approach to existing implementations. Our experimental study evaluates both synthetic and real-world datasets, and shows that our asynchronous approach is able to overcome data latencies and provide significant speedup over alternative approaches.

3.5.1 Graph Types and Sizes

We performed experiments using both synthetic and real graph inputs of various sizes. For the three algorithms tested in this work, the input graphs were organized as follows:

- BFS - Directed synthetic graphs, unweighted;
- SSSP - Directed synthetic graphs, two forms of random weights;
- CC - Undirected graphs, synthetic and real.

Synthetic Graphs. For synthetic graphs, we used scale-free graphs generated by the RMAT [18] graph generator. The RMAT graph generator uses a 'recursive matrix' model to create graphs that model 'real-world' graphs. We generated directed graphs with unique edges ranging from $2^{25} - 2^{30}$ vertices and an average out-degree of 16. The vertex identifiers are permuted after graph generation. Undirected versions of these graphs for use with Connected Components were created by adding reverse

				Size on Flash Device	
graph	# nodes	# edges	# components	directed	undirected
RMAT-A	2^{25}	2^{29}	34,008	small	
	2^{26}	2^{30}	72,647	small	
	2^{27}	2^{31}	154,179	9 GB	17 GB
	2^{28}	2^{32}	327,072	18 GB	34 GB
	2^{29}	2^{33}	689,979	36 GB	68 GB
	2^{30}	2^{34}	1,448,438	72 GB	136 GB
RMAT-B	2^{25}	2^{29}	13,739,228	small	
	2^{26}	2^{30}	28,448,613	small	
	2^{27}	2^{31}	58,757,785	9 GB	17 GB
	2^{28}	2^{32}	121,037,055	18 GB	34 GB
	2^{29}	2^{33}	249,937,778	36 GB	68 GB
	2^{30}	2^{34}	510,267,039	72 GB	136 GB
ClueWeb09 [17]	1,667,267,985	7,939,647,897	3,149,668	n/a	
it-2004 [12]	41,291,595	1,150,725,436	979	n/a	
sk-2005 [12, 11]	50,636,155	1,949,412,601	126	n/a	14 GB
uk-union [12]	133,633,041	5,507,679,822	2,097,197	n/a	36 GB
webbase-2001 [12]	118,142,156	1,019,903,190	2,721,051	n/a	

Table 3.1: Properties of graph datasets used in experiments.

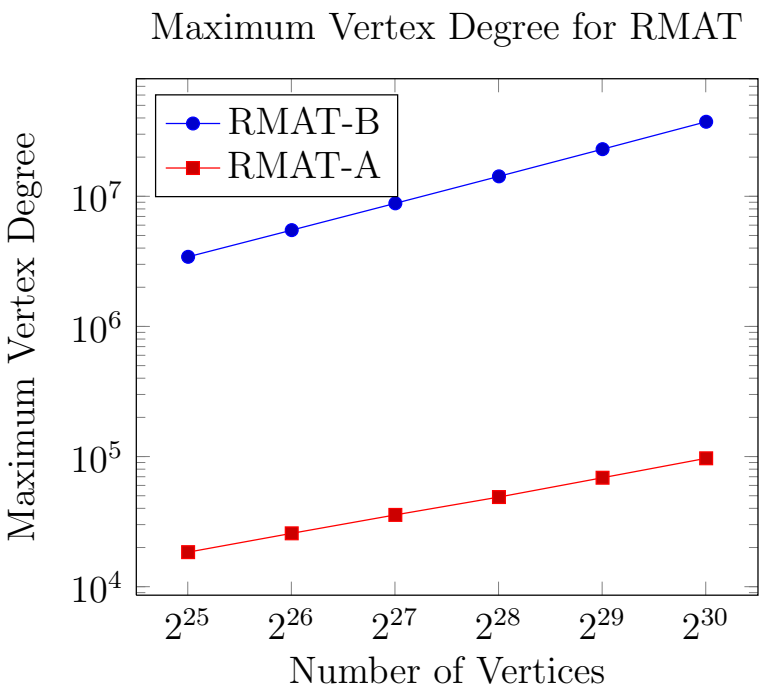


Figure 3.3: Maximum Vertex Degree for RMAT-A and RMAT-B graphs.

edges. Properties of the RMAT graphs are shown in Table 3.1. We generated 2 types of RMAT graphs with different RMAT parameters:

- RMAT-A : $a = 0.45, b = 0.15, c = 0.15, d = 0.25$: This creates a scale-free graph with medium sized hub vertices, shown in Figure 3.3;
- RMAT-B : $a = 0.57, b = 0.19, c = 0.19, d = 0.05$: This creates a scale-free graph with large high degree vertices, shown in Figure 3.3. This is the RMAT configuration for the Graph500 benchmark.

For weighted SSSP experiments, we added edge weights to the RMAT graphs in the following manner:

- UW - uniform weights range from $[0, num\_vertices)$

- LUW - log-uniform weights range from $[0, 2^i)$, where i is chosen uniformly from $[0, lg(num\_vertices))$

Real Graphs. For real graphs, we experimented with five different web traces that were treated as undirected. Properties of the five web graphs are shown in Table 3.1.

3.5.2 Hardware Resources

Our implementation and experiments were performed using Linux computers at Lawrence Livermore National Laboratory:

- AMD256GB – Single compute node; 16-core AMD Opteron(tm) 8356 with 256 GB of main memory. This machine is used for Async, MTGL, SNAP, DIMACS-SSSP, and BGL discussed in 3.5.3.

- AMDCluster – Linux cluster; 16-core AMD Opteron(tm) 8356 with 32 GB of main memory. This cluster is used only for PBGL discussed in 3.5.3.

- AMD16GB – Single compute node; 16-core AMD Opteron(tm) 8356 with 16 GB of main memory. In addition, this machine can be configured with 3 different types of NAND Flash storage.

Using the AMD16GB machine, we experimented with 3 types of NAND Flash based storage:

- AMD16GB-FusionIO – 4x 80GB FusionIO SLC, PCI-E cards in a software RAID 0 configuration. Our experiments show that this configuration is capable of close to 200,000 random reads per second. This is the fastest device that we have tested, and much of its speed is due to its PCI-E interface.

- AMD16GB-Intel – 4x 80GB Intel X25-M MLC, SATA SSDs in a software RAID 0 configuration. Our experiments show that this configuration is capable of close to 60,000 random reads per second.

- AMD16GB-Cosair – 4x 128GB Corsair P128 MLC, SATA SSDs in a software RAID 0 configuration. Our experiments show that this configuration is capable of close to 30,000 random reads per second.

Our semi-external approach requires a high level of IOPS to achieve good performance. For this reason, we have not studied our approach on traditional rotating media. Figure 2.5 shows the multithreaded random I/O performance of the three NAND Flash configurations that we test in this work.

3.5.3 In-Memory Experiments

In-Memory Graph Libraries. For experimental comparison, we show the performance of the following graph libraries:

- Async – our asynchronous approach.
- MTGL [8, 9] – a shared memory parallel graph library primarily designed for Cray's massively multithreaded machines. For commodity SMP systems, MTGL has implementations of BFS and CC that use the QThreads library [79] for threading support. It is important to note that MTGL's performance on SMP systems does not reflect on its performance on the Cray XMT. Our experiments use MTGL's Subversion Trunk revision 2827.
- SNAP [5] – a parallel graph library for shared memory which utilizes OpenMP for parallelism. Our experiments use SNAP version 0.3.
- PBGL [29] – a distributed memory parallel graph library. *Direct comparisons between distributed memory and shared memory implementations are not possible, however these experiments help to compare alternative techniques for large graph processing.* Our experiments use PBGL from version 1.43 of the Boost

library. PBGL experiments are performed on AMDCluster up to 2048-cores; we report the core-count that performs optimally.

- BGL [71] – a serial graph library. BGL is used as an efficient serial baseline to compute speedup. Our experiments use BGL from version 1.43 of the Boost library.

- DIMACS-SSSP – an implementation of Goldberg's multilevel bucket shortest path algorithm [28]. It was used as the reference solver from the 9th DIMACS Implementation Challenge on shortest paths.

3.5.3.1 Breadth First Search (BFS)

Figure 3.4 shows our asynchronous Breadth First Search (BFS) compared with MTGL, SNAP, PBGL all normalized by BGL. Our approach, MTGL, SNAP and BGL were tested on AMD256GB. PBGL is shown with the optimal number of cores between 64-2048 and 128GB-4TB of memory tested on AMDCluster.

At full parallelism, our asynchronous BFS is roughly 1.6-1.8x faster than MTGL's and 2.3-5.3x times faster than SNAP's BFS for our test cases. SNAP's BFS struggles with the highly skewed degree distribution of the RMAT-B datasets, leading to poor scaling and speedup. MTGL's and SNAP's graph data structures are implemented using 64-bit integers and are unable to fit the 2^{29} and 2^{30} vertex graphs in 256GB of memory; our implementation can be configured to use 32 or 64-bit integers.

Using significant resources, 128-1024 cores with 256GB-2TB of memory, PBGL can compute BFS on the 2^{28}, 2^{29}, and 2^{30} vertex graphs the fastest. However, PBGL's parallel speedup is small relative to the number of cores used.

The scalability of Async, MTGL, and SNAP on AMD256GB is shown in Figure 3.5 for the 2^{28} vertex graphs. The modest benefit of thread oversubscription for

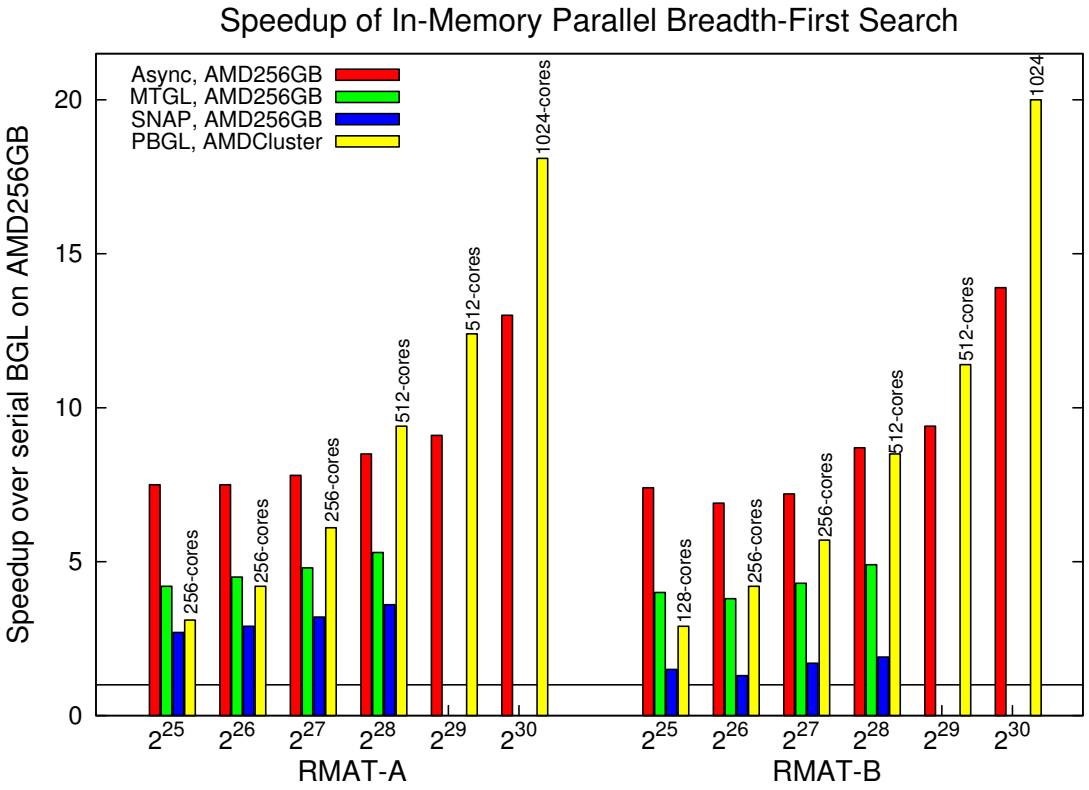


Figure 3.4: Performance comparison of In-Memory Breadth First Search (BFS). Speedup shows parallel libraries normalized by BGL. For PBGL, we report the optimal core-count up to 2048.

Async can be seen here, as in all test cases 512 threads outperform 16 threads using our approach; this indicates that further scaling is possible beyond 16-cores. MTGL and SNAP do not benefit from thread oversubscription.

Overall, our approach outperforms MTGL and SNAP in all of our test cases, and was competitive with PBGL which uses 4-64x the number of cores and up to 8x the memory.

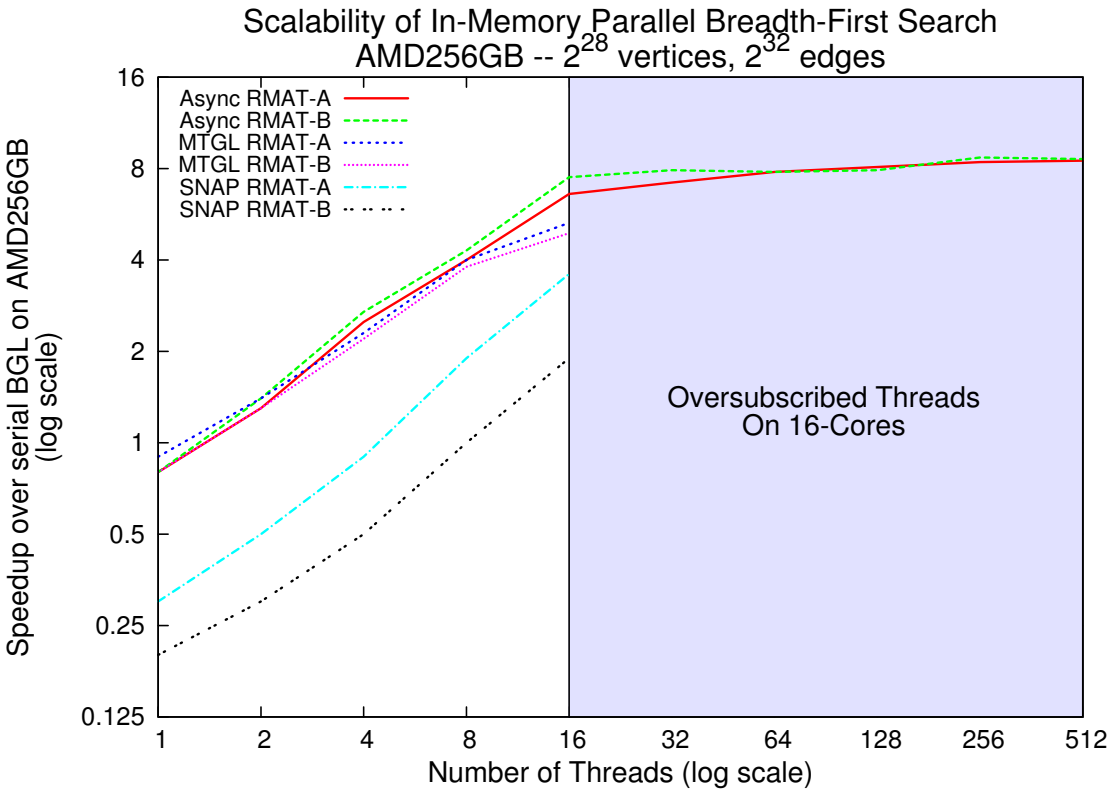


Figure 3.5: Scalability of In-Memory Breadth First Search (BFS). Scalability shows shared-memory libraries only normalized by BGL.

3.5.3.2 Single Source Shortest Path (SSSP)

Figure 3.6 shows Async and PBGL normalized with the DIMACS-SSSP tested on AMD256GB. PBGL's Δ-stepping SSSP is shown with the optimal number of cores between 64-2048 and 128GB-4TB of memory AMDCluster. These experiments use the same directed graphs as the BFS experiments in Section 3.5.3.1 and add uniform (UW) and log-uniform (LUW) edge weights.

PBGL's Δ-stepping SSSP is unable to scale past 256-cores. The optimal core-count was 64-256 cores for the UW graphs. An appropriate *lookahead* value could not be found for the LUW graphs. For both the UW and LUW graphs, the default heuristic for choosing a *lookahead* value was poor.

Again, we see that Async benefits from thread oversubscription as all tests perform best using 512 threads on 16 cores, which indicates that further scaling is possible beyond 16-cores, shown in Figure 3.7. Async achieves a speedup of up to 15.3 on 16-cores when normalized to DIMACS-SSSP.

3.5.3.3 Connected Components

Figure 3.8 shows Async, MTGL, PBGL normalized by BGL on AMD256GB. PBGL is shown with the optimal number of cores between 64-2048 and 128GB-4TB of memory on AMDCluster.

Our asynchronous CC is 2.8-4.5x faster than MTGL in all synthetic cases tested. For the real web-graphs, our asynchronous CC is 4-13 times faster than MTGL.

At 16 threads, our approach is consistently better than MTGL in our experiments. Oversubscribing to 512 threads further improves performance in all cases, which again indicates that further scaling is possible beyond 16-cores, shown in Figure 3.9

Using significant resources, PBGL is able to outperform both Async and MTGL

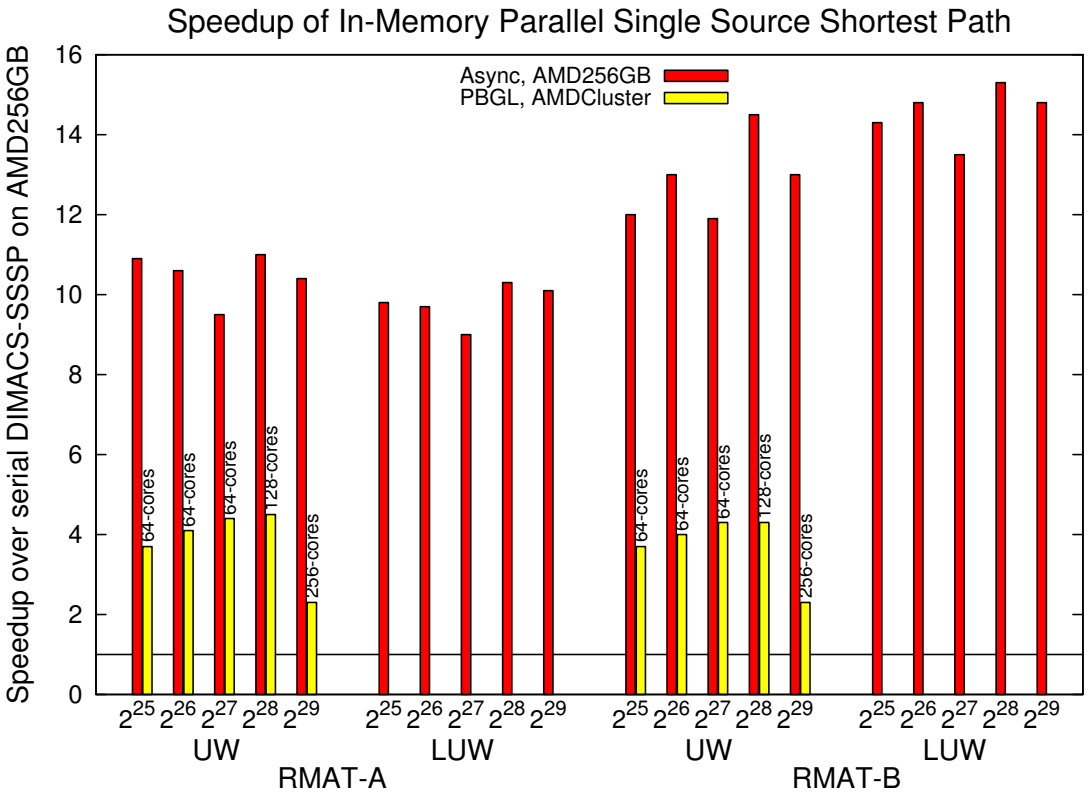


Figure 3.6: Performance comparison of In-Memory Singe Source Shortest Path (SSSP). Speedup shows parallel libraries normalized by BGL. For PBGL, we report the optimal core-count up to 2048.

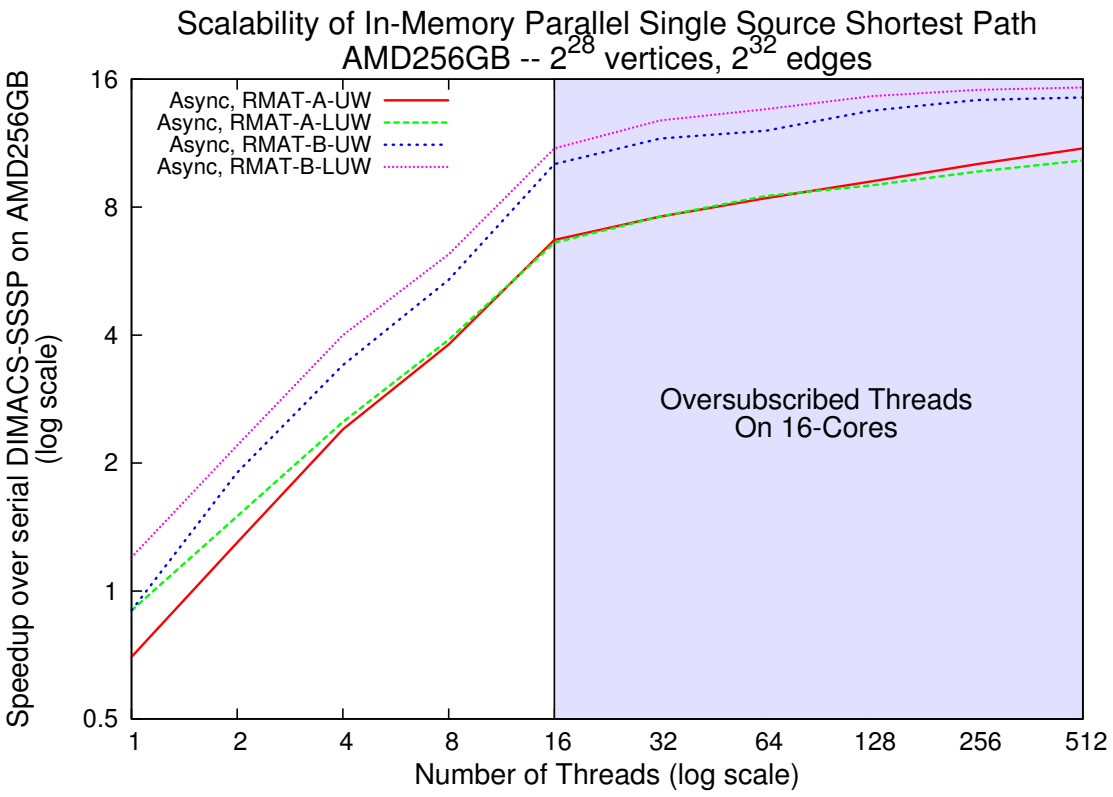


Figure 3.7: Scalability of In-Memory Singe Source Shortest Path (SSSP). Scalability shows shared-memory libraries only normalized by BGL.

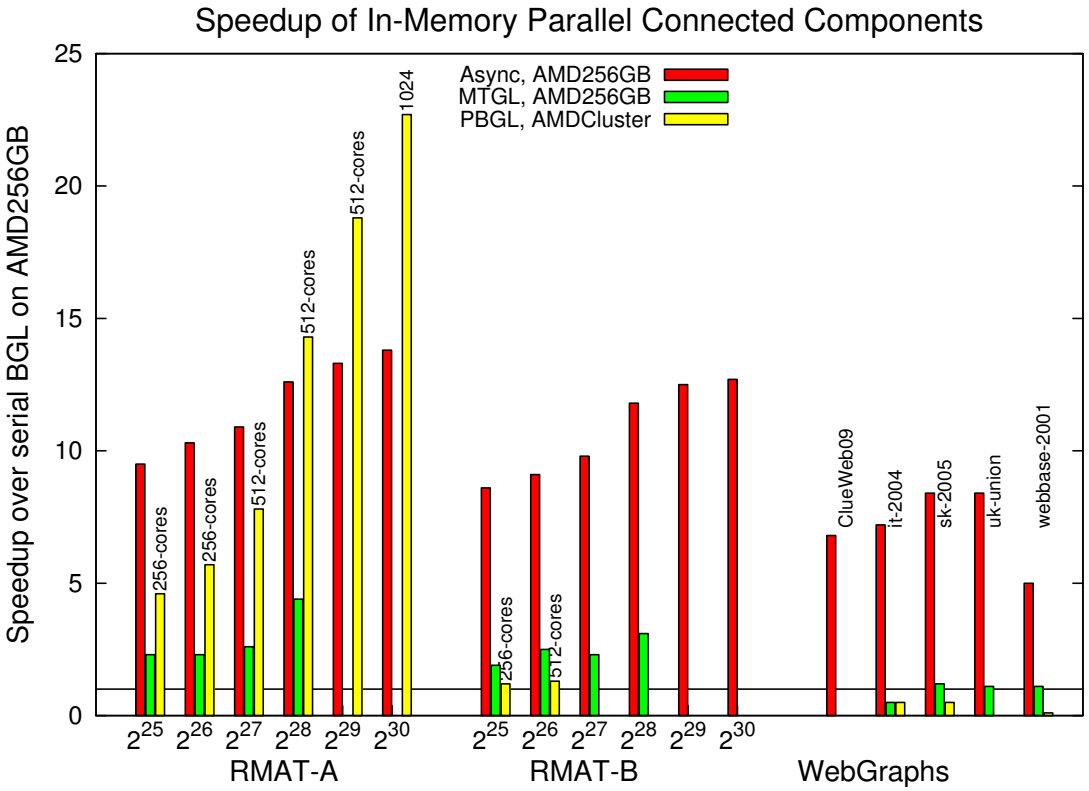


Figure 3.8: Performance comparison of In-Memory Connected Components (CC). Speedup shows parallel libraries normalized by BGL. For PBGL, we report the optimal core-count up to 2048.

on three of the largest RMAT-A graphs. However, PBGL performs extremely poorly on the RMAT-B and WebGraphs; in many cases it is unable to complete due to memory limitations.

3.5.4 Semi-External Memory Experiments

This section describes our experiments traversing Semi-External Memory (SEM) graphs stored on solid state FLASH devices on AMD16GB normalized to BGL running In-Memory on AMD256GB. It is important to note that the In-Memory BGL performance numbers are used as a baseline to evaluate the Semi-External perfor-

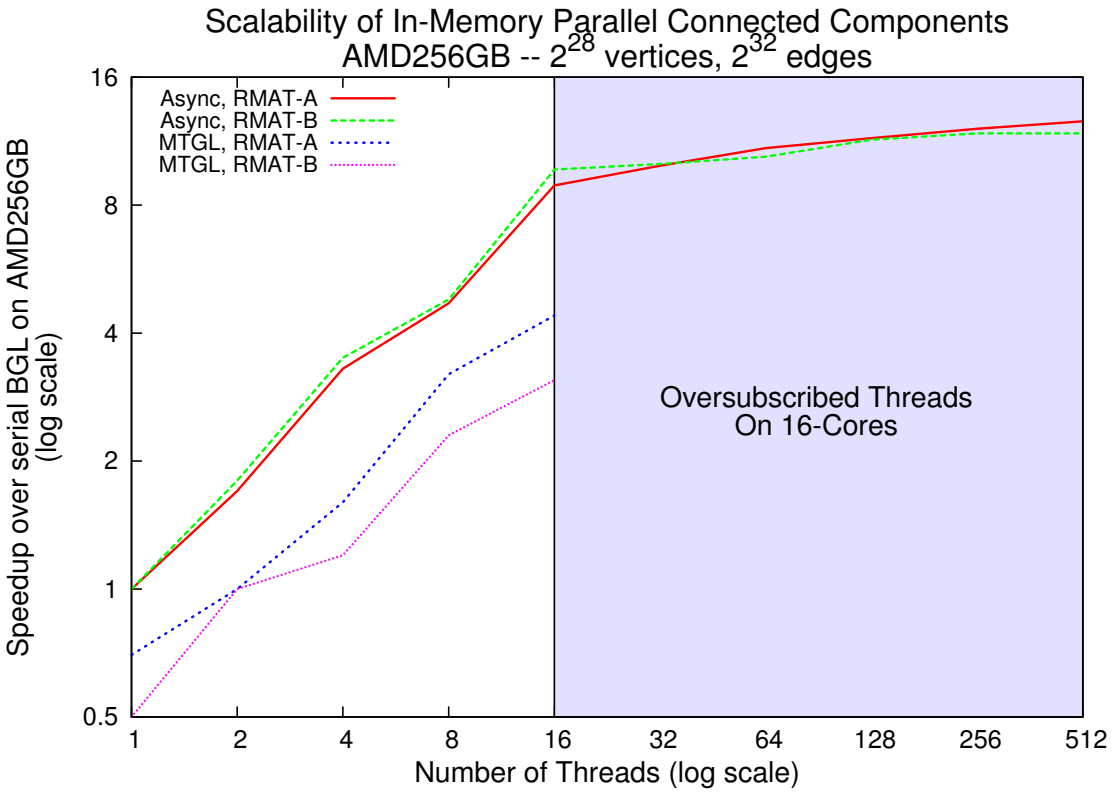


Figure 3.9: Scalability of In-Memory Connected Components (CC). Scalability shows shared-memory libraries only normalized by BGL.

mance. Also, the processors and motherboard on AMD16GB and AMD256GB are identical (only memory size differs), so a direct comparison between IM and SEM can be made.

The ability to process large graphs in semi-external memory at comparable or better to in-memory performance is important. The cost of solid state devices like NAND Flash SSDs can be significantly less than DRAM costs, and offers persistent data storage. Our SEM experiments show that for moderate and fast SSDs, our asynchronous approach is consistently faster than a serial In-Memory alternative like BGL, with even the slowest SSD tested performing comparable to BGL.

3.5.4.1 Breadth First Search

Figure 3.10 shows our Semi-External asynchronous BFS compared with the In-Memory BGL BFS. Using the FusionIO or Intel SSDs, we typically outperform the serial BGL which requires a larger amount of memory to store the graph in-memory. The FusionIO drive offers the highest random I/O access speed and typically outperforms other SSDs we have tested. Even the Corsair, the slowest of the SSDs we tested, shows performance comparable to BGL's in-memory performance.

3.5.4.2 Connected Components

Figure 3.11 shows our Semi-External asynchronous CC compared with the In-Memory BGL CC. As with BFS, our semi-external approach to connected components can outperform BGL's in-memory using the FusionIO and Intel SSDs. Again, the FusionIO drive typically offers the highest semi-external performance.

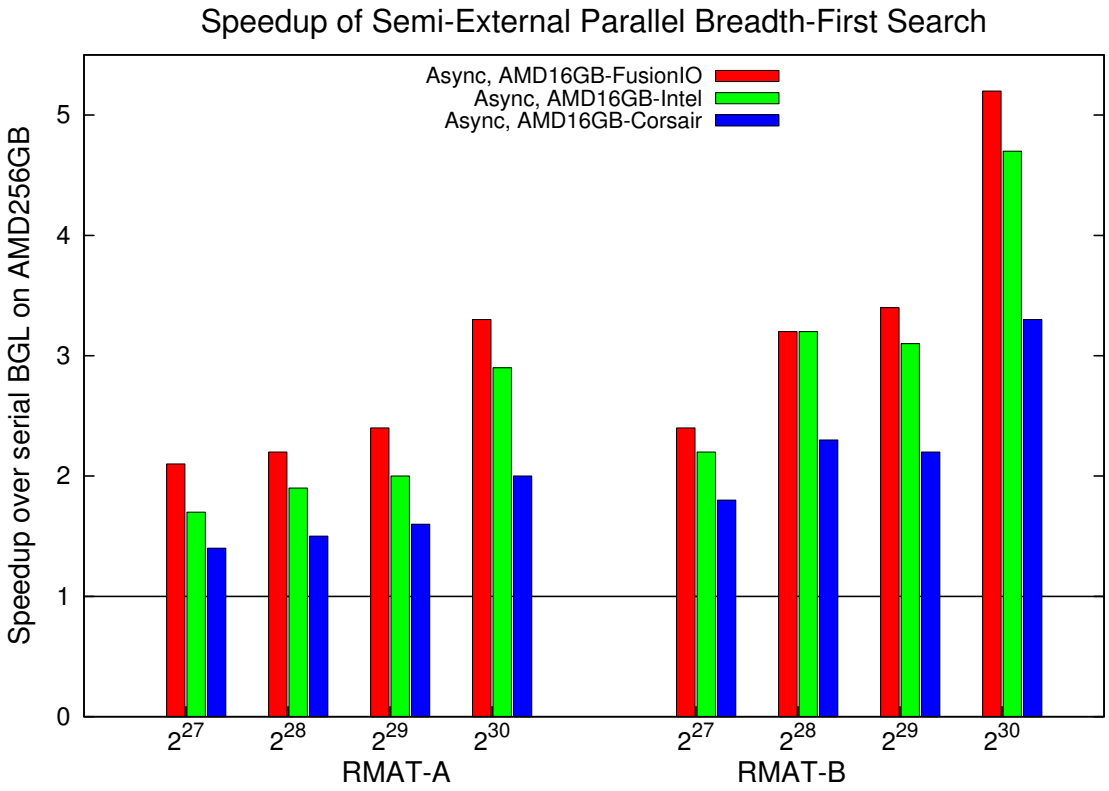


Figure 3.10: Performance comparison of Breadth-First Search in Semi-External Memory on three FLASH memory configurations. Async performance in semi-external memory normalized by In-Memory BGL's serial version on AMD256GB.

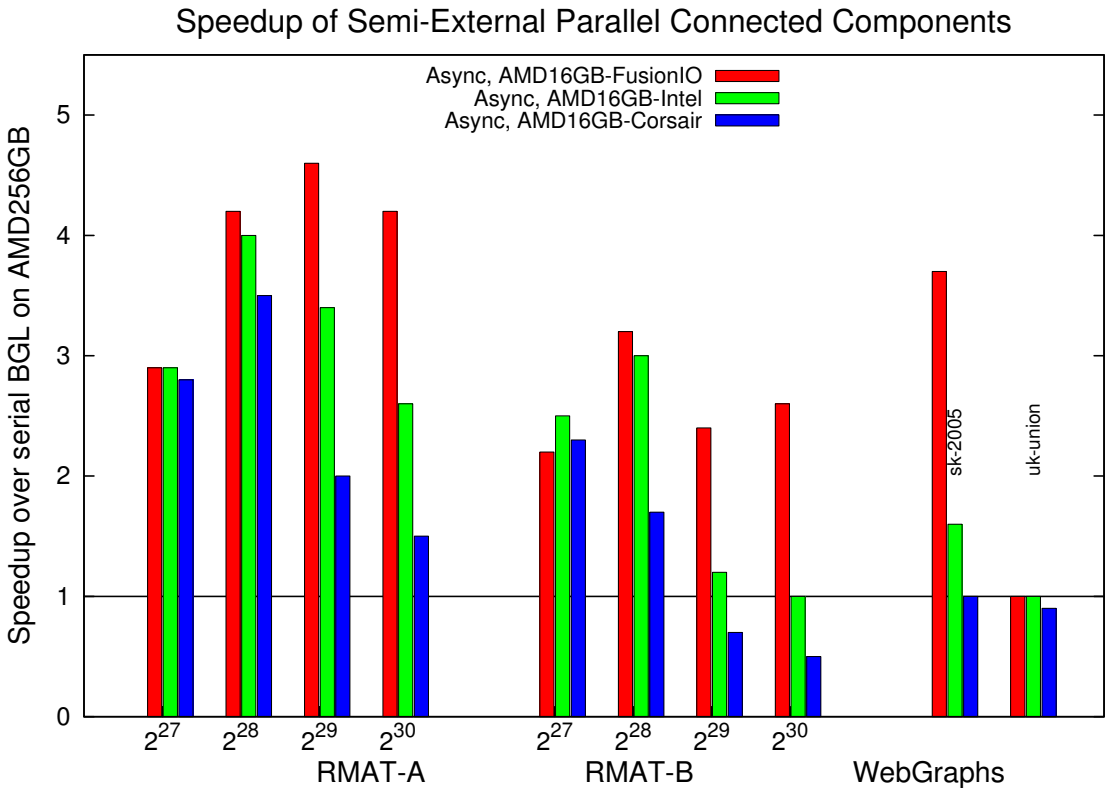


Figure 3.11: Performance comparison of Connected Components in Semi-External Memory on three FLASH memory configurations. Async performance in semi-external memory normalized by In-Memory BGL's serial version on AMD256GB.

Name	Configuration	Graph Storage	Num Vertices	TEPS
Kraken	32-cores, 512GB DRAM	DRAM	2^{31}	104.6 MTEPS
		Fusion-io	2^{34}	55.6 MTEPS
Leviathan	40-cores, 1TB DRAM	Fusion-io	2^{36}	52 MTEPS

Table 3.2: Graph500 results using NAND Flash in shared-memory. Scale 36 is a graph with over 1 trillion edges.

3.5.4.3 Graph500: Breadth-First Search

In 2011, we submitted two entries to the Graph500 benchmark that highlighted our work using NVRAM as semi-external memory. The two systems, *kraken* and *leviathan*, were located at Lawrence Livermore National Laboratory and their performance results in shown in Table 3.2. On Kraken, we showed that using NVRAM our approach can process a graph 8 times larger with only a 50% drop in performance in Traversed Edges Per Second (TEPS).

3.6 Summary

We developed a novel asynchronous approach for graph traversal and demonstrate it by performing Breadth First Search (BFS), Single Source Shortest Paths (SSSP), and Connected Components (CC) computations for large graphs in shared memory. Our approach allows the computation to proceed in an asynchronous manner, reducing the number of costly synchronizations.

We show an experimental study applying our technique to both In-Memory (IM) and Semi-External Memory (SEM) graphs utilizing multi-core processors and solid-state memory devices (SSDs). We provide a quantitative study comparing our approach to existing implementations such as the Boost Graph Library (BGL) [71], the Parallel Boost Graph library (PBGL) [29], the Multithreaded Graph Library (MTGL) [8], and the Small-world Network Analysis and Partitioning library (SNAP)

[5]. Our experimental study evaluates both synthetic and real-world datasets, and shows that our asynchronous approach is able to overcome data latencies and provide significant speedup over alternative approaches. Our In-Memory experiments show that our asynchronous BFS is 1.6-1.8x faster than MTGL's BFS and 2.3-5.3x faster than SNAP's BFS. Our BFS was competitive with PBGL which used 4-64x the number of cores and up to 8x the memory. Our asynchronous CC is 2.8-13x faster than MTGL's CC; the wide range is attributable to differences in the graph structure. Our Semi-External Memory experiments show that for moderate and fast SSDs, our asynchronous approach is consistently faster than a serial In-Memory alternative like BGL, with even the slowest SSD tested being competitive with BGL.

4. BALANCED PARTITIONING WITH HIGH-DEGREE VERTICES *

Partitioning a graph amongst p distributed processes becomes challenging with the presence of high-degree *hub* vertices. As discussed in Section 2.6.1, high-degree vertices can cause significant partition imbalance leading to degraded performance.

In this chapter, we present our *edge list partitioning* (ELP) approach that evenly partitions a graph's edges, mitigating the adverse effects of high-degree vertices. ELP solves the issues if partition imbalance present with 1D and 2D partitioning. An input graph is initially represented as a sorted edge list, and evenly distributed across the partitions. This leads to a balanced number of edges assigned per partition; however, vertices may have edges spanning multiple partitions. Our visitor queue technique manages the visitation of vertices whose edges span multiple partitions, and we show its application on three graph algorithms: Breadth-First Search, K-Core and Triangle Counting.

We demonstrate the scalability of our approach on up to 131K cores of BG/P Intrepid, and we show that by leveraging node-local NAND Flash, our approach can process 32x larger datasets with only a 39% performance degradation traversal performance.

Our approach is discussed in section 4.1. Our implementation using a distributed visitor queue is discussed in section 4.2. Examples of algorithms using edge list partitioning are discussed in section 4.3, followed by a complexity analysis in section 4.4. An experimental study performed on distributed-memory systems is shown in section 4.5.

*Part of the data reported in this chapter is reprinted with the kind permission of IEEE from "Scaling Techniques for Massive Scale-Free Graphs in Distributed (External) Memory" in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS)*, by R. Pearce, M. Gokhale, and N. M. Amato, 2013. Copyright 2013 by IEEE. [60]

4.1 Edge List Partitioning

To maintain a balance of edges across p partitions with ranks 0 to $p-1$, we designed a partitioning based on a sorted *edge list*. In this work, the graph's edge list is first sorted by the edges' source vertex, then evenly distributed. This causes many of the adjacency lists (including hubs) to be partitioned across multiple consecutive partitions. Our edge list partitioned graph supports the following partition-related operations:

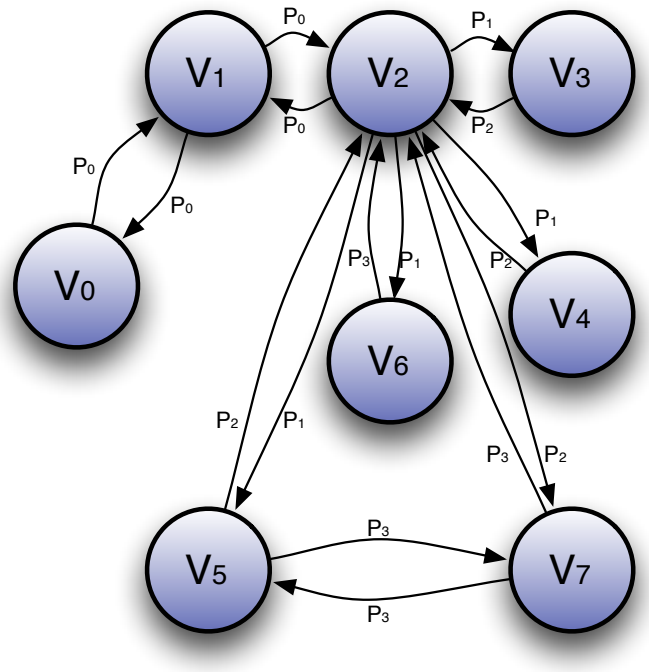
- $min\_owner(v)$ – returns the minimum partition rank that contains vertex v;
- $max\_owner(v)$ – returns the maximum partition rank that contains vertex v.

These operations can be performed in constant time by preserving the rank owner information with the identifier v, or by a $O(log(p))$ binary search. We choose to store the owner information as part of the identifier v. The underlying storage of each edge list partition is flexible; we choose to store each local partition as a *compressed sparse row*.

An example of edge list partitioning is illustrated in Figure 4.1. In this example, vertices 2 and 5 have adjacency lists that span multiple partitions: *min_owner(2) = 0, max_owner(2) = 2, min_owner(5) = 2, max_owner(5) = 3*.

Requiring the edge list to be globally sorted is an additional step that is not needed by 1D or 2D graph partitioning. There exists many distributed memory and external memory sorting algorithms, and in many graph file formats the edge list is already sorted.

Each partition that contains v also contains the algorithm state for v (e.g., BFS level). This means that state is replicated for vertices whose adjacency list spans multiple partitions. The *min_owner* partition is the *master* partition with all others



source	0	1	1	2	2	2	2	2	2	3	4	5	5	6	7	7
target	1	0	2	1	3	4	5	6	7	2	2	2	7	2	2	5
partition	p_0				p_1				p_2				p_3			

Figure 4.1: Example of *edge list partitioning* for a graph with 8 vertices and 16 directed edges, split into 4 partitions. The edge list is globally sorted by the *sources*, then evenly partitioned. For illustration, each edge is labeled according to its partition. In this example vertices 2 and 5 have adjacency lists that span multiple partitions. $min\_owner(2) = 0$, $max\_owner(2) = 2$, $min\_owner(5) = 2$, $max\_owner(5) = 3$.

acting as *replicas*. The algorithms for controlling access to each replica are discussed in Section 4.2.4. The global number of partitioned adjacency lists is bounded by $O(p)$, where each partition contains at most two split adjacency lists.

4.1.1 Ghost Vertices

The ratio of the number of incoming edges to hub vertices in scale-free graphs can grow very large, significantly larger than the total number of edges per partition. To mitigate the communication hotspots created by hubs, we selectively use ghost information. Ghosts can be used to filter excess visitors, reducing the communication hotspots created by high in-degree hubs.

Ghost information replicates distributed state to avoid communication. By replicating the state of vertices with large in-degree, the communication hotspot associated with that vertex can be reduced. The ideal case is to reduce the communication from hundreds of millions down to $O(p)$, where each partition only requires communicating once per hub vertex.

The ghost information is never globally synchronized, and represents only the local partitions' view of remote hubs. Each partition locally identifies high-degree vertices from its edges' targets to become ghost vertices. Ghosts cannot be used for every algorithm, so each algorithm must explicitly declare ghost usage.

We investigate the useful number of ghosts in our experimental study in Section 4.5.6. The number of ghosts required for scale-free graphs is small, because the number of high-degree vertices is small.

Ghosts can only be used as an imprecise filter for algorithms such as BFS, because the ghosts are not globally synchronized. Algorithms that require precise counts of events, such as K-Core and Triangle Counting discussed in Section 4.3, cannot use ghosts.

4.2 Distributed Visitor Queue

The driver of our graph traversal is the *distributed asynchronous visitor queue*; it provides the parallelism and creates a data-driven flow of computation. Traversal algorithms are created using a visitor abstraction, which allows an algorithm designer to define vertex-centric procedures to execute on traversed vertices with the ability to pass visitor state to other vertices. The visitor pattern is discussed in Section 4.2.1.

Each asynchronous traversal begins with an initial set of visitors, which may create additional visitors dynamically depending on the algorithm and graph topology. All visitors are asynchronously transmitted, scheduled, and executed. When the visitors execute on a vertex, they are guaranteed exclusive access to the vertex's data. The traversal completes when all visitors have completed, and the distributed queue is globally empty.

4.2.1 Visitor Abstraction

In our initial work [59], we used an asynchronous visitor pattern to compute Breadth-First Search, Single Source Shortest Path, and Connected Components in shared and external memory. We used multi-threaded *prioritized* visitor queues to perform the asynchronous traversal.

We build on the asynchronous visitor pattern with modifications to handle *edge list partitioning* and *ghost* vertices. The required visitor procedures and state are outlined in Table 4.1.

4.2.2 Visitor Queue Interface

The *visitor queue* has the following functionality that may be used by a visitor or initiating algorithm:

- $push(visitor)$ – pushes a visitor into the visitor queue.

Required	Description
pre_visit( )	Performs a preliminary evaluation of the state and returns *true* if the visit should proceed; this can be applied to *ghost* vertices.
visit( )	Main visitor procedures.
operator<( )	Less than comparison used to locally prioritize the visitors in a *min heap* priority queue.
vertex	Stored state representing the vertex to be visited.

Table 4.1: Visitor Procedures and State

- *do_traversal*() – initializes and runs the asynchronous traversal to completion. This is used by the initiating algorithm.

When an algorithm requires dynamic creation of new visitors, they are *pushed* into the visitor queue using the *push(visitor)* procedure. When an algorithm begins, an initial set of visitors is pushed into the queue, then the *do_traversal()* procedure is invoked and runs the asynchronous traversal to completion.

4.2.3 Example Traversal

To illustrate how an asynchronous traversal algorithm works, we will discuss Breadth-First Search (BFS) at a high level. The details of BFS are discussed in Section 4.3.1. The visitor is shown in Algorithm 12, and the initiating procedure is shown in Algorithm 13.

BFS begins by setting an initial path length for every vertex to ∞ (Alg. 13 line 4). A visitor is queued for the *source* vertex with path length 0, then the traversal begins (Alg. 13 lines 9–11). As the visitors proceed, if they contain a lower path length which is currently known for the vertex, they update the path information and queue new visitors for the outgoing edges (Alg. 12 lines 14–18).

4.2.4 Visitor Queue Design Details

In this section, we discuss the design of the *distributed visitor queue*. The visitor queue has the following functionality, which will be discussed in detail in the following subsections and is shown in Algorithm 5.

- $push(visitor)$ – pushes a visitor into the distributed queue;
- $check\_mailbox()$ – receives visitors from *mailbox* and queues them locally;
- $global\_empty()$ – returns *true* if globally empty;
- $do\_traversal()$ – runs the asynchronous traversal.

mailbox: The communication occurs though a *mailbox* abstraction with the following functionality:

- $send(rank, data)$ – Sends $data$ to $rank$, using the routing and aggregation network;
- $receive()$ – Receives messages from any sender.

ghost information: Our graph has the following ghost related operations used by the distributed visitor queue:

- $has\_local\_ghost(v)$ – returns true if local ghost information is stored for v;
- $local\_ghost(v)$ – returns ghost information stored for v.

push(visitor): This function pushes newly created visitors into the distributed visitor queue; the details are shown in Algorithm 5 (line 5). If the ghost information about the visitor's vertex is stored locally, it is *pre_visited* (line 8). If the *pre_visit* returns *true* or no local ghost information is found, then the visitor is sent to *min_owner*

Algorithm 5 Distributed Visitor Queue

```
1: state: g ← input graph
2: state: mb ← mailbox communication
3: state: my_rank ← local partition rank
4: state: local_queue ← local visitor priority queue

5: procedure PUSH(visitor)
6:     vertex = visitor.vertex
7:     master_partition = g.min_owner(vertex)
8:     if g.has_local_ghost(vertex) then
9:         ghost = g.local_ghost(vertex)
                                          ▷ pre_visit locally stored ghost
10:        if visitor.pre_visit(ghost) then
11:            mb.send(master_partition, visitor)
12:        end if
13:    else
14:        mb.send(master_partition, visitor)
15:    end if
16: end procedure

17: procedure CHECK_MAILBOX
18:    for all visitor ∈ mb.receive() do
19:        vertex = visitor.vertex
20:        if visitor.pre_visit(g[vertex]) then
21:            local_queue.push(visitor)
22:            if my_rank < g.max_rank(vertex) then
                                              ▷ forwards to next replica
23:                mb.send(my_rank + 1, visitor)
24:            end if
25:        end if
26:    end for
27: end procedure

28: procedure GLOBAL_EMPTY                          ▷ quiescence detection [48]
29:    return true if globally empty, else false
30: end procedure

31: procedure DO_TRAVERSAL(source_visitor)
32:    push(source_visitor)
33:    while !global_empty() do
34:        check_mailbox()
35:        if !local_queue.empty() then
36:            next_visitor = local_queue.pop()
37:            next_visitor.visit(g, this)
38:        end if
39:    end while
40: end procedure
```

using the mailbox (lines 11, 14). This functionality abstracts the knowledge of ghost vertex information from the visitor function. If local ghost information is found, the visitor is applied to the ghost. The ghosts act as local filters, reducing unnecessary visitors sent to *hub* vertices.

check_mailbox(): This function checks for incoming visitors from the *mailbox*, and forwards visitors to potential edge list partitioned *replicas*. The details are shown in Algorithm 5 (line 17). For all visitors received from the mailbox (line 18), if the visitor's *pre_visit* returns *true*, the visitor is queued locally (line 21). Additionally, if the vertex is owned by ranks larger than the current rank, the visitor is forwarded to the next replica (line 22). This forwarding chains together vertices whose adjacency lists span multiple edge list partitions. The replicas are kept loosely consistent because visitors are first sent to the master and then forwarded to the chain of replicas in an ordered manner.

global_empty(): This function checks if all global visitor queues are empty, returning *true* if all queues are empty, and is used for *termination detection*. It is implemented using a simple $O(lg(p))$ *quiescence detection* algorithm based on visitor counting [48]. The algorithm performs an asynchronous reduction of the global visitor send and receive count using non-blocking point-to-point MPI communication. It is important to note that to check for non-termination is an asynchronous event, and only becomes synchronous after the visitor queues are already empty.

do_traversal(): This is the driving loop behind the asynchronous traversal process and is shown in Algorithm 5 (line 31). The procedure assumes that a set of initial visitors has been previously queued, then the main traversal loops until all visitors globally have been processed (line 33). During the loop, it checks the *mailbox* for incoming visitors (line 34), and processes visitors already queued locally.

4.2.4.1 External Memory Locality Optimization

The *less than comparison* operation used for local visitor ordering is defined by the algorithm. When using external memory, if two visitors have equal order priority, then they are prioritized to improve locality. In our experiments, the graphs are stored in a *compressed sparse row* format. To improve page-level locality, we order visitors by their vertex identifier when the algorithm does not define an order for a set of visitors. This additional sorting by the vertex identifier improves page-level locality for the graph data stored in NVRAM.

4.3 Algorithms

In this section we discuss three algorithms implemented using our distributed visitor queue framework: breadth-first search, k-core decomposition, and triangle counting. In Section 4.4, we describe an asymptotic analysis framework to express the complexity of the asynchronous traversal in terms of the number of visitors.

4.3.1 Breadth-First Search

The visitor used to compute the BFS level for each vertex is shown in Algorithms 12 and 13. Before the traversal begins, each vertex initializes its *length* to ∞, then a visitor is queued for the source vertex with $length = 0$.

When a visitor *pre_visits* a vertex, it checks if the visitor's length is smaller than the vertex's current length (Alg. 12 line 14). If smaller, the *pre_visit* updates the level information and returns *true*, signaling that the main visit function may proceed. Then, the main *visit* function will send new *bfs_visitors* for each outgoing edge (Alg. 12 line 18).

The *less than comparison* procedure orders the visitors in the queue by *length*

Algorithm 6 BFS Visitor

```
1: visitor state: vertex ← vertex to be visited
2: visitor state: length ← BFS length
3: visitor state: parent ← BFS parent

4: procedure PRE_VISIT(vertex_data)
5:     if length < vertex_data.length then
6:         vertex_data.length ← length
7:         vertex_data.parent ← parent
8:         return true
9:     end if
10:    return false
11: end procedure

12: procedure VISIT(graph, visitor_queue)
13:    if length == graph[vertex].length then
14:        for all vi ∈ out_edges(g, vertex) do
15:            new_vis ← bfs_visitor(vi, length + 1, vertex)
16:            visitor_queue.push(new_vis)
17:        end for
18:    end if
19: end procedure

20: procedure OPERATOR < ()(visitor_a, visitor_b)
                                    ▷ Less than comparison, sorts by length
21:    return visitor_a.length < visitor_b.length
22: end procedure
```

Algorithm 7 BFS Traversal Initiator

```
1: input: graph ← input graph G(V, E)
2: input: source ← BFS traversal source vertex
3: input: vis_queue ← Visitor queue

4: for all v ∈ vertices(graph) parallel do
5:     graph[v].length ← ∞
6:     graph[v].parent ← ∞
7: end for
8: source_visitor ← bfs_visitor(source, 0, source)
9: vis_queue.push(source_visitor)
10: vis_queue.do_traversal()
```

(Alg. 12 line 26). When a set of visitors all contain equal *length*, then the BFS algorithm does not specify an order and the framework can order based on locality, discussed in Section 4.2.4.1.

4.3.2 K-Core Decomposition

Algorithm 8 K-Core Visitor

```
1:  visitor state: vertex ← vertex to be visited
2:  static parameter: k ← k-core requested

3:  procedure PRE_VISIT(vertex_data)
4:      if vertex_data.alive == true then
5:          vertex_data.kcore ← vertex_data.kcore − 1
6:          if vertex_data.kcore < k then
7:              vertex_data.alive ← false
8:              return true
9:          end if
10:     end if
11:     return false
12: end procedure

13: procedure VISIT(graph, visitor_queue)
14:     for all vi ∈ out_edges(g, vertex) do
15:         new_visitor ← kcore_visitor(vi)
16:         visitor_queue.push(new_visitor)
17:     end for
18: end procedure
                                          ▷ No visitor order required
```

To compute the k-core decomposition of an undirected graph, we asynchronously remove vertices from the core whose degree is less than k. As vertices are removed, they may create a dynamic cascade of recursive removals as the core is decomposed.

Algorithm 9 K-Core Traversal Initiator

```
1: input: graph ← input graph G(V, E)
2: input: k ← k-core requested
3: input: vis_queue ← Visitor queue

4: kcore_visitor :: k ← k                          ▷ Set static visitor parameter
5: for all v ∈ vertices(graph) parallel do
6: |   graph[v].alive ← true
7: |   graph[v].kcore ← degree(v, graph) + 1
8: end for
9: for all v ∈ vertices(graph) parallel do
10: |   vis_queue.push(kcore_visitor(v))
11: end for
12: vis_queue.do_traversal()
```

The visitor used to compute the k-core decomposition of an undirected graph is shown in Algorithms 16 and 9. Before the traversal begins, each vertex initializes its *k-core* to *degree(v) + 1* and *alive* to *true*, then a visitor is queued for each vertex.

The visitor's *pre_visit* procedure decrements the vertex's *k-core* number and checks if it is less than k (Alg. 16 line 8). If less, it sets *alive* to false and returns *true*, signaling that the visitors's main *visit* procedure should be executed (Alg. 16 line 10). The *visit* function notifies all neighbors of *vertex* that it has been removed from the k-core (Alg. 16 line 18). After the traversal completes, all vertices whose *alive* equals *true* are a member of the k-core.

4.3.3 Triangle Counting

The visitor used to count the triangles in an undirected graph is shown in Algorithms 10 and 11. Each vertex maintains the count of the number of triangles for which the vertex identifier is the largest member of, initialized to zero.

The visitor's *pre_visit* always returns true; every visitor will execute its *visit* procedure. The *visit* procedure (Alg. 10) has three main duties: first visit (line 8),

length-2 path visit (line 15), and search for closing edge of length-3 cycle (line 22). At each step, the vertices of the triangle are visited in increasing order (lines 10, 17) to prevent the triangle from being counted multiple times. If the closing edge is found, *num_triangles* is incremented (line 24). The global number of triangles can be accumulated after the traversal completes (Alg. 11 line 14).

This algorithm can be extended to count the number of triangles amongst a subset of vertices, or for individual vertices. It can also be extended to use approximate sampling based triangle counting methods [69].

4.4 Asymptotic Analysis

Here we analyze the upper bounds on the number of visitors required for each algorithm. We simplify the analysis by assuming the computation proceeds in synchronized *rounds*; this models the ideal case for the asynchronous system. Our analysis makes the following assumptions, for a graph $G(V, E)$ using p processors:

Parallel Rounds. The asynchronous algorithm proceeds in synchronized parallel *rounds*, in which each processor executes at most one visitor. There is a single shared visitor queue that all p processors access without contention. At the end of each round, the visitor queue is updated with newly queued visitors as necessary. The transmission latency for newly queued visitors is instantaneous, occurring at the end of the round. During a parallel round only one visitor can be selected per vertex, guaranteeing the visitor private access to the vertex. The analysis proceeds by bounding the number of parallel rounds required by the algorithm.These assumptions remove the complexities of distributing a graph amongst p processors, and the communication latencies that may be between the processors. We focus our analysis on the number of visitors required, and the complexity of the visitors.

Algorithm 10 Triangle Count Visitor

```
1: visitor state: vertex ← vertex to be visited
2: visitor state: second ← initialized to ∞
3: visitor state: third ← initialized to ∞

4: procedure PRE_VISIT(vertex_data)
5:     return true
6: end procedure

7: procedure VISIT(graph, visitor_queue)
8:     if second == ∞ then                                  ▷ Visiting first vertex
9:         for all vi ∈ out_edges(graph, vertex) do
10:            if vi > vertex then
11:                new_vis ← tri_count_vis(vi, vertex)
12:                visitor_queue.push(new_vis)
13:            end if
14:        end for
15:    else if third == ∞ then                              ▷ Visiting second vertex
16:        for all vi ∈ out_edges(graph, vertex) do
17:            if vi > vertex then
18:                new_vis ← tri_count_vis(vi, vertex, second)
19:                visitor_queue.push(new_vis)
20:            end if
21:        end for
22:    else                                                 ▷ Search for closing edge
23:        if third ∈ out_edges(graph, vertex) then
24:            graph[vertex].num_triangles += 1
25:        end if
26:    end if
27: end procedure
                                                            ▷ No visitor order required
```

Algorithm 11 Triangle Count Traversal Initiator

```
1: input: graph ← input graph G(V, E)
2: input: vis_queue ← Visitor queue

3: for all v ∈ vertices(graph) parallel do
4:   | graph[v].num_triangles = 0
5: end for
6: for all v ∈ vertices(graph) parallel do
7:   | vis_queue.push(tri_count_vis(v))
8: end for
9: vis_queue.do_traversal()
10: local_count = 0
11: for all v ∈ vertices(graph) parallel do
12:   | local_count+ = graph[v].num_traingles
13: end for
14: global_count = all_reduce(local_count, SUM)
15: return global_count
```

Graph Properties. The underlying graph properties may have a significant impact on the complexity of algorithms. We use the following list of graph properties to parameterize our analysis:

- D – The Graph's diameter;
- d_{max}^{out} – Maximum out-degree, $\max_{v \in V} (\textit{out-degree}(v))$;
- d_{max}^{in} – Maximum in-degree, $\max_{v \in V} (\textit{in-degree}(v))$.

4.4.1 Analysis of BFS

Each parallel round executes up to p visitors, however only one of the visitors is guaranteed to belong to a shortest or critical path. For a connected graph, the length of the shortest path, and also the number of required parallel rounds is proportional to the diameter of the graph. The total number of parallel rounds is bounded by $\Theta(D + \frac{|E|}{p} + d_{max}^{in})$ without the use of ghosts. When using ghosts, the term d_{max}^{in}

decreases to p, because the ghosts filter the high-degree visitors to one per partition. With ghosts, the number of parallel rounds is bounded by $\Theta(D + \frac{|E|}{p} + p)$.

4.4.2 Analysis of K-Core

Similarly to BFS, each parallel round executes up to p visitors, however only one of the visitors is guaranteed to belong to the critical path. For a connected graph, the length of the critical path, and the number of required parallel rounds, is proportional to the diameter of the graph. Unlike BFS, k-core cannot use ghost vertices for filtering, therefore the largest hub will require processing d^{in}_{max} visitors The total number of parallel rounds is bounded by $\Theta(D + \frac{|E|}{p} + d^{in}_{max})$.

4.4.3 Analysis of Triangle Counting

The visitor for triangle counting performs three basic duties: first visit, length-2 path visit, and search for closing edge of length-3 cycle. The *first visit* duty is performed for every vertex in the graph, and these visitors create *length-2 path* visitors. Each edge in the graph will have a corresponding *length-2 path* visitor, and these visitors will create at most $O(d^{out}_{max})$ visitors to search for the enclosing *length-3 cycle*. Triangle counting cannot use ghost vertices for filtering, discussed in Section 4.1.1 therefore the largest hub will require processing d^{in}_{max} visitors. The total number of parallel rounds is bounded by $O(\frac{|E|d^{out}_{max}}{p} + d^{in}_{max})$.

4.5 Experimental Study

In this section we experimentally evaluate the scalability of our approach both in distributed memory and distributed external memory. We demonstrate that our approach is scalable in two dimensions: it is scalable to large processor count for leadership class supercomputers, and it is scalable to distributed external memory using emerging HPC clusters containing node-local NVRAM. We also demonstrate

the effects of using *edge list partitioning* and *ghosts* to mitigate the effects of high-degree hubs.

Currently, large scale HPC clusters with node-local NVRAM are not readily available; therefore we demonstrate scalability using two sets of experiments. To show scalability to large core count, we performed experiments using IBM BG/P supercomputers up to 131K cores; these experiments do not use the external memory aspect of the algorithm. BG/P experiments were performed using Intrepid at Argonne National Laboratory and uDawn at Lawrence Livermore National Laboratory. Next, to demonstrate external memory scalability on distributed memory clusters, we performed experiments on Hyperion-DIT at LLNL; Hyperion has node-local NVRAM.

4.5.1 Experimental Setup

We implemented our distributed visitor queue and routed mailbox in C++ using only non-blocking point-to-point MPI communication. For our external memory experiments, where the graph data is completely stored in NVRAM, we used our custom user-space *page cache* to interface with NVRAM, discussed in Section 2.5.3. We show experiments using three synthetic graph models discussed in Section 2.4.

4.5.2 Scalability on BG/P Supercomputer

We demonstrate the scalability using IBM BG/Ps at the Argonne National Laboratory and Lawrence Livermore National Laboratory. Intrepid was ranked 5^{th} on the November 2011 and 15^{th} June 2012 Graph500 list, with an efficient high performance custom implementation of the benchmark.

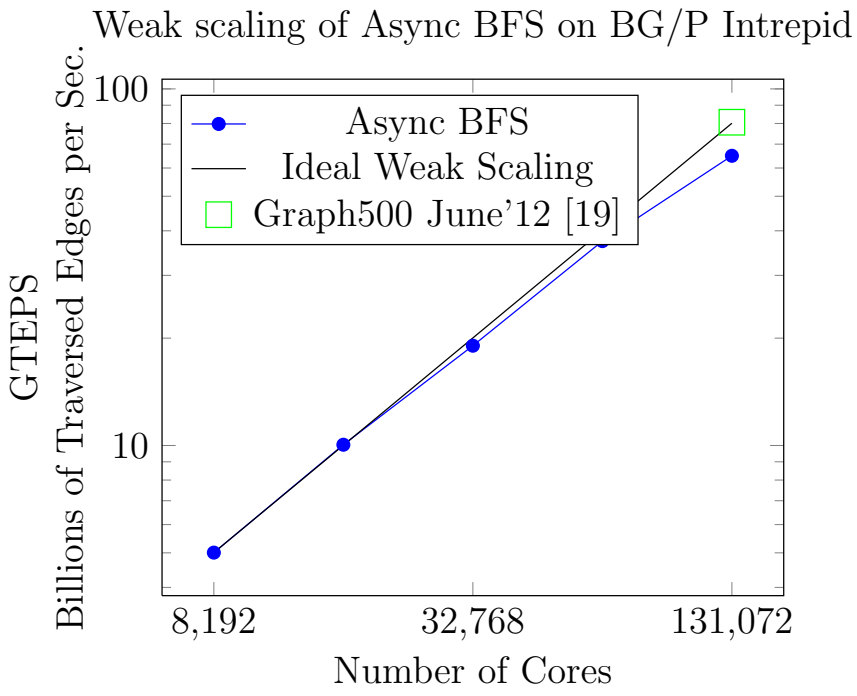


Figure 4.2: Weak scaling of Asynchronous BFS on BG/P Intrepid. Compared to Intrepid BFS performance from the Graph500 list. There are 2^{18} vertices per core, with the largest scale graph having 2^{35}.

4.5.2.1 BFS

We have scaled our BFS algorithm up to 131K cores using the 3D routed mailbox discussed in Section 2.6.1. We achieved excellent weak scaling as shown in Figure 4.2. In addition to showing weak scalability, we demonstrate the efficiency of our implementation by comparing to the current best known Graph500 result for Intrepid from the June 2012 list [19]. Our approach, designed to use portable MPI and external memory, achieved 64.9 GTEPS with 2^{35} vertices, which is only 19% slower than the best known BG/P implementation.

We use this experiment to establish the scalability of our approach, and the efficiency of our implementation. This experiment forms the basis for the NVRAM vs. DRAM experiments we performed on Hyperion-DIT.

4.5.2.2 K-Core Decomposition

We show weak scaling of k-core decomposition on BG/P up to 131K cores using RMAT graphs in Figure 4.3. The time to compute the cores 4 and 16 are shown for each graph size. Our techniques enable near linear weak scaling for computing k-core.

4.5.2.3 Triangle Counting

We show weak scaling of triangle counting on BG/P up to 131K cores using Small World graphs in Figure 4.4. We show the time to count the triangles on small world graphs with rewire probabilities 0%, 10%, 20%, and 30%. The small world generator creates vertices with a uniform vertex degree (in this case 32). As will be discussed in Section 4.5.4, the performance of triangle counting is dependent on the maximum vertex degree of the graph. For this weak scaling study, we use small world graphs

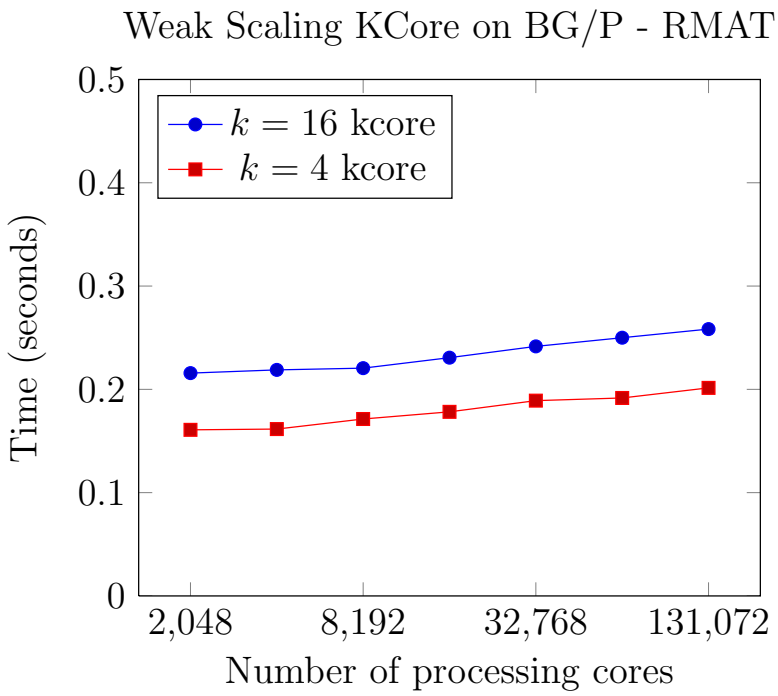


Figure 4.3: Weak Scaling of kth-core on BG/P using RMAT graphs. Time shown to compute cores 4 and 16. There are 2^{16} vertices and 2^{20} undirected edges per core; at 131K cores, the graph has 2^{33} vertices and 2^{37} edges.

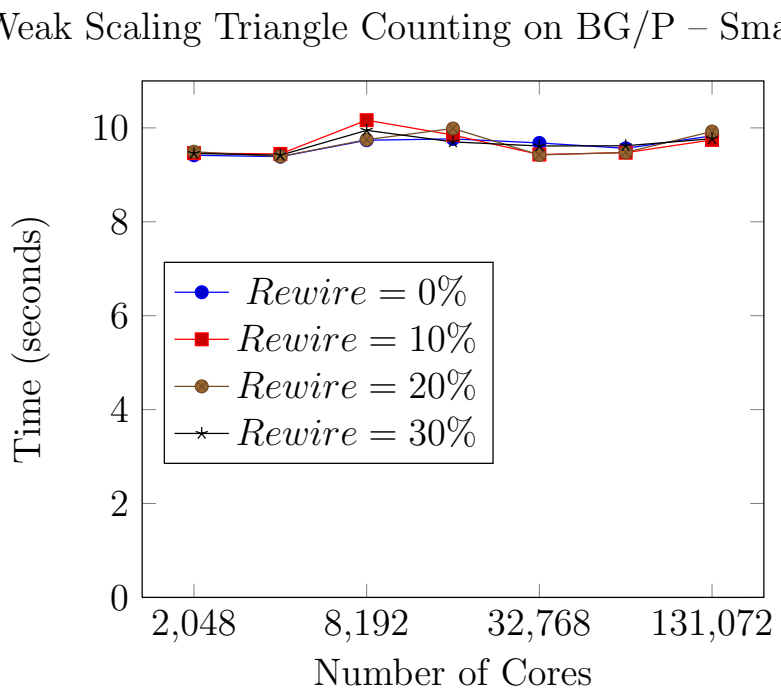


Figure 4.4: Weak scaling of triangle counting on BG/P using Small World graphs. Performance shown at with different small world rewire probabilities. There are 2^{12} vertices and 2^{16} undirected edges per core; at 131K cores, the graph has 2^{29} vertices and 2^{33} edges.

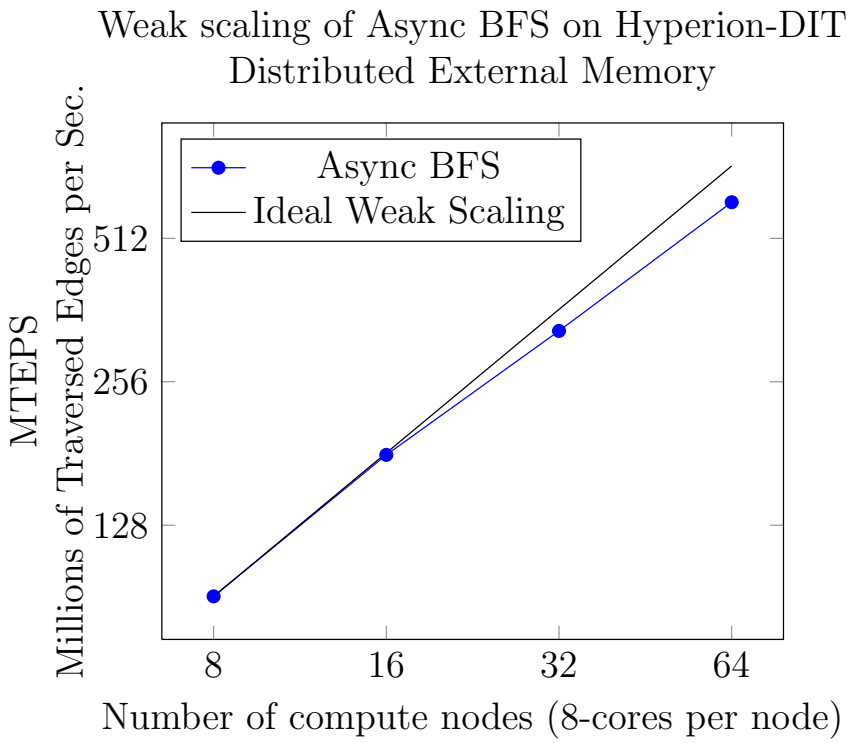


Figure 4.5: Weak scaling of distributed external memory BFS on Hyperion-DIT. Each compute node has 8-cores, 24GB DRAM, and is using 169GB NAND Flash to store graph data. There are 17B edges per compute node; the largest scale graph has over one-trillion edges and 2^{36} vertices.

to isolate the effects of hub growth that would occur with PA or RMAT graphs.

4.5.3 Scalability of Distributed External Memory BFS

We demonstrate the distributed external memory scalability of our BFS algorithm on the Hyperion-DIT cluster at Lawrence Livermore National Laboratory (LLNL). The Hyperion-DIT is an 80-node subset of Hyperion that is equipped with node-local Fusion-io NAND Flash. Each compute node has 8 cores, 24 GB DRAM, and 600 GB NVRAM.

We performed a weak scaling study, in which each compute node stores 17 billion

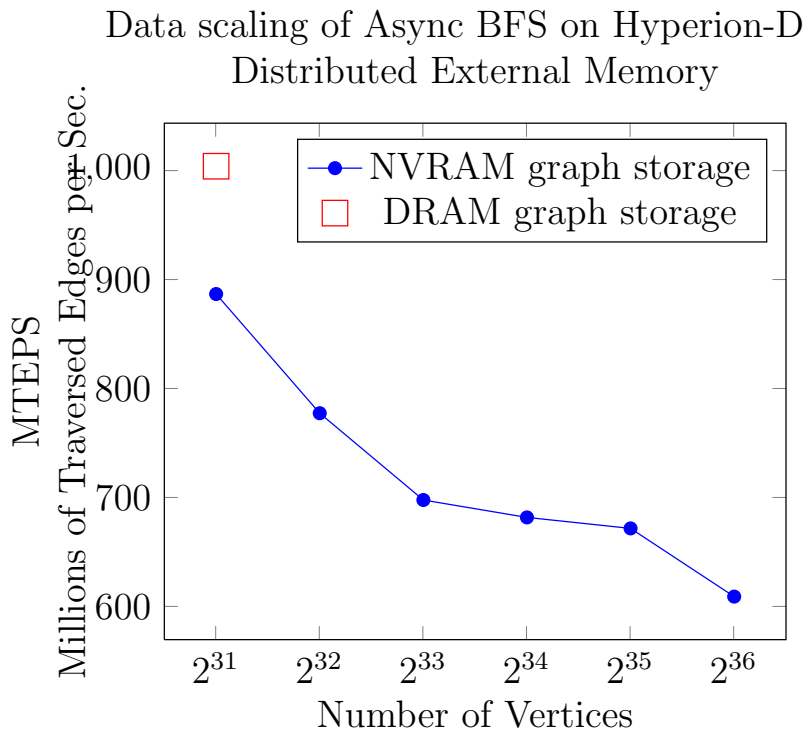


Figure 4.6: Effects of increasing external memory usage on 64 compute nodes of Hyperion-DIT. At 2^{36}, which is 32x larger data than DRAM-only, the NVRAM performance is only 39% slower than DRAM graph storage.

Machine Name	Location	Machine Type	Graph Storage	$\|V\|$	MTEPS
Hyperion-DIT	LLNL	64 nodes, 512 cores	DRAM	2^{31}	1,004
			Fusion-io	2^{36}	609
Trestles	SDSC	144 nodes, 4608 cores	SATA SSD	2^{36}	242
Leviathan	LLNL	single node, 40 cores	Fusion-io	2^{36}	52

Table 4.2: November 2011 Graph500 results using NAND Flash. Scale 36 is a graph with over 1 trillion edges.

edges on its local NVRAM, roughly 169GB in a compressed sparse row format. The results of the weak scaling are shown in Figure 4.5; at 64 compute nodes the graph has over one trillion undirected edges and 2^{36} vertices, twice the size as experiments on BG/P Intrepid. We expect our approach to continue scaling as larger HPC clusters with node-local NVRAM become available.

To experiment with the effects of increasing NVRAM usage per compute node, we performed an experiment where the computational resources are held constant while the data size increases. The results of our data scaling experiment are shown in Figure 4.6. As the data size increases, from 34 billion to 1 trillion edges (10.8 TB), the additional data we store on NVRAM results in only a moderate performance reduction. The overall performance reduction from DRAM-only to 32x larger graph stored in NVRAM is only 39%. This is a significant increase in data size with only a moderate performance degradation.

Results using our distributed external memory BFS for the Graph500 are shown in Table 4.2. In addition to the Hyperion-DIT cluster, we performed experiments on Trestles at the San Diego Supercomputing Center (SDSC) and Leviathan (LLNL). Each compute node in Trestles has commodity SATA SSDs, and shows that our approach is not limited to enterprise class NVRAM. Leviathan is a single-node system using implementations from our previous multithreaded work [59]. Leviathan has

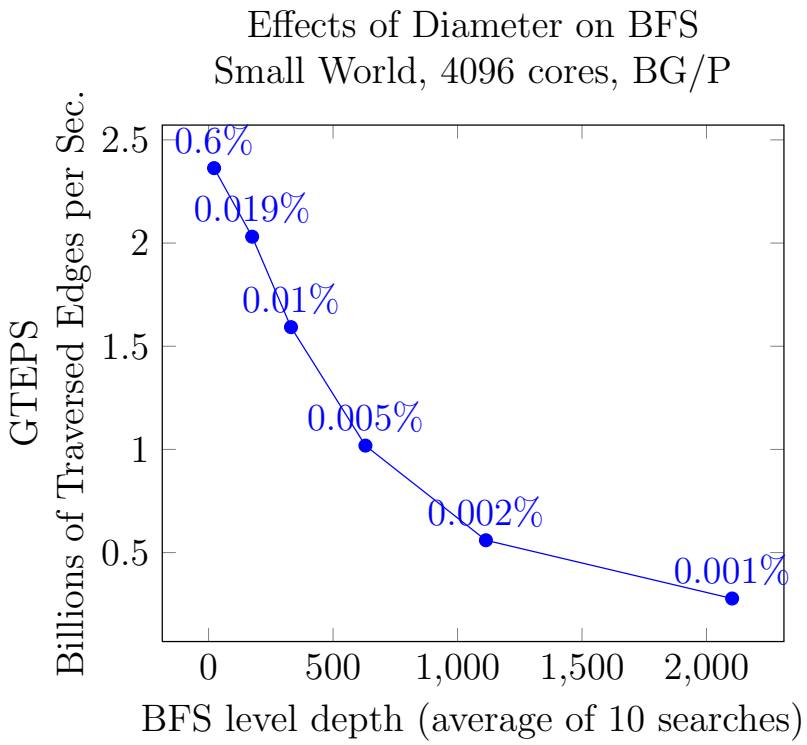


Figure 4.7: Effects of diameter on BFS performance. Small World graph model with varying *random rewire* probabilities (shown above each point). BFS level depth used for x-axis. Computational resources fixed at 4096 cores of BG/P. Graph size is fixed at 2^{30} vertices and 2^{34} undirected edges.

1TB of DRAM and 12TB of Fusion-io in a single host. These represent 3 of 8 total systems on the November 2011 Graph500 list traversing trillion-edge graphs.

4.5.4 Topological Effects on Performance

The performance of BFS and triangle counting are dependent on topological properties of the graph. For BFS, the performance is dependent on the diameter (the longest shortest-path) of the graph. Using the Small World graph generator, we show the effects of increasing diameter on BFS in Figure 4.7. By decreasing the small-world random rewire probability while keeping the size of the graph constant,

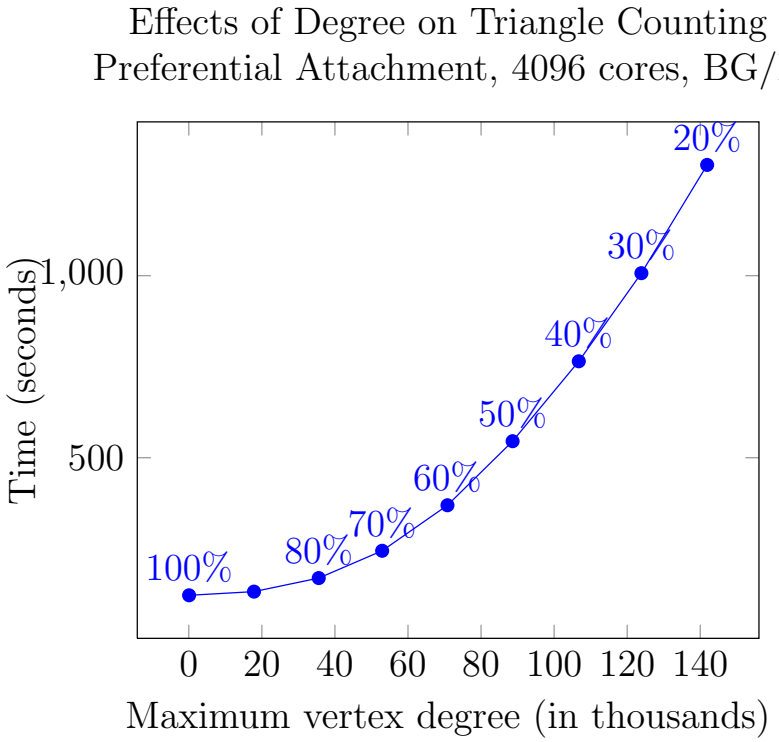


Figure 4.8: Effects of vertex degree on Triangle Counting performance. Preferential Attachment graph model with varying *random rewire* probabilities (shown above each datapoint). Maximum vertex degree used for x-axis. Computational resources fixed at 4096 cores of BG/P. Graph size is fixed at 2^{28} vertices and 2^{32} undirected edges.

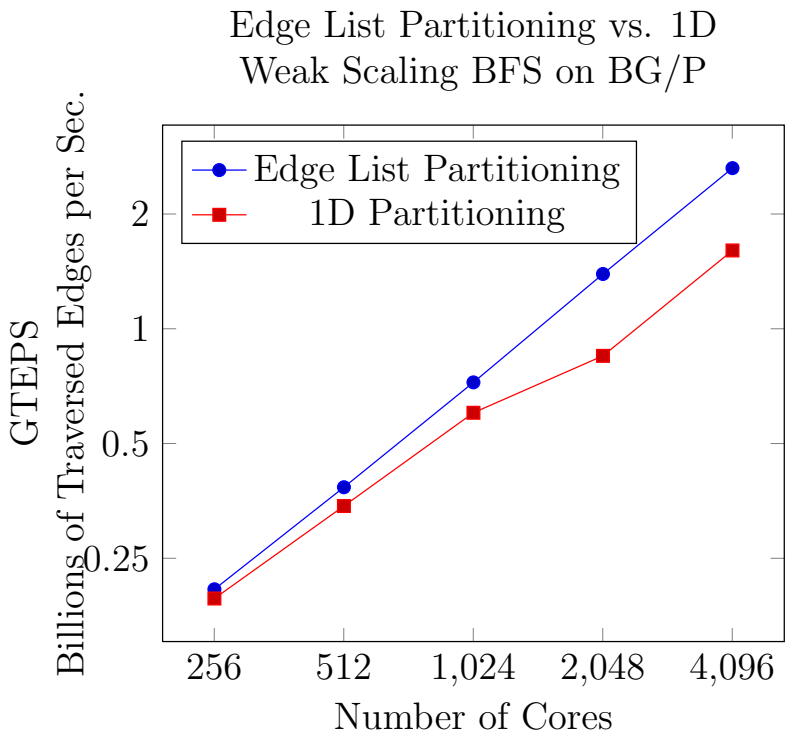


Figure 4.9: Comparison of *edge list partitioning* vs 1D. Performance of BFS on RMAT graphs shown on BG/P. Important note: the graph sizes are reduced to prevent 1D from running out of memory. There are 2^{17} vertices and 2^{21} undirected edges per core.

the diameter of the graph increases. The increasing diameter causes the performance of BFS to decrease. Similarly, the performance of triangle counting is dependent on the maximum vertex degree. Using the Preferential Attachment graph generator with an added step of random rewire, we show the effects of increasing hub degree while keeping the graph size constant in Figure 4.8. The topological effects can also be seen in the algorithm analysis in Sections 4.4.1 and 4.4.3.

4.5.5 Edge List Partitioning vs 1D

To demonstrate the effects of *edge list partitioning* vs. traditional 1D partitioning we show the weak scaling of both with BFS on RMAT graphs on BG/P in Figure 4.9.

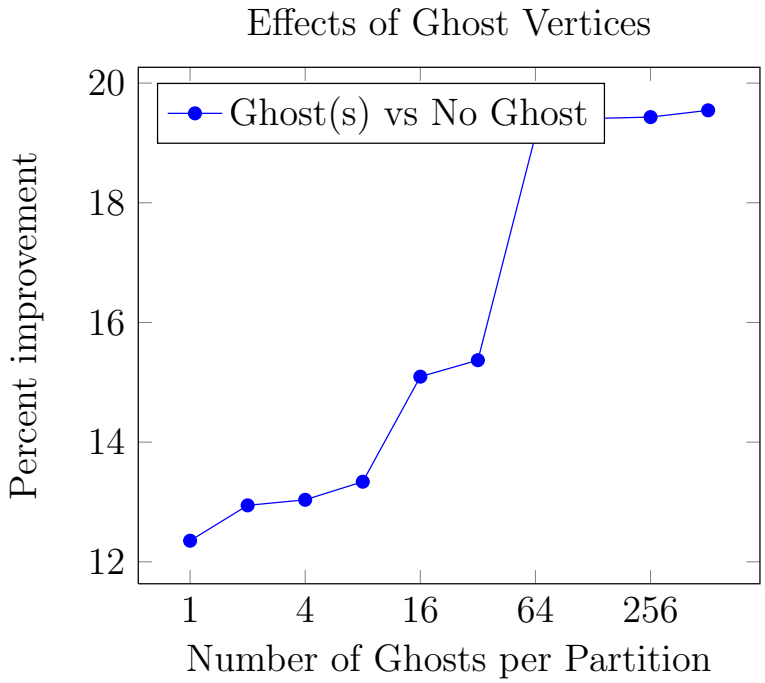


Figure 4.10: Experiment showing the percent improvement of ghost vertices vs. no ghost vertices. Results from 4096 cores of BG/P on a graph with 2^{30} vertices.

Because 1D partitioning suffers from data imbalance, the graph sizes in the experiments were reduced to prevent 1D from running out of memory. The weak scaling of edge list partitioning is almost linear, while 1D suffers slowdowns from the partition imbalance.

4.5.6 Use of Ghost Vertices

We experimented with the effects on BFS performance of choosing a different size k on a graph with 2^{30} vertices on 4096 cores of BG/P. The results of this experiment are shown in Figure 4.10, where the percent improvement of k ghosts per partition vs. no ghosts is shown. Using a single ghost shows more than a 12% improvement, and 512 ghosts shows an 19.5% improvement. This improvement is highly dependent on

the graph, and as hubs grow to larger sizes it may have a larger effect. For scale-free graphs, when $degree(v) > p$, there is an opportunity for ghosts to have a positive effect, because at least one partition will have multiple edges to v. All other BFS experiments in this work use 256 ghost vertices per partition.

4.6 Summary

Our work focuses on an important *data intensive* problem of massive graph traversal. We aim to scale to trillion-edge graphs using both leadership class supercomputers and node-local NVRAM that is emerging in many new HPC systems.

We address scaling challenges, created by scale-free power-law degree distributions, by applying an *edge list partitioning* strategy, and show its application to three important graph algorithms: BFS, K-Core, and triangle counting.

We demonstrate the scalability of our approach on up to 131K cores of BG/P Intrepid, and we show that by leveraging node-local NAND Flash, our approach can process 32x larger datasets with only a 39% performance degradation in TEPS.

Our work breaks new ground for using NVRAM in the HPC environment for data intensive applications. We show that by exploiting both distributed memory processing and node-local NVRAM, significantly larger datasets can be processed than with either approach in isolation. Further, we demonstrate that our asynchronous approach mitigates the effects of both distributed and external memory latency. The architecture and configuration of NVRAM in supercomputing clusters is an active research topic. To our knowledge, our work is the first to integrate node-local NVRAM with distributed memory at extreme scale for important data intensive problems.

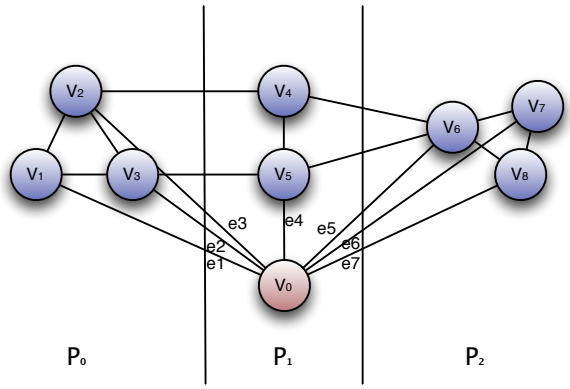
5. DISTRIBUTED STORAGE, COMPUTATION, AND COMMUNICATION OF HIGH-DEGREE VERTICES

In this chapter, we present new techniques to distribute the storage, computation, and communication for hubs in large scale-free graphs. To balance the processing workload, we distribute hub vertex data structures and related computation among a set of delegates. An illustration of a graph before and after partitioning the hub is shown in Figure 5.1. Each partition containing a portion of the hub is assigned a local representative of the hub. One representative is distinguished as the *controller*, and the others are the *delegates*. The controller and its delegates coordinate using asynchronous broadcast and reduction operations rooted at the controller.

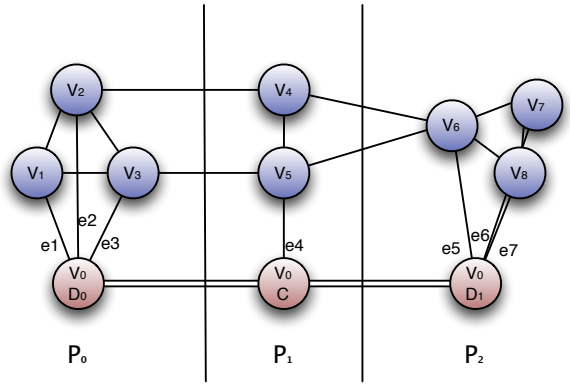
Our delegate technique leads to significant communication reduction through the use of asynchronous broadcast and reduction operations. For hubs whose degree is greater than the number of processing cores, p, using delegates reduces the required volume of communication. This reduction occurs because a broadcast, rooted at the controller, requires only $O(p)$ communication, while without delegates the volume of communication is proportional to the degree of the hub.

Our distributed delegate approach extends our previous work of an asynchronous visitor model [59, 60], discussed in Chapters 3 and 4. Using the visitor computation model, the controller may broadcast visitors to all its delegates. Similarly, the delegates may participate in an asynchronous reduction rooted at the controller.

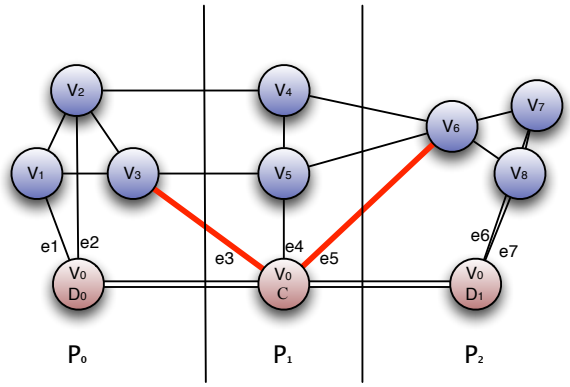
We demonstrate the approach and evaluate performance and scalability using Breadth-First Search (BFS), Single Source Shortest Path (SSSP), K-Core Decomposition, and PageRank on synthetically generated scale-free graphs. The data-intensive community has identified BFS as a key challenge for the field and estab-



(a) 1D Partitioning



(b) Distributed Delegates Partitioning. P_1 contains the *controller*.



(c) Distributed Delegates Partitioning after balancing.

Figure 5.1: Comparison of 1D partitioning vs. distributed delegates partitioning for the same graph. In 1D partitioning (a), V_0 is a high-degree vertex that maps to a single partition and may lead to imbalances. In distributed delegates partitioning (b), V_0 is distributed across multiple partitions while low-degree vertices remain 1D partitioned. V_0C is the *controller* assigned to P_1. Delegates V_0D_0 and V_0D_1 are assigned to P_0 and P_2. After balancing (c), edges e_3 and e_5 have been relocated to balance the partitions, and are now delegate cut edges.

lished it as the first kernel for the Graph500 benchmark [20]. We demonstrate scalability up to 131K cores using the IBM BG/P supercomputer, and show portability on a standard HPC Linux cluster. We compare our work to existing approaches for processing scale-free graphs in distributed memory, most notably 2D graph partitioning, [16, 82] by comparing our algorithm to the best known Graph500 performance on IBM BG/P Intrepid supercomputer at Argonne [19].

Summary of contributions presented in this chapter:

- We present a new algorithmic technique, distributed delegates, designed to load balance the computation, communication, and storage associated with high-degree vertices;
- We demonstrate our techniques using Breadth-First Search (BFS), Single Source Shortest Path (SSSP), K-Core Decomposition, and Page-Rank;
- We demonstrate excellent scalability up to 131K cores on BG/P Intrepid, and portability on a standard HPC Linux cluster;
- Our algorithm improves the best known Graph500 results for BG/P Intrepid, a custom BG/P implementation, by 15%.

5.1 Distributed Delegates

To balance the storage, computation, and communication of high-degree hubs, we distribute hub data structures and related computation amongst many partitions. Each partition containing a portion of the hub data structure is assigned a local representative; one representative is distinguished as the *controller*, and the others are designated as *delegates*. An illustration of a delegate partitioned graph is shown in Figure 5.1(b). The controller and its delegates communicate via asynchronous broadcast and reduction operations rooted at the controller.

The delegates maintain a copy of the state for the vertex and a portion of the adjacency list of the vertex. Because a delegate only contains a subset of a vertex's edges, the operations performed may need to be coordinated across multiple delegates.

5.1.1 Delegate Partitioning in Visitor Framework

We have integrated delegate partitioning into our asynchronous visitor framework. Vertices with degree greater than d_{high} are distributed and assigned delegates, while vertices with low-degree are left in a basic 1D partitioning. When a visitor visits a delegate, it only operates on the subset of adjacent edges managed by the delegate; it does not operate on the entire distributed adjacency list.

Ideally, the outgoing edges of high-degree vertices are stored at the edges' target vertex location. Such edges are called *co-located* because their delegate and target vertex reside on the same partition. Co-located edges do not require additional communication beyond the delegates' broadcast and reduction communication, so having multiple co-located edges per individual delegate leads to an overall reduction in communication.

This technique alone is not sufficient to produce balanced partitions. In some cases, including our experiments, simply storing high-degree edges at the edges' target vertex location can lead to imbalance amongst the delegates. To balance partitions, the delegated edges belonging to high-degree vertices can be moved to *any* partition at the cost of additional communication for the non-co-located edges.

5.1.2 Distributed Delegate Partitioning

In this section, we describe a simple technique to partition an input graph using distributed delegates. A distributed input graph, $G(V, E)$ with vertex set V and edge set E, is partitioned into p partitions in three steps. First, the high degree

vertices in G are identified. Second, the edges in E belonging to low-degree vertices are 1D partitioned such that all of a low-degree vertex's edges reside on a single partition. The edges in E belonging to high-degree hubs are partitioned according to the partition of the edge target vertex. Finally, the partitions are balanced by offloading delegate edges from partitions with an above average number of edges.

The input distributed edge set, E, is assumed to be unordered, and is distributed over the p partitions. Undirected edges are represented by creating directed forward and backward edges that may reside on different partitions.

First, the high-degree vertices with degree larger than a threshold, d_{high}, are identified. Then the distributed edge set, E, is partitioned into two distributed edge sets: E_{high} for edges whose source vertex is high-degree, and E_{low} for edges whose source vertex's degree is less than d_{high}. Delegates are created on all partitions for high-degree vertices. The degree of every vertex must be accumulated to identify the high-degree vertices, which may require an all-to-all exchange amongst the partitions.

The second step uses a simple vertex-to-partition mapping (e.g., round-robin) to define a 1D partitioning. The edges in E_{low} are distributed according to the partition mapping of the source vertex of each edge. The edges in E_{high} are distributed according to the target vertex partition mapping of each edge. In the worst case, every edge will need to be relocated to a new partition which may require an all-to-all exchange amongst the partitions.

The third step attempts to correct partition imbalances. The number of edges locally assigned to each partition (both E_{high} and E_{low}) can be imbalanced. An edge in E_{high} may be reassigned to any partition, because the edge's source is a delegated vertex. A new distributed edge set $E_{overflow}$ is created and filled with edges of E_{high} from partitions with greater than $\frac{|E|}{p}$ edges. The edges in $E_{overflow}$ are distributed such that the local partitions' sum of edges $|E_{low}| + |E_{high}| + |E_{overflow}| = \frac{|E|}{p}$. For

Behavior	Description	Complexity	Examples
pre_visit_parent	Visitor is sent to parent delegate and executes *pre_visit*. If *pre_visit* returns *true*, visitor continues to visit parents until the *controller* is reached.	$O(h_{tree})$	BFS, SSSP
lazy_merge_parent	Lazily merges visitors using an asynchronous reduction tree. Merges visitors locally, and sends to parent in reduction tree when local visitor queue is idle. When *controller* is reached, normal visitation proceeds. Requires that visitors provide a *merge* function.	$O(h_{tree})$	k-core
post_merge	Visitors are merged into parent reduction tree after traversal completes. Requires that visitors provide a *merge* function.	$O(h_{tree})$	PageRank

Table 5.1: Delegate Visitor Behaviors

performance reasons, minimizing the size of $E_{overflow}$ is desirable, because the edges are located on different partitions than their targets. An illustration of a delegate partitioned graph after edge balance is shown in Figure 5.1. Here, edges e_3 and e_5 have been relocated to partition p_1 to balance the partitions, and edges e_3 and e_5 are now delegate cut edges. In the worst case, each partition will either send or receive overflow edges and may require an all-to-all exchange amongst the partitions.

The complete partitioning can be accomplished in three parallel operations over the edges, $O(\frac{|E|}{p})$. In the worst case, each step may require all-to-all communication, $O(p^2)$. This partitioning cost is asymptotically the same as partitioning an unorganized edge set using 1D or 2D partitioning. In our weak-scaling experiments shown in section 5.5.2, the delegate partitioning produced evenly balanced partitions.

Behavior	Description	Complexity
bcast_delegates	Controller broadcasts the current visitor to all delegates.	$O(h_{tree})$
terminate_visit	Controller terminates the current visitor without sending to delegates.	$\Theta(1)$

Table 5.2: Controller Visitor Commands

Required	Description
pre_visit( )	Performs a preliminary evaluation of the state and returns *true* if the visitation should proceed, this can be applied to *delegate* vertices.
visit( )	Main visitor procedures.
operator<( )	Less than comparison used to locally prioritize the visitors in a *min heap* priority queue.
vertex	Stored state representing the vertex to be visited.
delegate_behavior	Desired delegate visitation behavior, see Table 5.1.
merge(visitor_a, visitor_b)	Returns the merge of two visitors. Used for *lazy_merge_parent* and *post_merge* behaviors.

Table 5.3: Visitor Procedures and State

5.2 Asynchronous Visitor Queue

The driver of our graph traversal is the *distributed asynchronous visitor queue* [60]; it provides the parallelism and creates a data-driven flow of computation. Traversal algorithms are created using a visitor abstraction, which allows an algorithm designer to define vertex-centric procedures to execute on traversed vertices with the ability to pass visitor state to other vertices.

5.2.1 Visitor Abstraction

In our earlier work presented in Chapter 4, we used an asynchronous visitor pattern to compute Breadth-First Search, Single Source Shortest Path, Connected Com-

ponents, k-core, and triangle counting in shared, distributed and external memory. We used *edge-list partitioning* and *ghosts* to address the scaling challenges created by high-degree vertices [60]. We showed these techniques to be useful, however the application of *ghosts* was limited to simple traversals such as BFS.

In this chapter, we build on the asynchronous visitor pattern and introduce new techniques designed to handle distributed delegates. The coordination of the controller and its delegates must be considered when designing a visitor for an algorithm. The algorithm developer must specify a *delegate behavior* for each visitor, and *controller commands* must be specified at the return of the visitor's procedure. A list of delegate behaviors is described in Table 5.1, and a list of controller commands is described in Table 5.2. There are three types of reduction operations, *pre_visit_parent*, *lazy_merge_parent*, and *post_merge*, that allow algorithms to distribute computation amongst the delegates. For the controller, there is a broadcast operation, *bcast_delegates*, that broadcasts a visitor to all the delegates of the controller.

The visitor procedures required by our asynchronous visitor queue framework are summarized in Table 5.3. When executing, the visitors have exclusive access to the vertex's data.

5.2.2 Visitor Queue Interface

The *visitor queue* has the following functionality that may be used by a visitor or initiating algorithm:

- *push(visitor)* – pushes a new visitor into the distributed queue.
- *do_traversal()* – initializes and runs the asynchronous traversal to completion. This is used by the initiating algorithm.

When an algorithm needs to dynamically create new visitors, they are *pushed* onto the visitor queue using the *push()* procedure. When an algorithm begins, an initial set of visitors are pushed onto the queue, then the *do_traversal()* procedure is invoked which runs the asynchronous traversal to completion.

To support efficient broadcast and reduction operations, the distributed delegates for a vertex are arranged in a tree structure (a *delegate tree*) with the root of the tree defined as the *controller*. The height of the delegate tree is denoted by h_{tree}, and the value of h_{tree} for out experiments is discussed in Section 5.2.4.

5.2.3 Controller and Delegate Coordination

Operations on the controller and its delegates are coordinated through asynchronous broadcast and reduction operations. The return value of the *visit* procedure notifies the framework which controller action it is required to perform. A controller can broadcast commands to all delegates of a vertex by returning *bcast_delegates* from the *visit* procedure. The controller may choose to not broadcast a visitor by returning *terminate_visit* from the *visit* procedure.

Delegates can lazily participate in reductions by using the *lazy_merge_parent* behavior. This instructs the visitor framework to locally merge visitors, and send a merged visitor to the parent in the reduction tree when local visitor queue is idle. We show K-Core decomposition as an example algorithm using this behavior. To fully reach the controller, requires $O(h_{tree})$ visits.

Asynchronous filtering can be performed using the *pre_visit_parent* behavior. This tells the framework to immediately send the visitor to the delegate's parent where the *pre_visit* procedure will be executed. If the *pre_visit* returns *true* the visit will proceed up the delegate tree. We show Breadth-First Search as an example algorithm using this behavior.

Post-traversal reductions are performed when the visitor's behavior is set to *post_merge*. This tells the framework to merge the visitors into the parent reduction tree after the traversal completes. PageRank is an algorithm using this behavior.

5.2.4 Routed Point-to-Point Communication

In our earlier work, we applied communication routing and aggregation through a synthetic network to reduce dense communication requirements [60]. For dense communication patterns, where every processor needs to send messages to all p other processors, we route the messages through a topology that partitions the communication. Figure 5.2 illustrates a 2D routing topology that reduces the number of communicating channels a processor requires to $O(\sqrt{p})$. This reduction in the number of communicating pairs comes at the expense of message latency because messages require two hops to reach their destination. In addition to reducing the number of communicating pairs, 2D routing increases the amount of message aggregation possible by $O(\sqrt{p})$.

In this technique, we embed the *delegate tree* into the synthetic routed communication topology, as illustrated in Figure 5.2. In this example, delegates residing on *Rank 11* are assigned *delegate parents* on *Rank 9* when the controller is on *Rank 5*. A *pre_visit_parent* originating on *Rank 11* is sent to the parent on *Rank 9* before being sent to the controller on *Rank 5*. An illustration of a broadcast tree is also shown for *Rank 5*. When *Rank 5* broadcasts, it first sends to the first level {*4,5,6,7*}. The second level of the broadcast is illustrated for *Rank 7*, which sends to {*12,13,14,15*}. The value of h_{tree} is 2 when using 2D partitioning; with 3D it is 3.

Scaling to hundreds of thousands of cores requires additional reductions in communication channels. Our experiments on IBM BG/P use a 3D routing topology that is very similar to the 2D illustrated in Figure 5.2, and on the BG/P, our routing

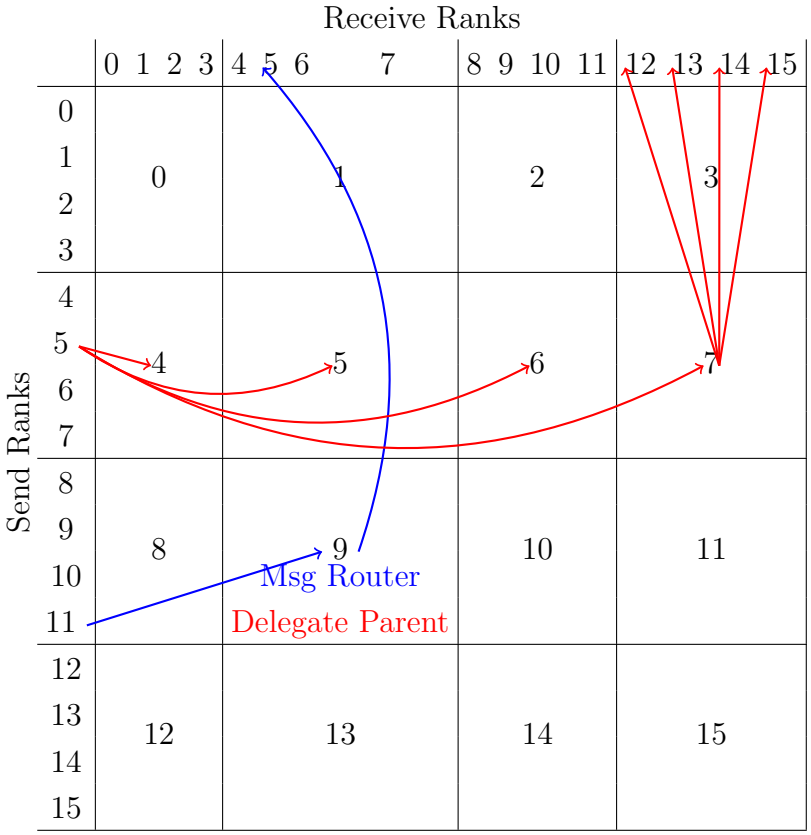


Figure 5.2: Illustration of 2D communicator routing of 16 ranks with distributed delegate operations. As an example, when *Rank 11* sends to *Rank 5*, the message is first aggregated and routed through *Rank 9*. Delegate tree operations are also embedded onto this topology. In this example, delegates residing on *Rank 11* are assigned *delegate parents* on *Rank 9* when the controller is on *Rank 5*. A *pre_visit_parent* originating on *Rank 11* is sent to the parent on *Rank 9* before being sent to the controller on *Rank 5*. An illustration of a broadcast tree is also shown for *Rank 5*. When *Rank 5* broadcasts, it first sends to the first level {*4,5,6,7*}. The second level of the broadcast is illustrated for *Rank 7*, which sends to {*12,13,14,15*}.

is designed to mirror the 3D torus interconnect topology.

5.3 Visitor Algorithms

In this section we discuss three algorithms implemented using distributed delegates: Breadth-First Search, Single Source Shortest Paths, K-Core Decomposition, and PageRank.

5.3.1 Breadth-First Search & Single Source Shortest Path

The visitor used to compute the BFS level or SSSP for each vertex is shown in Algorithms 12 and 13. Before the traversal begins, each vertex initializes its *length* to ∞ (Alg. 13, line 4); then a visitor is queued for the source vertex with $length = 0$ (Alg. 13, line 8).

When a visitor *pre_visits* a vertex, it checks if the visitor's length is smaller than the vertex's current length (Alg. 12, line 14). If smaller, the *pre_visit* updates the level information and returns *true*, signaling that the main visit function may proceed. Then, the main *visit* function will send new *bfs_visitors* for each outgoing edge (Alg. 12, line 18). The *less than comparison* procedure orders the visitors in the queue by *length* (Alg. 12, line 26).

The *delegate behavior* is configured to *pre_visit_parent* (Alg. 12, line 4), which means that visitors of delegated vertices traverse up the *delegate tree* before reaching the *controller*. Forcing visitors to traverse up the delegate tree provides the opportunity to filter out visitors that are not part of the shortest path.

When a visitor successfully updates the controller's state, the controller broadcasts the visitor to all of its delegates (Alg. 12, line 20). If the visitor does not update the controller's state, then the visitor is terminated (Alg. 12, line 22).

Algorithm 12 BFS & SSSP Visitor

```
1: visitor state: vertex ← vertex to be visited
2: visitor state: length ← path length
3: visitor state: parent ← path parent

4: delegate behavior: pre_visit_parent

5: procedure PRE_VISIT(vertex_data)
6:     if length < vertex_data.length then
7:         vertex_data.length ← length
8:         vertex_data.parent ← parent
9:         return true
10:    end if
11:    return false
12: end procedure

13: procedure VISIT(graph, visitor_queue)
14:    if length == graph[vertex].length then
15:        for all vi ∈ out_edges(g, vertex) do
                                          ▷ Creates and queues new visitors
16:            new_len ← length + edge_weight(g, vertex, vi)
                                          ▷ edge_weight equals 1 for BFS
17:            new_vis ← bfs_visitor(vi, new_len, vertex)
18:            visitor_queue.push(new_vis)
19:        end for
20:        return bcast_delegates
21:    else
22:        return terminate_visit
23:    end if
24: end procedure

25: procedure OPERATOR < ()(visitor_a, visitor_b)
                                          ▷ Less than comparison, sorts by length
26:    return visitor_a.length < visitor_b.length
27: end procedure
```

Algorithm 13 BFS & SSSP Traversal Initiator

```
1: input: graph ← input graph G(V, E)
2: input: source ← BFS traversal source vertex
3: input: vis_queue ← Visitor queue

4: for all v ∈ vertices(graph) parallel do
5: |   graph[v].length ← ∞
6: |   graph[v].parent ← ∞
7: end for
8: source_visitor ← bfs_visitor(source, 0, source)
9: vis_queue.push(source_visitor)
10: vis_queue.do_traversal()
```

5.3.2 PageRank

The visitor used to asynchronously compute the PageRank for each vertex is shown in Algorithms 14 and 15. For our experiments, we are concerned with the performance of a single PageRank iteration. Many iterations may be required for convergence, depending on the topology of the graph. Before the asynchronous PageRank begins, a temporary *sum* is initialized to 0 for all vertices, and a visitor containing the initial PageRank value is queued for every vertex (Alg. 15, line 9).

When a visitor *pre_visits* a vertex, it simply increments the PageRank sum for the vertex (Alg. 14, line 4). The *delegate behavior* is set to *post_merge* which requires a visitor merge function, that also simply returns a sum (Alg. 14, line 16). When every vertex is initially visited with the initial PageRank value, new visitors are queued for every outgoing edge (Alg. 14, line 12). When a controller is visited, it broadcasts the visitor to all its delegates (Alg. 14, line 14).

When the traversal completes, and the delegates have merged their visitors, the final PageRank value has been calculated for every vertex (Alg. 15, line 15).

Algorithm 14 Page-Rank Visitor (*pr_visitor*)

```
1:  visitor state: vertex ← vertex to be visited
2:  visitor state: rank ← partial Page-Rank value

3:  delegate behavior: post_merge

4:  procedure PRE_VISIT(vertex_data)
5:      vertex_data.sum += rank
6:      return false
7:  end procedure

8:  procedure VISIT(graph, visitor_queue)
9:      for all vi ∈ out_edges(g, vertex) do
                                              ▷ Creates and queues new visitors
10:         edge_rank ← rank / out_degree(g, vertex)
11:         new_vis ← pr_visitor(vi, edge_rank)
12:         visitor_queue.push(new_vis)
13:     end for
14:     return bcast_delegates
15: end procedure

16: procedure MERGE(visitor_a, visitor_b)
17:     visitor_a.rank += visitor_b.rank
18:     return visitor_a
19: end procedure
                                          ▷ No visitor ordering required
```

Algorithm 15 Page-Rank Initiator (single iteration)

```
1: input: graph ← input graph G(V, E)
2: input: vis_queue ← Visitor queue
3: input: damp ← Page-Rank damping factor (e.g., 0.85)
4: input: init_rank ← initial rank for every vertex
5: output: out_rank → output rank for every vertex

6: for all v ∈ vertices(graph) parallel do
7: |   graph[v].sum = 0
8: |   vis ← pr_visitor(v, init_rank[v])
9: |   vis_queue.push(vis)
10: end for

11: vis_queue.do_traversal()
                              ▷ Traversal complete, delegate visitors' merged
12: for all v ∈ vertices(graph) parallel do
13: |   vertex_sum = graph[v].sum
14: |   V = num_vertices(graph)
15: |   out_rank[v] = (1 − damp)/V + (damp ∗ vertex_sum)
16: end for
```

5.3.3 K-Core Decomposition

To compute the k-core decomposition of an undirected graph, we asynchronously remove vertices from the core whose degree is less than k. As vertices are removed, they may create a dynamic cascade of recursive removals as the core is decomposed.

The visitor used to compute the k-core decomposition of an undirected graph is shown in Algorithm 16. Before the traversal begins, each vertex initializes its *k-core* to *degree(v) + 1* and *alive* to *true*, then a visitor is queued for each vertex with *ntrim* set to 1.

The visitor's *pre_visit* procedure decrements the vertex's *k-core* number by *ntrim*, and checks if it is less than k (Alg. 16, line 8). If less, it sets *alive* to false and returns *true*, signaling that the visitors's main *visit* procedure should be executed (Alg. 16,

Algorithm 16 K-Core Visitor

```
1: visitor state: vertex ← vertex to be visited
2: visitor state: ntrim ← count of edges trimmed
3: static parameter: k ← k-core requested

4: delegate behavior: lazy_parent_merge

5: procedure PRE_VISIT(vertex_data)
6:     if vertex_data.alive == true then
7:         vertex_data.kcore ← vertex_data.kcore − ntrim
8:         if vertex_data.kcore < k then
9:             vertex_data.alive ← false
10:            return true
11:        end if
12:    end if
13:    return false
14: end procedure

15: procedure VISIT(graph, visitor_queue)
16:    for all vi ∈ out_edges(g, vertex) do
17:        new_visitor ← kcore_visitor(vi, 1)
18:        visitor_queue.push(new_visitor)
19:    end for
20:    return bcast_delegates
21: end procedure

22: procedure MERGE(visitor_a, visitor_b)
23:    visitor_a.ntrim += visitor_b.ntrim
24:    return visitor_a
25: end procedure
                                          ▷ No visitor order required
```

line 10). The *visit* function notifies all neighbors of *vertex* that it has been removed from the k-core (Alg. 16, line 18). After the traversal completes, all vertices whose *alive* equals *true* are a member of the k-core.

The *delegate behavior* is configured to *lazy_merge_parent* (Alg. 16, line 4), which means that visitors of delegated vertices are lazy merged up the *delegate tree* before reaching the *controller*. Visitors are merged using the procedure shown in Alg. 16,

line 22. Merging visitors before visiting the controller reduces the number of times the controller is required to execute the *pre_visit* procedure.

5.4 Asymptotic Analysis

We build on the analysis framework discussed in Section 4.4 to analyze the complexity of algorithms using distributed delegates. When high degree vertices are delegated, their storage, computation, and communication are parallelized and distributed. The algorithmic effects are:

- High-degree storage reduces from $O(d_{max})$ to $O(\frac{d_{max}}{p})$. The storage of high-degree vertices is now evenly stored across the partitions. This enables all partitions to participate in the computation and communication of high-degree vertices.

- High-degree computation reduces from $O(d_{max})$ to $O(\frac{d_{max}}{p})$. The computation for high-degree vertices is now evenly distributed across the partitions.

- High-degree communication performed through the delegate tree reduces from $O(d_{max})$ to $O(p)$ communication and $O(h_{tree})$ steps. The communication of high-degree vertices is performed using tree based broadcasts and reductions.

A comparison of the BFS and PageRank using 1D, Edge List Partitioning (ELP), and Distributed Delegates is shown in Table 5.4, and analysis parameters are described in Table 5.5. For BFS, the *pre_visit_parent* delegate behavior replaces the cost of ELP's ghosts ($O(p)$) with delegate tree operations ($O(h_{tree})$). For PageRank, the *post_merge* delegate behavior adds reductions for high-degree vertices, reducing ELP's $O(d^{in}_{max})$ with delegate tree operations ($O(h_{tree})$).

Alg.	1D	ELP	Delegates
BFS	$O(D + p_{max} + d^{in}_{max} + d^{out}_{max})$	$O(D + \frac{\|E\|}{p} + p)$	$O(D + \frac{\|E\|}{p} + h_{tree})$
PageR.	$O(p_{max} + d^{in}_{max} + d^{out}_{max})$	$O(\frac{\|E\|}{p} + d^{in}_{max})$	$O(\frac{\|E\|}{p} + h_{tree})$

Table 5.4: Comparison of 1D, Edge List Partitioning (ELP) and Distributed Delegates

Variable	Description
D	The Graph's diameter
d^{out}_{max}	Maximum out-degree, $\max_{v \in V}(\textit{out-degree}(v))$
d^{in}_{max}	Maximum in-degree, $\max_{v \in V}(\textit{in-degree}(v))$
p	Number of processors
p_{max}	Maximum partition size
h_{tree}	Height of delegate tree

Table 5.5: Analysis Parameters

5.5 Experiments

In this section we experimentally evaluate the performance and scalability of our approach. We use the IBM BG/P Intrepid supercomputer at Argonne National Laboratory [26] up to 131K processors to show scalability to large core count. We also use Cab [41] at Lawrence Livermore National Laboratory, which is a standard HPC Linux cluster with an Infiniband interconnect. We begin by exploring the effects of varying the *delegate degree threshold*. Next, we show a weak scaling study for Breadth-First Search, Single Source Shortest Path, K-Core Decomposition and PageRank, followed by comparisons to our previous *edge list partitioning* [60] and 1D partitioning. Finally, we compare performance to the best known Graph500 performance for Intrepid which uses a 2D partitioning approach [19].

For this experimental study, the only optimization specific to IBM BG/P is

matching the routed communication topology to the 3D torus as discussed in Section 5.2.4. We use the Graph500 performance metric of Traversed Edges per Second (TEPS) for both BFS, SSSP and PageRank. Similar to TEPS, we used the rate of trimmed edges per second as the performance metric for K-Core Decomposition.

5.5.1 Effects of Delegate Degree Threshold

The delegate degree threshold (d_{high}) is the threshold at which vertices are selected to be delegated. Vertices whose degree is less than d_{high} are 1D partitioned, while those above the threshold are delegate partitioned.

We explore the scaling effects of d_{high} on overall performance, number of co-located edges, and partition imbalance, shown in Figure 5.3. For a fixed graph size of 2^{30} vertices, using 4096 cores, we demonstrate the performance effects of (a) BFS and (b) PageRank as d_{high} is scaled. The best performing degree threshold for both BFS and PageRank is 4096 (equal to the number of cores). Decreasing d_{high} results in a higher percentage of co-located edges (Fig. 5.3(c)). However, when the threshold decreases below 4096, the broadcasts to all partitions become wasteful as many delegates will have zero edges on some partitions. At large values of d_{high}, the partitioning reduces to a 1D partitioning with fewer vertices selected to become delegates. In addition to reducing overall performance, the partition imbalance increases when few delegates are created (Fig. 5.3(d)). The percentage of vertices selected to be delegated is small for all values of d_{high} (Fig. 5.3(e)); this means that the additional overhead of managing delegate information is also small.

The optimal d_{high} is roughly equal to the number of cores (p), so for the remainder of our delegate experiments we set d_{high} equal to p. This means that d_{high} increases during our weak-scaling studies.

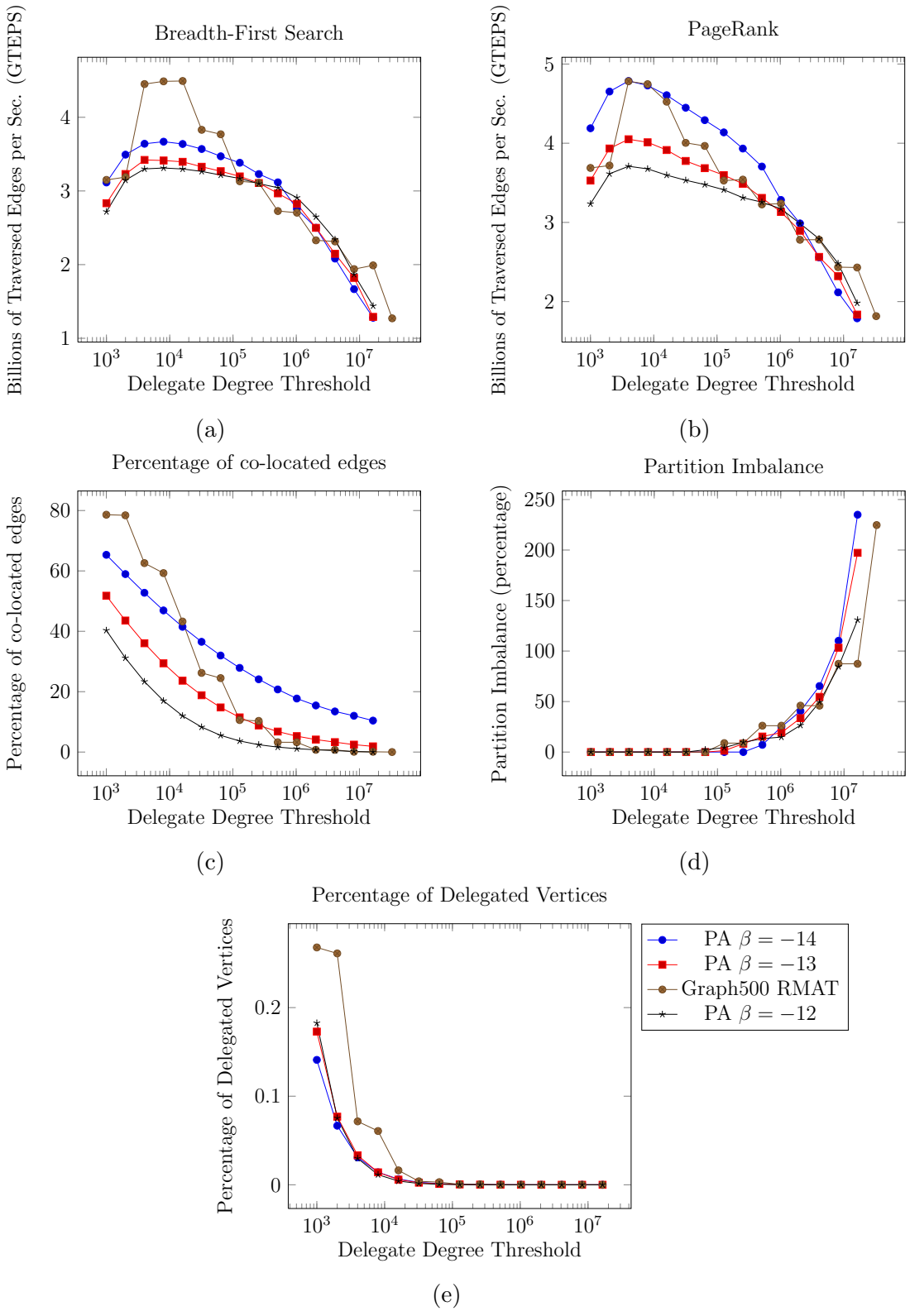


Figure 5.3: Effects of delegate degree threshold (d_{high}) using 4096 cores on graphs with 2^{30} vertices. The performance effects of (a) BFS and (b) PageRank, (c) the effects on the percentage of co-located edges, (d) partition imbalance, and (e) the percentage of delegated vertices are shown.

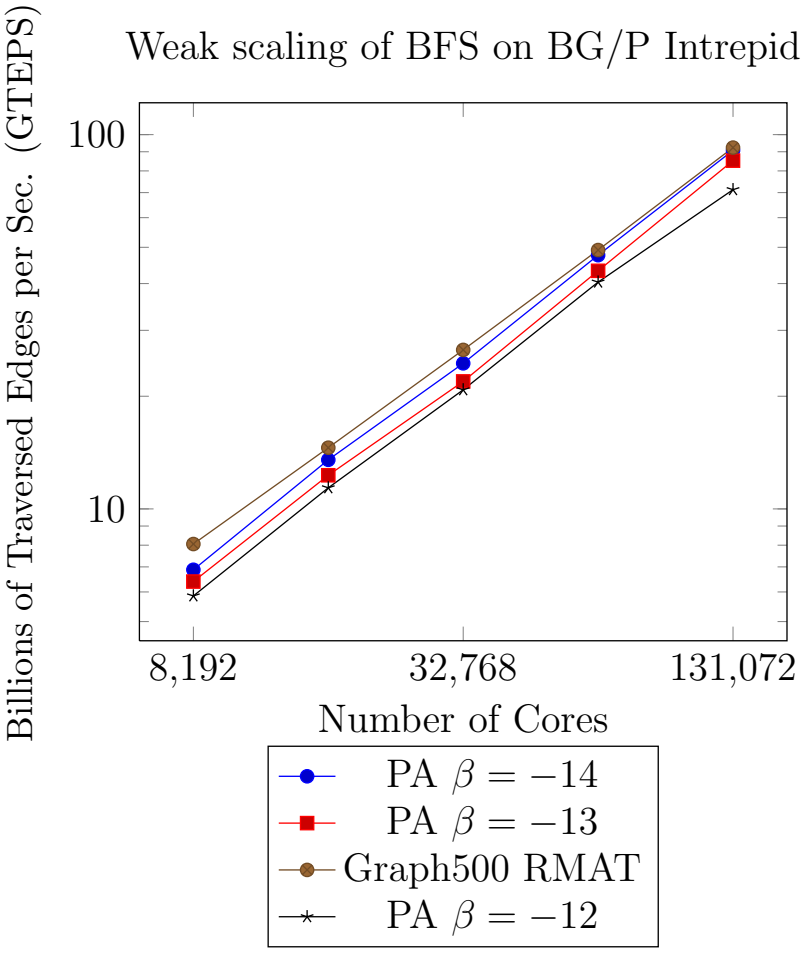


Figure 5.4: Weak scaling of BFS on BG/P Intrepid. There are 2^{18} vertices per core, with the largest scale graph having 2^{35}.

5.5.2 Weak Scaling of BFS and PageRank

The weak scaled performance using distributed delegates on BG/P Intrepid is shown in Figures 5.4 and 5.5 for BFS and PageRank, respectively. The approach demonstrates excellent weak-scaling up to 131k cores with 2^{35} vertices. There are 2^{18} vertices per core, with the largest scale graph having 2^{35}.

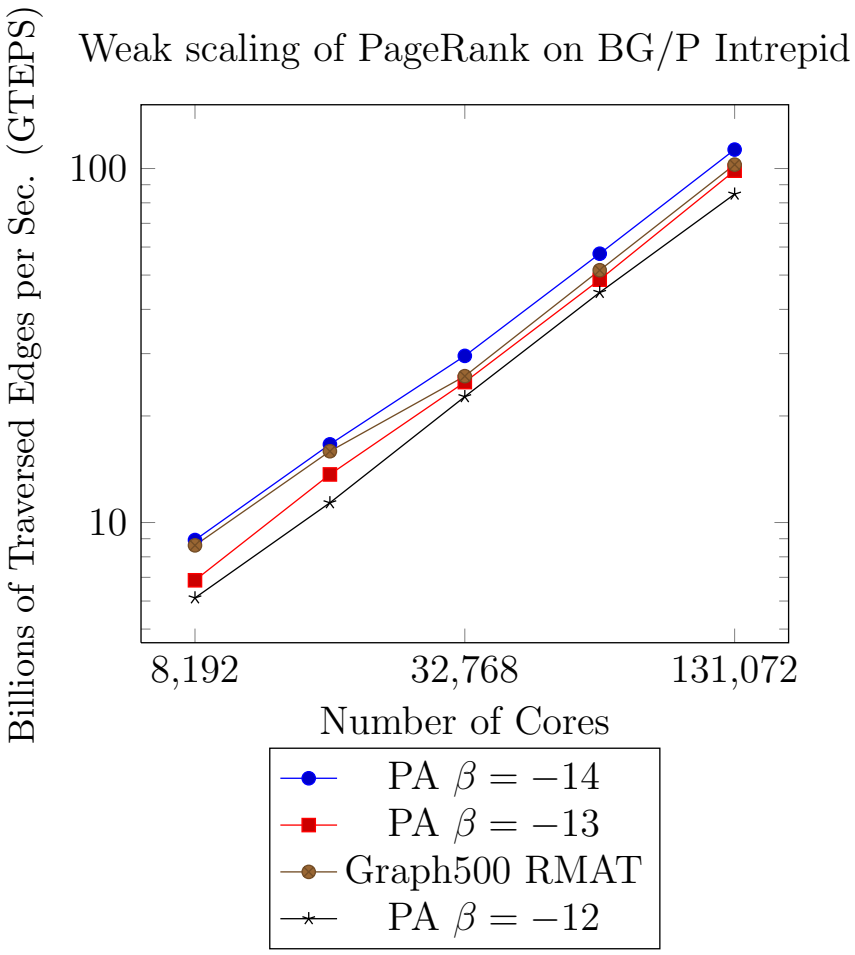


Figure 5.5: Weak scaling of PageRank on BG/P Intrepid. There are 2^{18} vertices per core, with the largest scale graph having 2^{35}.

5.5.3 Weak Scaling of SSSP and K-Core Decomposition

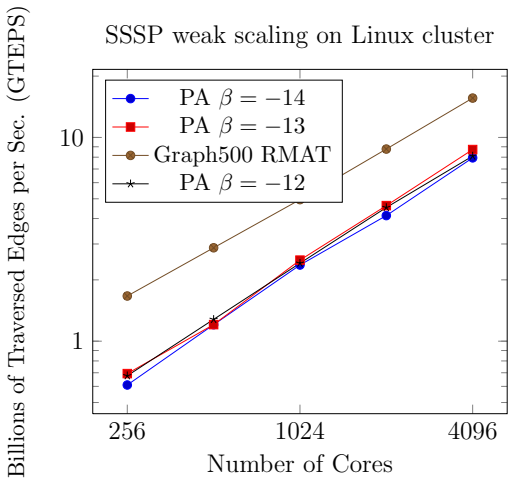
The weak scaled performance using distributed delegates on Cab at LLNL is shown in Figure 5.6 for SSSP and K-Core decomposition. In addition to good scaling, this demonstrates the portability of our approach to a broader class of HPC resources. For SSSP, edges are randomly weighted with integers ranging $[1, 2^{30})$.

5.5.4 Comparison to 1D and Edge Partitioning

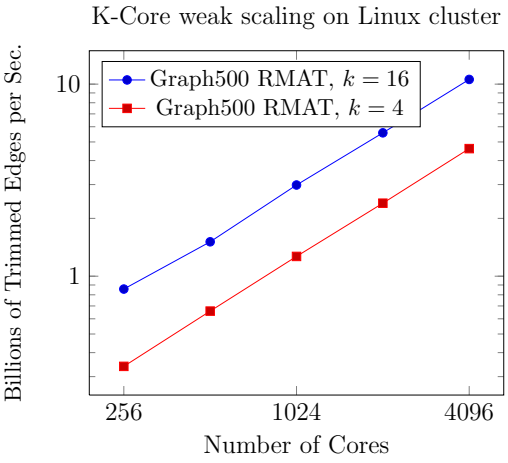
We compare distributed delegate partitioning to our previous work on edge-list partitioning [60] and 1D partitioning in Figure 5.7. 1D partitioning is widely used by many graph libraries such as PBGL [29], and is used in these experiments as a baseline. For this experiment, the number of vertices per core have been reduced to prevent 1D partitioning from exhausting local partition memory due to imbalance. Also, the experiments are limited to 4096 cores due to increasing hub growth causing additional imbalance. At 4096 cores, our delegate partitioning is 42% faster than edge-list partitioning and 2.3x faster than 1D. PBGL was not able to run with more than 512 processors without exhausting available memory. At 512 cores, our delegate partitioning is 5.6x faster than PBGL.

5.5.5 Comparison to Previous Graph500 Gesults

We compare distributed delegates to the best known performance for Intrepid [19] on the Graph500 list in Figure 5.8. Our approach demonstrates excellent weak scaling, and achieves 93.1 GTEPS on a Scale 35 Graph500 input using 131k cores. The delegates approach outperforms the current best known Graph500 performance for Intrepid by 15%.



(a)



(b)

Figure 5.6: Weak scaling of delegate partitioned (a) SSSP and (b) K-Core on Cab Linux cluster at LLNL. There are 2^{20} vertices per core, with the largest scale graph having 2^{32} vertices. For SSSP, edges are randomly weighted with integers ranging $[1, 2^{30})$.

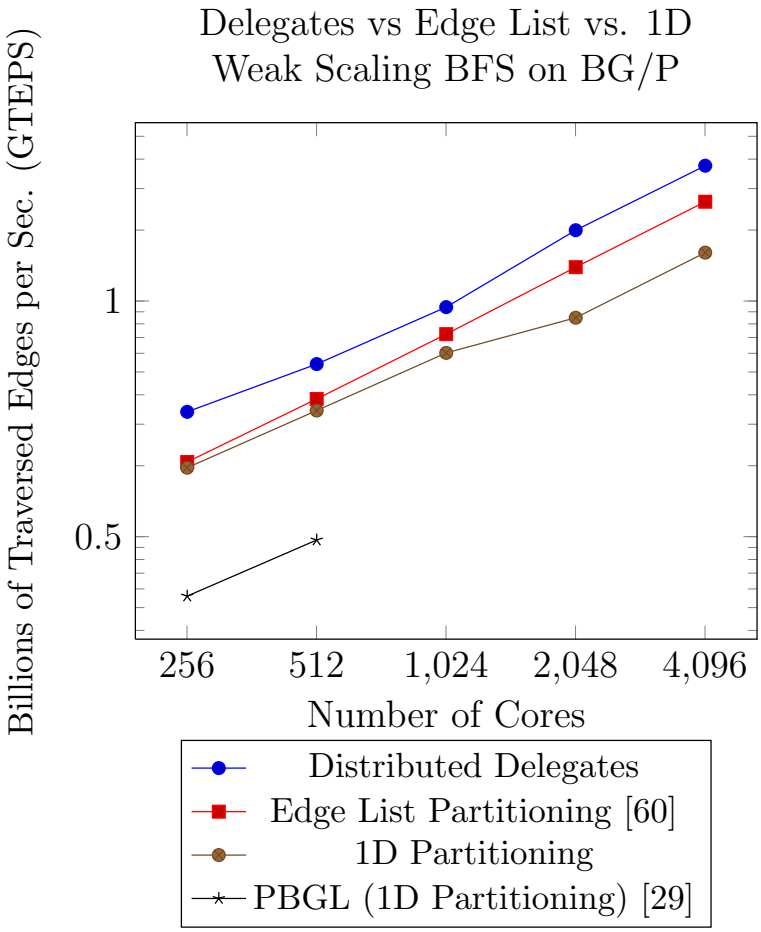


Figure 5.7: Comparison of *distributed delegates* vs. edge list partitioning [60], 1D partitioning, and PBGL [29]. Performance of BFS on RMAT graphs shown on BG/P. Important note: the graph sizes are reduced to prevent 1D from running out of memory. There are 2^{17} vertices and 2^{21} undirected edges per core.

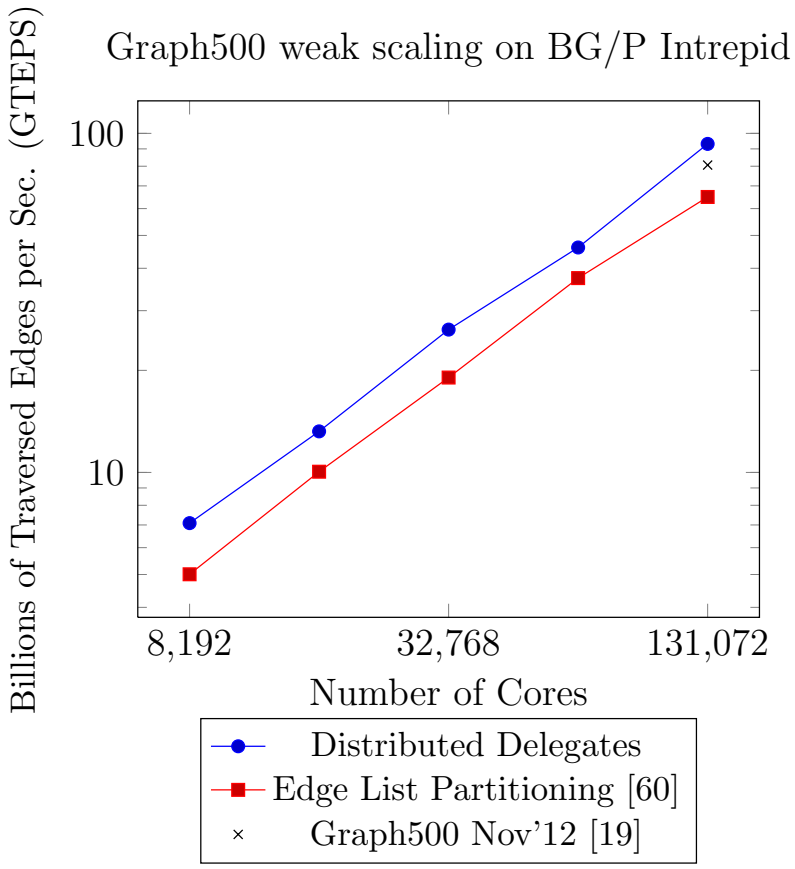


Figure 5.8: Weak scaling of delegate partitioned BFS on BG/P Intrepid. Compared to Intrepid BFS performance from the Graph500 list. Delegate Partitioning is 15% faster than best results published for Intrepid on the Graph500 list. There are 2^{18} vertices per core, with the largest scale graph having 2^{35} vertices.

5.6 Summary

In this chapter, we present a novel technique, distributed delegates, to parallelize the storage, processing, and communication of high-degree vertices in large scale-free graphs. To balance the processing workload, we distribute hub data structures and related computation among a set of delegates. Computation is coordinated between the delegates and their *controller* through a set of commands and behaviors.

Our delegate technique leads to significant communication reduction through the use of asynchronous broadcast and reduction operations. For hubs whose degree is greater than the number of processing cores, p, using delegates reduces the required volume of communication.

We demonstrate the approach and evaluate performance and scalability using Breadth-First Search (BFS), Single Source Shortest Path (SSSP), K-Core Decomposition, and PageRank on synthetically generated scale-free graphs. We demonstrate scalability up to 131K cores using the IBM BG/P supercomputer, and show portability on a typical HPC Linux cluster. Our algorithm improves the best known Graph500 results for BG/P Intrepid, a custom BG/P implementation, by 15%.

6. CONCLUSION

Efficiently storing and processing large amounts of graph data is a challenging problem in data intensive computing as researchers seek to leverage "Big Data" to answer next-generation scientific questions. This dissertation presents new techniques to parallelize the storage, computation, and communication of high-degree vertices in scale-free graphs. Our work facilitates the processing of large real-world graph datasets through the development of parallel algorithms and tools that scale to large computational and memory resources, overcoming challenges not addressed by existing techniques. Towards this goal, we begin by identifying key challenges to storing and processing massive scale-free graphs. Many important graph datasets have unstructured and irregular topologies with data locality which thrashes multi-level memory hierarchies, including external memory. These irregular topologies produce dense processor-processor, approaching all-to-all, communication when algorithms are parallelized, leading to poor overall performance. Also, the growth of high-degree vertices, also known as *hubs*, provides significant challenges for balancing storage, computation, and communication.

We address these challenges with three novel techniques for processing large scale-free graphs. First, we developed an asynchronous graph traversal technique capable of expressing fine-grained parallelism at the individual vertex level [59]. Data latencies associated with the external graph storage media and message passing communication are mitigated by the asynchronism of the computation.

Second, we created a new partitioning technique that guarantees balanced partitions containing challenging high-degree vertices [60]. Previous partitioning strategies using 1D and 2D partitioning may produce an imbalanced number of edges per

partition for scale-free graphs. Our approach partitions the graphs edges such that each partition contains an equal number of edges, overcoming the storage balance issues created by high-degree vertices.

Finally, we developed a technique to parallelize and distribute the storage, computation, and communication of high-degree vertices [58]. We make a distinction between low and high degree vertices, and distribute the high-degree vertices. The number of edges per partition is balanced, and the large amount of computation and communication for the high-degree vertices is distributed over all of the processors.

Our techniques provide new tools to analyze large scale-free graph datasets on a wide rage of data-intensive computational resources. Our research is targeted at leadership class supercomputers containing significant distributed memory resources, clusters with node-local non-volatile random access memory (NVRAM), and small shared-memory systems containing large NVRAM storage devices.

The research contributions of this dissertation can be summarized as:

- We developed novel algorithmic techniques to process large scale-free graphs:
  - An asynchronous computation model using prioritized visitor queues that tolerates latencies associated with external memory and distributed message passing [59];
  - An edge list partitioning technique that guarantees balanced partitions for scale-free graphs containing high-degree vertices [60];
  - A technique we call distributed delegates to parallelize and distribute the storage, computation, and communication of high-degree vertices [58];
- We demonstrated our techniques using: Breadth-First Search, Single Source Shortest Path, Connected Components, K-Core decomposition, Triangle Count-

ing, and PageRank;

- We demonstrate the scalability of our approach on leadership class supercomputers using 131k processors;
- We show that by leveraging node-local NAND Flash, our approach can process larger datasets with only modest performance degradation over a DRAM-only solution.

In the future, we plan to extend this work to new classes of graph algorithms that require distributed set operations. Set operations can be used to represent communities or clusters of vertices. High-degree vertices also create challenges when maintaining the set relationships, and we plan to extend our delegate approach to address the challenges.

REFERENCES

[1] Deepak Ajwani, Andreas Beckmann, Riko Jacob, Ulrich Meyer, and Gabriel Moruz. On computational models for flash memory devices. In *Experimental Algorithms*, pages 16–27, 2009.

[2] Deepak Ajwani, Roman Dementiev, and Ulrich Meyer. A computational study of external-memory BFS algorithms. In *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 601–610, New York, NY, USA, 2006. ACM.

[3] Deepak Ajwani, Itay Malinger, Ulrich Meyer, and Sivan Toledo. Characterizing the performance of flash memory storage devices and its impact on algorithm design. In *Proceedings of the 7th International Workshop on Experimental Algorithms (WEA)*, pages 208–219, 2008.

[4] Deepak Ajwani and Ulrich Meyer. Design and engineering of external memory traversal algorithms for general graphs. In *Algorithmics of Large and Complex Networks: Design, Analysis, and Simulation*, pages 1–33, Berlin, Heidelberg, 2009. Springer-Verlag.

[5] D.A. Bader and K. Madduri. SNAP, Small-world Network Analysis and Partitioning: an open-source parallel graph framework for the exploration of large-scale networks. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–12, 2008.

[6] A.L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[7] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[8] B.W. Barrett, J.W. Berry, R.C. Murphy, and K.B. Wheeler. Implementing a portable multi-threaded graph library: The MTGL on Qthreads. In *Proceedings of the 2009 IEEE International Symposium on Parallel &Distributed Processing (IPDPS)*, pages 1–8. IEEE, 2009.

[9] Jonathan W. Berry, Bruce Hendrickson, Simon Kahan, and Petr Konecny. Software and algorithms for graph queries on multithreaded architectures. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–14, March 2007.

[10] D. P. Bertsekas, F. Guerriero, and R. Musmanno. Parallel asynchronous label-correcting methods for shortest paths. *J. Optim. Theory Appl.*, 88(2):297–320, 1996.

[11] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice & Experience*, 34(8):711–726, 2004.

[12] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[13] Andrei Broder, Ravi Kumar, Farzin Maghoul, Prabhakar Raghavan, Sridhar Rajagopalan, Raymie Stata, Andrew Tomkins, and Janet Wiener. Graph structure in the web. *Computer Networks*, 33(1–6):309 – 320, 2000.

[14] Thang Nguyen Bui and Curt Jones. Finding good approximate vertex and edge partitions is np-hard. *Inf. Process. Lett.*, 42(3):153–159, May 1992.

[15] A. Buluç and J.R. Gilbert. On the representation and multiplication of hypersparse matrices. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–11. IEEE, 2008.

[16] A. Buluç and K. Madduri. Parallel breadth-first search on distributed memory systems. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–12. IEEE, 2011.

[17] J. Callan, M. Hoy, C. Yoo, and L. Zhao. Clueweb09 data set. `http://www.lemurproject.org/clueweb09/`, 2009.

[18] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *Fourth SIAM International Conference on Data Mining*, April 2004.

[19] Fabio Checconi, Fabrizio Petrini, Jeremiah Willcock, Andrew Lumsdaine, Anamitra Roy Choudhury, and Yogish Sabharwal. Breaking the speed and scalability barriers for graph exploration on distributed-memory machines. In *Supercomputing*, 2012.

[20] Graph 500 Steering Committee. The graph500 benchmark. `http://www.graph500.org`, 2010.

[21] T.H. Cormen. *Introduction to algorithms*. The MIT press, 2001.

[22] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[23] N.B. Ellison, C. Steinfield, and C. Lampe. The benefits of facebook "friends:" social capital and college students' use of online social network sites. *Journal of Computer-Mediated Communication*, 12(4):1143–1168, 2007.

[24] Paul Erdös and A Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hungar. Acad. Sci*, 5:17–61, 1960.

[25] Brian Van Essen, Roger Pearce, Sasha Ames, and Maya Gokhale. On the role of NVRAM in data-intensive architectures: an evaluation. In *International Parallel and Distributed Processing Symposium (IPDPS)*, Shanghai, China, May 2012.

[26] Argonne Leadership Computing Facility. IBM BG/P Intrepid. `http://www.alcf.anl.gov/intrepid`, 2013.

[27] Michelle Girvan and Mark EJ Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12):7821–7826, 2002.

[28] Andrew V. Goldberg. A simple shortest path algorithm with linear average time. In *ESA '01: Proceedings of the 9th Annual European Symposium on Algorithms*, pages 230–241, London, UK, 2001. Springer-Verlag.

[29] Douglas Gregor and Andrew Lumsdaine. The parallel BGL: A generic library for distributed graph computations. In *In Parallel Object-Oriented Scientific Computing (POOSC)*, 2005.

[30] Chris Groër, Blair D Sullivan, and Steve Poole. A mathematical analysis of the r-mat random graph generator. *Networks*, 58(3):159–170, 2011.

[31] F. Guerriero and R. Musmanno. Parallel asynchronous algorithms for the k shortest paths problem. *Journal of Optimization Theory and Applications*, 104(1):91–108, 2000.

[32] Harshvardhan, Adam Fidel, Nancy M. Amato, and Lawrence Rauchwerger. The STAPL Parallel Graph Library. In *Int. Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2012.

[33] Bruce Hendrickson and Jonathan W. Berry. Graph analysis with high-performance computing. *Computing in Science and Engineering*, 10(2):14–19, 2008.

[34] Bruce Hendrickson and Robert Leland. A multilevel algorithm for partitioning graphs. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 28. ACM, 1995.

[35] Joseph JáJá. *An introduction to parallel algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992.

[36] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[37] George Karypis and Vipin Kumar. Parallel multilevel series k-way partitioning scheme for irregular graphs. *Siam Review*, 41(2):278–300, 1999.

[38] T. Kolda, D. Brown, J. Corones, T. Critchlow, T. Eliassi-Rad, L. Getoor, B. Hendrickson, V. Kumar, D. Lambert, C. Matarazzo, K. McCurley, M. Merrill, N. Samatova, D. Speck, R. Srikant, J. Thomas, M. Wertheimer, and P. C. Wong. Data sciences technology for homeland security information management and knowledge discovery: Report of the dhs workshop on data sciences. Technical Report UCRL-TR-208926, Jointly released by Sandia National Laboratories and Lawrence Livermore National Laboratory, September 2004.

[39] Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew Tomkins. Trawling the web for emerging cyber-communities. *Computer Networks*, 31(11–16):1481 – 1493, 1999.

[40] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 591–600, New York, NY, USA, 2010. ACM.

[41] Lawrence Livermore National Laboratory. Cab at LLNL. `http://computing.llnl.gov/resources`, 2013.

[42] Dominique LaSalle and George Karypis. Multi-threaded graph partitioning. In *International Parallel and Distributed Processing Symposium (IPDPS)*, May 2013.

[43] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[44] Jure Leskovec, Kevin J Lang, and Michael Mahoney. Empirical comparison of algorithms for network community detection. In *Proceedings of the 19th international conference on World wide web*, pages 631–640. ACM, 2010.

[45] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan W. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(1):5–20, 2007.

[46] Benjamin Machta and Jonathan Machta. Parallel dynamics and computational complexity of network growth models. *Phys. Rev. E*, 71:026704, Feb 2005.

[47] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[48] Friedemann Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2:161–175, 1987. 10.1007/BF01782776.

[49] David W. Matula and Leland L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. *J. ACM*, 30(3):417–427, July 1983.

[50] Kurt Mehlhorn and Ulrich Meyer. External-memory breadth-first search with sublinear i/o. In *ESA '02: Proceedings of the 10th Annual European Symposium on Algorithms*, pages 723–735, London, UK, 2002. Springer-Verlag.

[51] U. Meyer and P. Sanders. δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114–152, 2003.

[52] Burkhard Monien and Stefan Schamberger. Graph partitioning with the party library: Helpful-sets in practice. In *Computer Architecture and High Performance Computing, 2004. SBAC-PAD 2004. 16th Symposium on*, pages 198–205. IEEE, 2004.

[53] A. Moody, G. Bronevetsky, K. Mohror, and B.R. De Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–11. IEEE, 2010.

[54] Tamás F Móri. On random trees. *Studia Scientiarum Mathematicarum Hungarica*, 39(1):143–155, 2002.

[55] Kameshwar Munagala and Abhiram Ranade. I/O-complexity of graph algorithms. In *SODA '99: Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, pages 687–694, Philadelphia, PA, USA, 1999. Society for Industrial and Applied Mathematics.

[56] Mark EJ Newman. Modularity and community structure in networks. *Proceedings of the National Academy of Sciences*, 103(23):8577–8582, 2006.

[57] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: bringing order to the web. 1999.

[58] Roger Pearce, Maya Gokhale, and Nancy M. Amato. Faster parallel traversal of scale free graphs at extreme scale with vertex delegates. In *(Under Review)*.

[59] Roger Pearce, Maya Gokhale, and Nancy M. Amato. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *2010 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1 –11, Nov. 2010.

[60] Roger Pearce, Maya Gokhale, and Nancy M. Amato. Scaling techniques for massive scale-free graphs in distributed (external) memory. In *International Parallel and Distributed Processing Symposium (IPDPS)*, May 2013.

[61] François Pellegrini. Scotch and libscotch 5.1 user's guide. 2008.

[62] François Pellegrini and Jean Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *High-Performance Computing and Networking*, pages 493–498. Springer, 1996.

[63] J. Raacke and J. Bonds-Raacke. Myspace and facebook: Applying the uses and gratifications theory to exploring friend-networking sites. *CyberPsychology & Behavior*, 11(2):169–174, 2008.

[64] Matthew Richardson, Rakesh Agrawal, and Pedro Domingos. Trust management for the semantic web. In *The Semantic Web-ISWC 2003*, pages 351–368. Springer, 2003.

[65] Peter Sanders and Christian Schulz. Engineering multilevel graph partitioning algorithms. In *Algorithms–ESA 2011*, pages 469–480. Springer, 2011.

[66] Peter Sanders and Christian Schulz. Distributed evolutionary graph partitioning. In *ALENEX*, pages 16–29, 2012.

[67] Stephen B. Seidman. Network structure and minimum degree. *Social Networks*, 5(3):269 – 287, 1983.

[68] C Seshadhri, Ali Pinar, and Tamara G Kolda. An in-depth study of stochastic kronecker graphs. In *Data Mining (ICDM), 2011 IEEE 11th International Conference on*, pages 587–596. IEEE, 2011.

[69] Comandur Seshadhri, Ali Pinar, and Tamara G. Kolda. Triadic measures on graphs: The power of wedge sampling. In *SIAM International Conference on Data Mining*, 2013.

[70] Yossi Shiloach and Uzi Vishkin. An o(logn) parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57 – 67, 1982.

[71] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: user guide and reference manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[72] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[73] B. Viswanath, A. Mislove, M. Cha, and K.P. Gummadi. On the evolution of user interaction in facebook. In *Proceedings of the 2nd ACM workshop on Online social networks*, pages 37–42. ACM, 2009.

[74] Jeffrey Scott Vitter. Algorithms and data structures for external memory. *Found. Trends Theor. Comput. Sci.*, 2(4):305–474, 2008.

[75] Jeffrey Scott Vitter and Elizabeth A.M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2-3):110–147, 1994.

[76] Chris Walshaw and Mark Cross. Mesh partitioning: a multilevel balancing and refinement algorithm. *SIAM Journal on Scientific Computing*, 22(1):63–80, 2000.

[77] Chris Walshaw and Mark Cross. Parallel optimisation algorithms for multilevel mesh partitioning. *Parallel Computing*, 26(12):1635–1660, 2000.

[78] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440–442, Jun 1998.

[79] K.B. Wheeler, R.C. Murphy, and D. Thain. Qthreads: An API for programming with millions of lightweight threads. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8, 2008.

[80] J.J. Willcock, T. Hoefler, N. Edmonds, and A. Lumsdaine. Active pebbles: parallel programming for data-driven applications. In *Proceedings of the International Conference on Supercomputing*, pages 235–244. ACM, 2011.

[81] Anthony Williams and Vicente J. Botet Escriba. BOOST Threads. `http://www.boost.org/doc/libs/release/libs/thread/`, 2013.

[82] A. Yoo, A.H. Baker, R. Pearce, and V.E. Henson. A scalable eigensolver for large scale-free graphs using 2d graph partitioning. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–11. IEEE, 2011.

[83] Andy Yoo, Edmond Chow, Keith Henderson, William McLendon, Bruce Hendrickson, and Umit Catalyurek. A scalable distributed parallel breadth-first search algorithm on BlueGene/L. In *Supercomputing, 2005. Proceedings of the*

ACM/IEEE SC 2005 Conference, page 25, Washington, DC, USA, 2005. IEEE Computer Society.

[84] W.W. Zachary. An information flow model for conflict and fission in small groups. *Journal of Anthropological Research*, pages 452–473, 1977.