ALGORITHM-LEVEL OPTIMIZATIONS FOR SCALABLE PARALLEL GRAPH PROCESSING

A Dissertation

by

HARSHVARDHAN

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

May 2018

Major Subject: Computer Science

ABSTRACT

Efficiently processing large graphs is challenging, since parallel graph algorithms suffer from poor scalability and performance due to many factors, including heavy communication and load-imbalance. Furthermore, it is difficult to express graph algorithms, as users need to understand and effectively utilize the underlying execution of the algorithm on the distributed system. The performance of graph algorithms depends not only on the characteristics of the system (such as latency, available RAM, etc.), but also on the characteristics of the input graph (small-world scale-free, mesh, long-diameter, etc.), and characteristics of the algorithm (sparse computation vs. dense communication). The best execution strategy, therefore, often heavily depends on the combination of input graph, system and algorithm.

Fine-grained expression exposes maximum parallelism in the algorithm and allows the user to concentrate on a single vertex, making it easier to express parallel graph algorithms. However, this often loses information about the machine, making it difficult to extract performance and scalability from fine-grained algorithms.

To address these issues, we present a model for expressing parallel graph algorithms using a fine-grained expression. Our model decouples the algorithm-writer from the underlying details of the system, graph, and execution and tuning of the algorithm. We also present various graph paradigms that optimize the execution of graph algorithms for various types of input graphs and systems. We show our model is general enough to allow graph algorithms to use the various graph paradigms for the best/fastest execution, and demonstrate good performance and scalability for various different graphs, algorithms, and systems to 100,000+ cores.

DEDICATION

To my Mother and Father, who inspired by example, encouraged and supported me, and inculcated in me dedication, strong ethics and morals.

ACKNOWLEDGMENTS

I would like to thank my advisers, Prof. Nancy Amato and Prof. Lawrence Rauchwerger for their continued mentoring, support and guidance throughout my undergraduate and graduate career. I would like to thank my committee members Prof. Andrew Jiang and Prof. Marvin Adams for their mentorship and guidance.

Special thanks to Adam Fidel, Dr. Gabriel Tanase, Antal Buss, Dr. Ioannis Papadopoulos, Glen Hordemann, Brandon West and Mani Zandifar for not only being excellent colleagues and lab-mates, but also close friends throughout my time in graduate school. Finally, I would like to thank Dr. Timmie Smith and Dr. Nathan Thomas, and the members of the Parasol Lab for all their help and collaboration.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a dissertation committee consisting of Professors Nancy M. Amato [advisor], Lawrence Rauchwerger [advisor] and Anxiao Jiang of the Department of Computer Science and Engineering, and Professor Marvin L. Adams of the Department of Nuclear Engineering.

Portions of this research were previously published. The basic framework was presented as The STAPL Parallel Graph Library in the proceedings of the 25th International Workshop on Languages and Compilers for Parallel Computing (LCPC) [1]. The k-level-async paradigm was published in the proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques (PACT) [2], where it won the Best Paper Award. The techniques for communication reduction in graph algorithms were published as An Algorithmic Approach to Communication Reduction in Parallel Graph Algorithms in the proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques (PACT) [3], where it was a Best Paper Award Finalist. The techniques for processing big data graphs were published as A Hybrid Approach to Processing Big Data Graphs on Memory-Restricted Systems in the proceedings of the 29th IEEE International Parallel and Distributed Processing Symposium (IPDPS) [4]. All work for the dissertation was completed in collaboration with the co-authors of the aforementioned works.

Funding Sources

This research is supported in part by NSF awards CCF 0702765, CNS-0551685, CCF-0833199, CCF-1439145, CCF1423111, CCF-0830753, IIS-0917266, by DOE awards DEAC02-06CH11357, DE-NA0002376, B575363, by Samsung, IBM, Intel, and by Award KUS-C1-016-04, made by King Abdullah University of Science and Technology (KAUST). This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Dept. of Energy under Contract No. DE-AC02-05CH11231.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Processing large graphs is essential in a variety of domains, from social network and web-scale graphs to scientific meshes and nuclear reactor-design [5]. Processing these graphs in a reasonable time usually requires parallelism. Using a distributed data-structure allows massive graphs to be processed quickly and concurrently. However, parallelizing graph algorithms efficiently is a challenging problem that has received significant attention for several decades [6]. Despite many attempts over the past decade [7, 8, 9] to allow programmers to easily express their graph computations in parallel, graph algorithms remain notoriously hard to scalably parallelize. Existing graph libraries are restrictive in allowing users to express algorithms and may require them to manage details of data-distribution and communication. Developers also often correlate an increase in the level of abstraction with a corresponding decrease in performance. Vice-versa, most highly-optimized implementations tend to introduce system/parallelism details in the algorithm, thereby decreasing the level of abstraction. Furthermore, the performance of many parallel graph algorithms tend to heavily depend not only on the system characteristics, but also on the characteristics of the input graph. For example, a breadth-first search on a binary-tree has significantly different performance characteristics from the same algorithm on a social-network.

In particular, graph algorithms suffer from poor scalability and performance at scale due to poor work balance (both, among processors and across supersteps of execution of the algorithm), heavy communication, and irregular access. These bottlenecks manifest due to the combination of the graph, algorithm and the system. For example, given an input graph, the PageRank algorithm exhibits heavier communication than a breadth-first traversal. Conversely, though, PageRank, where all supersteps process every vertex, displays a more balanced computational pattern across the supersteps of the algorithm than breadth-first traversal, where some supersteps may process the majority of the graph, while other levels only process small percentage of vertices.

Similarly, changing input graphs from uniform or semi-uniform mesh networks used in scientific computations and simulations, to small-world scale-free graphs representing social and web networks, has a dramatic impact on the execution of the algorithm. For the same algorithm, a scale-free network will produce dense communication to almost every processor due to the presence of *hub-vertices* that have very high out-degrees. The small-world property will result in the majority of vertices being processed in a few supersteps. The challenge in such graphs is to improve the effective bandwidth of the system, as well as improve the load-balance in order efficiently process the hub vertices and their connections. A mesh network, on the other hand, may typically have a large diameter and low variance in the vertex degree. This combination results in a very large number of supersteps, each with only a small fraction of the total work. The challenge here is to improve latency and reduce the cost of global synchronizations, in order to address the large number of supersteps. Existing frameworks such as Google's Pregel [9] or the Parallel Boost Graph Library (PBGL) [7] can only effectively solve certain limited subsets of these combinations, and suffer from poor performance in other cases. Other libraries, such as Galois [10], Ligra [11] and MTGL [8] are limited to shared-memory, and are therefore limited in the scale of the problem they can solve. There doesn't exist a framework that can address all these challenges at large-scale.

Another instance where these bottlenecks become prohibitive is out-of-core graph processing. Reading from and writing to disks are high-latency operations. Due to the irregular nature of graph access-patterns, conventional techniques of paging and prefetching do little to alleviate the latency. The best available out-of-core graph processing systems still require $O(N^2)$ random reads and writes [12], and consequently are unable to perform well at scale.

1.1 Contributions

This research designs a generic parallel graph library (STAPL GL) that allows users to concentrate on graph algorithm development by abstracting them from the input data and the details of the underlying distributed environment, while providing portable, scalable performance. STAPL GL allows for the separation of the algorithm from the container, as well as decouples the algorithm from the input graph and machine through the use of traversal strategies. It provides a generic, customizable graph container, a collection of common parallel graph algorithms, as well as multiple graph traversal strategies (including level-synchronous, asynchronous and hierarchical). Further, STAPL GL automates load balancing and locality-related optimizations.

Specific contributions of this research include:

- **Programmability and Abstraction.** Provide a high-level graph abstraction to allow programmers to completely decouple the design of the graph algorithm from the details of parallelism, input graph, implementation of the graph container and the machine. Users express their computation in its most natural form, while the underlying data-structure implementation and traversal strategy can be chosen to offer maximum performance based on the input graph and platform.
- **Multiple Graph Traversal Strategies.** Provide multiple traversal strategies for expressing graph algorithms to allow the algorithms to achieve the best performance for different classes of input graphs and machines. In particular, this research will focus on the following traversal strategies:
  - **KLA** (k-level Asynchronous). This novel traversal strategy unifies asynchronous and level-synchronous traversal strategies, while allowing parametric control of asynchrony. We also provide a model to estimate the appropriate level of asynchrony for a given input graph, a strategy to adaptively determine k, and show how to transform common asynchronous and level-synchronous graph algorithms to KLA.
  - **Hierarchical Traversal Strategy.** This traversal strategy allows users to express computations hierarchically. It also allows expression of certain algorithms (e.g. graph-partitioning and Boruvka's minimum spanning-tree) in their most natural form, as well as allows for reduction of communication in certain cases by taking advantage of the machine hierarchy.
  - **Out-of-Core Strategy.** This strategy uses sub-graph paging together with the asynchronous-push model to eliminate non-sequential reads and minimize writes, making it suitable for out-of-core processing. A theoretical model shows the I/O to be linear in the num-

ber of reads and writes, improving on the existing $O(n^2)$ strategies. This translates to a $10\times$ improvement in real-world performance.

- **Scalable Performance.** Provide improved scalability in performance and data size over tens of thousands of cores compared with existing graph libraries on standard benchmarks. Moreover, provide light-weight support for load balancing through asynchronous data migration, and demonstrate improved performance and scalability in a real-world production application by mitigating load-imbalance through automatic redistribution of vertices.

An implementation of the STAPL Parallel Graph Library infrastructure[1] provides a parallel graph container (pGRAPH), associated graph pVIEWS, common graph algorithms, as well as the k-level-async, hierarchical, and out-of-core processing strategies. Results show improved performance over existing graph libraries and frameworks, and scalability to 100,000+ cores. Experiments with k-level-async show substantial performance improvements over existing level-synchronous and asynchronous traversal strategies, while the hierarchical and out-of-core strategies show an order-of-magnitude improvement in performance over flat graph processing and other out-of-core graph processing strategies respectively.

1.2 Overview of Our Approach

We provide a graph-processing engine (the *graph_paradigm*) to control the execution of algorithms, that can effectively utilize available resources for fast, scalable processing of graphs. To do this, we allow users to express their algorithms in a vertex-centric fine-grained manner that abstracts them from details of parallelism and exposes the maximum amount of parallelism available in the algorithm. The graph algorithms themselves do not change and are decoupled from execution policies (distributed/shared, disk/in-memory, etc.), allowing the execution strategies to change, without changing the algorithm.

We further provide multiple strategies of algorithm execution for tolerating high-latencies and low bandwidths in multiple situations. The k-Level asynchronous paradigm allows latency-hiding in high-latency systems by trading- off costly global barriers for point-to-point local synchronizations and redundant work. The hierarchical paradigm increases effective bandwidth by minimizing the amount of communication, and the out-of-core processing paradigm uses sub-graph paging with the asynchronous push-model to eliminate non-sequential reads and minimize writes.

Much of our research was previously published. The overall framework design and implementation was published at the *International Workshop on Languages and Compilers in Parallel Computing* (LCPC) 2012 [1]. The k-level-asynchronous paradigm for unifying existing BSP and asynchronous paradigms was published at the *International Conference on Parallel Architectures and Compilation Techniques* (PACT) 2014 [2], and won the Best Paper Award. The work on out-of-core graph processing was published at the *IEEE International Parallel & Distributed Processing Symposium* (IPDPS) 2015 [4]. Finally, the work on hierarchical communication reduction in graph algorithms was published at the *International Conference on Parallel Architectures and Compilation Techniques* (PACT) 2015 [3], and was nominated for the Best Paper Award.

1.3 Outline

This dissertation is organized as follows. Section 2 begins with an overview of the STAPL GL model, followed by a brief discussion of existing graph processing models and systems, and tradeoffs involved with each. The section then describes the asynchronous push model and how various graph algorithms may be expressed in this model.

Section 3 describes the two existing approaches to processing graphs – Bulk Synchronous Parallel (BSP) and Asynchronous – and discusses the tradeoffs involved in using them. It then introduces a novel paradigm for graph processing – the k-Level Asynchronous paradigm (KLA), that unifies the two existing approaches, along with requirements, guarantees and a proof of correctness. It further describes two approaches for finding the optimal level of asynchrony for a given combination of input graph, algorithm, and system.

Section 4 presents an approach to reducing the amount of communication done in a graph processing system, and shows how it can improve the performance of algorithms on small-world scale-free networks. It then analyses bounds on the amount of communication reduction and the space overhead required. Finally, it concludes with a discussion of related work.

Section 5 introduces the problem of out-of-core graph processing, including the challenges involved in processing large graphs from disk. It then presents a novel approach for scalably and efficiently processing out-of-core graphs on a distributed machine, along with several optimizations to reduce the disk I/O. The approach elliminates non-sequential disk reads and minimizes random disk writes. The section next analyss the I/O costs associated with processing graphs in this paradigm, and concludes with a brief discussion of the related work.

Section 6 provides an overview of real-world applications from different domains built using STAPL GL. Details of an implementation of STAPL GL in STAPL, along with the various graph processing paradigms are provided in Section 7, and results of experimental evaluation of this implementation is provided in Section 8. Section 8 also provides an overview of the experimental setup and input graphs used, along with comparisons with existing graph processing systems. It then provides detailed evaluation of each of the different graph processing paradigms at large scale for various input graphs and algorithms on different systems.

2. THE STAPL GL MODEL

One of the challenges to processing graphs is the difficulty in expressing parallel graph algorithms naturally and effectively. To overcome this, STAPL GLuses different *models* to decouple the algorithm's expression from its execution, abstracting away the parallelism and communication details. Graph algorithms in STAPL-GL are expressed using either a fine-grained vertex-centric manner, or a coarse-grained hierarchical manner.

The fine-grained "vertex-centric" approach, popularized by the Pregel graph processing framework [9], is suitable for expressing a large class of graph algorithms such as breadth-first search (and other traversals), PageRank, connected components, k-core decomposition, centrality metrics, and coloring. Several existing graph processing frameworks use a vertex-centric model to express algorithms. Algorithm writers specify operations to be performed on vertices and their neighbors, which are iteratively applied to the vertices of the graph by the execution engine to execute the algorithm.

However, there is a class of naturally hierarchical graph algorithms such as Boruvka's minimum spanning trees, graph partitioning, and agglomerative clustering, for which the vertex-centric approach is not effective. Due to their inherent hierarchical nature and the need to operate on coarse chunks of the graph instead of individual vertices, the fine-grained approach makes it hard for algorithm developers to express such algorithms. On the other hand, coarse graph algorithms are harder to express on a parallel system and limit the amount of parallelism exposed – a fine-grained algorithm exposes the maximum amount of parallelism as each unit of work can be independently processed in parallel. To address this, STAPL GL also provides a hierarchical model that eliminates the need for the algorithm writer to think about the details of creating the hierarchy itself, providing instead well-specified operators that define the hierarchy, executing the algorithm as the hierarchy is built.

This chapter starts with a discussion of relevant related work in this area, along with existing approaches of expressing graph algorithms and their tradeoffs. Section 2.2 describes the fine-grained graph paradigm in STAPL GL, along with example algorithms. Finally, Section 2.3 describes the hierarchical (coarse-grained) paradigm in STAPL GL, and how algorithms are expressed in it.

2.1 Related Work

2.1.1 Existing Models

Pull vs. Push Model. Graph algorithms typically exchange data between the vertices of the graph. This may be done by "pushing" the current state of a vertex to a subset of its neighbors, or "pulling" (or reading from) the current state of its neighboring vertices. Systems such as Pregel [9] and SGL rely on the push model, while GraphLab [13] and GraphChi [12] rely on the pull model of communication. While both methods produce the same end result, the push model can be implemented asynchronously, by allowing overlapped communication and computation – a vertex can send its state to its neighbors and then perform local computation. On the other hand, the pull model needs to be synchronous. Furthermore, since in most graph algorithms vertices do not communicate with all their neighbors (and only a small subset of total vertices are actively communicating at all in each iteration), it may not be possible for a vertex to know which subset of its neighbors (if any) to pull data from, and therefore, systems implementing the pull model must read the states of all neighboring vertices at every step. The push model only needs to communicate data that is required, allowing the algorithm to perform much lower communication than in the pull model. Push models are also more naturally suited for distributed-memory computation than pull models. Due to these reasons, we chose to use a push model for our system. We show comparisons in scalability and speed between implementations of the two models in our experiments section.

Vertex vs. Edge Centric Models. Another way to classify fine-grained graph models is whether they emphasize computations upon vertices or edges. Vertex-centric models incorporate the "think like a vertex" [9] philosophy, where the algorithm is expressed in terms of operations on vertices and their associated states (including any associated outgoing-edge states), and the edges are used to establish communication between vertices. Operations read and update vertex states, perform computations on these vertices, and send the result of the computations to neighboring vertices.

On the other hand, edge-centric algorithms are expressed in terms of operations on edges, with associated source- and target-vertex states. Edge-centric operators read states from both, source and target vertices of an edge, and perform computations to update their states. For certain algorithm, an edge-centric approach may provide a more natural expression than a vertex-centric model. However, since multiple edges may be incident upon the same vertex, a coherence mechanism is required to keep the vertex's state consistent while potentially multiple parallel updates read from and write to the same vertex. This, in practice limits the scalability of edge-centric models, making them harder to implement efficiently on distributed systems.

Coarse-Grained Models. There has also been work in expressing parallel graph algorithms in a coarse-grained manner. Such models are subgraph-centric, where the graph is partitioned into subgraphs and some computation is performed on them, followed by an exchange of boundary data between the subgraphs. Certain algorithms such as Boruvka's MST, graph partitioning and clustering are naturally hierarchical, and are therefore also better expressed in such models. However, non-hierarchical algorithms such as traversals, PageRank, and several others, are better expressed in fine-grained models which do not require a separate specification of the boudary condition.

2.1.2 Other Graph-Processing Systems

The Parallel Boost Graph Library (PBGL) [7] is a stand-alone distributed-memory graph library, with similar goals as STAPL GL. An important difference from STAPL GL is that since PBGL does not provide a shared-object view, it exposes users to explicit knowledge of parallelism and data distribution details through the use of process groups. This makes it more difficult to program. As an example, PBGL's interface requires the user to know explicitly the location of a vertex before any operations may be performed on it, and therefore, there is no locality-agnostic way to access remote vertices. Another difference is that PBGL's algorithms are inherently level-synchronous.

The Multi-Threaded Graph Library (MTGL) [8] is a shared-memory graph library designed to effectively utilize the unique architectural features of Cray XMP massively multithreaded machines. It can be ported to other platforms using the QThreads library. However, the model exposes programmers to the QThreads API, as well as the details of multi-threaded programming.

Google's Pregel [9] is a library for processing graphs in parallel that emphasizes vertex-centric computation and algorithm design, making it easy to express and versatile. Pregel uses the Bulk Synchronous Processing (BSP) computational model [14] for executing algorithms.

GraphLab/PowerGraph [13] is a distributed-memory graph framework that allows users to choose between synchronous and asynchronous execution at runtime, but it does not support parametric control of asynchrony. It uses a pull-model (gather/scatter approach) to express computations. However, this model impacts scalability on large parallel systems, and may incur extra communication. It also requires users to understand low-level details of parallelism such as memory consistency and concurrency, as they have to choose a consistency model for their application and understand its implications.

There are also edge-centric graph processing systems such as Ligra [11], which express algorithms on edges, rather than the more commonly used vertex-centric model. However, due to the need for maintaining coherence among vertices, edge-centric libraries are limited to shared-memory systems in practice, and there doesn't exist a practical distributed-memory implementation of this paradigm.

GoFFish is a subgraph-centric graph processing framework for large-scale graph analytics. This framework aims to address scalability bottlenecks in fine-grained models by exposing partition information of the input graph to the algorithm writer. While this allows for performance and locality-based optimizations such as reduced communication and reduced number of supersteps, it makes the algorithm harder to express, since the user has to understand and account for data-distribution and graph partitioning. STAPL GL's fine-grained model overcomes challenges of scalability by optimizing communication separately from the user's expression of the algorithm. This allows STAPL GL algorithms to be expressible and scalable.

2.2 The Fine-Grained Graph Processing Paradigm

Fine-grained vertex-centric algorithms in STAPL GL use an asynchronous push-update model, which exposes a large amount of parallelism, while allowing the most flexibility in choosing execution and communication strategies based on input and system. In this section, we show how an example algorithm, breadth-first traversal (BFS) (Figure 2.1) can be expressed in STAPL GL's fine-grained graph paradigm. Other algorithms such as connected components, k-core, PageRank, community detection, graph coloring, betweenness centrality, pseudo-diameter, etc. can also be expressed in a similar manner.

Breadth-first traversal (BFS) is an important graph algorithm due to its extensive usage in traversing graphs, and due to its indirect usage as a part of numerous other graph algorithms, such as betweenness centrality. A BFS of a graph marks each of its vertices with their distance from a given source vertex.

To express an algorithm, the user provides two operators – a *vertex-operator* (Figure 2.1(a)) which performs the computation of the algorithm locally on a single vertex, and a *neighbor-operator* (Figure 2.1(b)) that updates the neighbor-vertices of the source vertex with the results of the computation based on incoming values from active vertices or edges. The user can optionally also provide an *edge-operator* to perform local computation on adjacent edges of a vertex based on values passed-in from the source vertex. The edge-operator is mostly needed when the algorithm sends different data on every edge. This provides a complete specification of the algorithm, decoupled from details of execution and communication. Thus, algorithm writers can focus on the expression of the algorithm.

For BFS, the vertex-operator checks if a vertex is active (grey) and propagates its distance to its neighbors through the neighbor-operator. The neighbor-operator updates the distance of the neighbor if needed, and marks it as active (grey). Active neighbors are processed by vertex-operators in the next iteration.

The overall BFS algorithm results from invoking the *execute* method (Figure 2.1(c)) and providing it an *execution policy* and the operators presented earlier to obtain the BFS. The execute method (Figure 2.2) applies the provided operators on active vertices of the input graph and handles communication, termination-detection of the algorithm, current active vertices, and the execution strategy (BSP, asynchronous, KLA, out-of-core, or communication-reduction), as specified by the *execution policy*. The algorithm terminates when all vertex-operators return false. The *execution*

```
bool bfs_vertex_op(vertex v)
  if (v.color == GREY)          // Active if GREY
    v.color = BLACK;
    VisitAllNeighbors(bfs_neighbor_op(_1, v.dist+1), v);
    return true;                // vertex was Active
  else  return false;           // vertex was Inactive
```

(a) vertex-operator

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

(b) neighbor-operator

```
void BFS(ExecutionPolicy policy, Graph graph, vertex source)
  source.color = GREY;
  execute(policy, bfs_vertex_op(), bfs_neighbor_op(), graph);
```

(c) Algorithm-driver

Figure 2.1: The fine-grained BFS algorithm.

policy, in addition to specifying the exection strategy, also encapsulates additional information required for the specified strategy. This can include specifying the level of asynchrony (k) for the KLA strategy, or the machine-hierarchy (h) for the communication-reduction strategy, etc. The selection of the execution policy can be done via user-provided hints, by sampling the graph, or adaptively.

This paradigm allows the expression of several general graph algorithms easily, while allowing several execution strategies and optimizations for fast and scalable execution. New execution strategies can be added by registering them as an *execution policy* in STAPL GL. The STAPL GL implementation includes over 30 fundamental graph algorithms expressed using this paradigm.

2.2.1 PageRank and Other Algorithms

The PageRank algorithm [15, 16] is a representative random-walk algorithm used to rank webpages on the internet in order of relative importance. The PageRank computation proceeds in iterations, where each vertex calculates its rank in iteration i based on the ranks of its neighbors in iteration $i - 1$, and then sends its new rank to its neighbors for the next iteration. Termination happens upon convergence of ranks or upon reaching a predetermined threshold for iterations. The

```
void graph_paradigm(Graph graph, VertexOp wf, NeighborOp uf)
  bool active = true;

  while(active) {
    pre_compute(graph);

    // apply vertex-operator to each vertex, reduce to
    // find #active vertices. wf returns true (active),
    // or false (otherwise), spawns neighbor-operators.
    active =
      reduce(map(vertex_wf(wf, visitor(uf)), graph), logical_or());
    global_fence();

    post_compute(graph);
  }
```

Figure 2.2: Pseudocode for the graph paradigm (level-sync execution).

```
bool pagerank_vertex_op(Vertex v)
  if (v.iteration < 20)
    v.rank = 0.15/num_vertices + 0.85*v.sum_ranks;
    v.iteration++;
    v.sum_ranks = 0;
    int n = v.neighbors().size();
    VisitAllNeighbors(v, pr_neighbor_op(_1, v.rank/n));
    return true;              // vertex was Active
  else  return false;         // vertex was Inactive
```

(a) vertex-operator

```
bool pr_neighbor_op(Vertex u, double rank)
  u.sum_ranks += rank;
  return true;
```

(b) neighbor-operator

Figure 2.3: The PageRank algorithm.

STAPL GL implementation of PageRank's vertex and neighbor operators is shown in Figure 2.3.

Other algorithms, such as k-core decomposition (Figure 2.4) and connected components can be expressed in a similar two operator fashion. More details can be found in [2].

2.3 The Hierarchical Graph Processing Paradigm

There is a class of graph algorithms that are naturally hierarchical, such as Boruvka's minimum spanning trees, graph partitioning, and agglomerative clustering. Due to their inherent hierarchical nature and the need to operate on coarse chunks of the graph instead of individual vertices, the fine-grained model is unsuitable for expressing such algorithms.

```
bool kcore_vertex_op(Vertex v)
  if (v.degree_count < K && !v.deleted)  // Propagate if degree <K
    v.deleted = true;
    VisitAllNeighbors(v, kcore_neighbor_op(_1));
    return true;                // vertex was Active
  else  return false;           // vertex was Inactive
```

(a) vertex-operator

```
bool kcore_neighbor_op(Vertex u)
  if (!u.deleted)
    u.degree_count--;         // update vertex degree
    return true;              // vertex was updated
  else  return false;
```

(b) neighbor-operator

Figure 2.4: The k-core algorithm

On the other hand, coarse graph algorithms are harder to express on a parallel system and limit the amount of parallelism exposed – a fine-grained algorithm exposes the maximum amount of parallelism as each unit of work can be independently processed in parallel. To alleviate this, we propose a novel paradigm for expressing hierarchical graph algorithms, which allows users to write fine-grained operators to express the coarsening phase of the algorithm, while construction of the hierarchy is handled by the execution engine.

The paradigm decouples the algorithm expression from the execution, eliminating the need for the algorithm writer to think about the details of creating the hierarchy itself. The hierarchical paradigm execution engine applies the user-provided operators on the input graph to create vertex-matchings which are then subsequently coarsened to form another level in the hierarchy. This process is iteratively repeated until some precondition is achieved, or only unconnected super-vertices remain in the top-level of the hierarchy.

The process of creating the hierarchy executes the algorithm. For example, selecting the neighbor-vertex with the lowest id in each fine-grained operator will result in executing the connected-components algorithm. Similarly, selecting the highest-weighted neighbor that is not a self-edge for every vertex, and executing it hierarchically will result in the selected edges forming a minimum spanning tree.

STAPL GL providese a general paradigm to naturally express hierarchical graph algorithms. To the best of our knowledge, none of the existing parallel graph-processing frameworks provide a similar facility. Furthermore, the paradigm is general enough to be implemented in other frameworks, and exposes the maximum parallelism available to the problem.

2.3.1 Graph Hierarchies

This section presents the concept of hierarchies in graphs. In the next sections, we will use this concept to demonstrate how our hierarchical paradigm allows the creation of hierarchies, and how it ties to the expression and execution of hierarchical graph algorithms.

We first start with the concept of a supergraph. Given a graph $G = (V, E)$, where V and E represent the set of G's vertices and edges, a supergraph S consists of vertices $\mathcal{V}$ and edges $\mathcal{E}$, such that each vertex in the supergraph represents an exclusive partition of vertices in graph G (given by the non-injective and surjective function $\mathcal{F}$), and each edge in the supergraph represents the group of inter-partition edges of G induced by partition function $\mathcal{F}$.

In other words,

SuperGraph: $S = (\mathcal{V}, \mathcal{E})$

$\upsilon = \mathcal{F}(V), \forall \upsilon \in \mathcal{V}$ [$\mathcal{F} : V \rightarrow \mathcal{V}$ is non-injective and surjective]

$e \in \mathcal{E} = (\upsilon_1, \upsilon_2, w), \upsilon_1, \upsilon_2 \in \mathcal{V}, w \in \mathcal{W}(E)$ [w is the weight]

A graph-hierarchy is a sequence of graphs where each graph is a super-graph of the preceding graph:

$H(G_0) = \{G_0, G_1, G_2, ..., G_h\}$

where $G_i = SuperGraph(G_{i-1})$, $0 < i \leq h$

A graph-hierarchy allows natural expression of hierarchical algorithms, and the size reduction obtained by creating super-graphs representative of the original graph may allow for faster processing, and may reduce communication overhead.

```
hierarchy = create_hierarchy(match(), vertex_reducer(), edge_reducer(),
                             done(), pre_compute(), post_compute());
```

Figure 2.5: Specification of a hierarchical algorithm.

In SGL, we provide support for creating super-graphs based on an input graph and some user-provided grouping or partial grouping, and allow the automatic calculation of super-vertex/edge properties from a reduction of the input graph's vertex/edge properties. A hierarchy is obtained by repeating this process till a specified user-condition is met.

2.3.2 Hierarchical Operators

The paradigm consists of four operators (Figure 2.5) – a match operator to decide which neighbor a given vertex should match to, two reducer operators for merging vertex- and edge-properties to compute the super-vertex and super-edge properties, and a done-operator that indicates the completion of hierarchy creation. In addition, pre- and post-compute operators can be provided to aid in computing the matching (for example, computing nearest neighbor metrics for agglomerative clustering) or creating side-effects (for example, adding edges to an output graph for MST). The process of creation of the hierarchy executes the algorithm. We introduce and define the user-operators (through which the user expresses their algorithms), outline the process of creating the hierarchy, and analyze the costs of hierarchy creation.

The Match Operator: The match operator takes as input a vertex and an integer indicating the current level of hierarchy, and marks the vertex with the vertex it should be grouped with in the next level of the hierarchy. These matchings or groupings can be absolute or relative. Absolute groupings have all vertices in the grouping matched with a single "leader" vertex, whereas relative groupings may have chains of match vertices, and the final grouping will be computed by the create-hierarchy algorithm.

Property-Reducer Operators: The vertex- and edge-reduce operators take as input two vertex or edge properties and an integer indicating the current level of the hierarchy, and return a vertex (or edge) property that is the reduction of the two input properties. This output will be used to set the property of the super-vertex/super-edge in the next level of the hierarchy.

The Done Operator: The done-operator takes as input the graph at level-i of the hierarchy, along with an integer specifying the level of the hierarchy, and returns true if the next level of the hierarchy should be created, or false if the hierarchy is completed.

Pre- and Post-Compute Operators: Pre- and post-compute methods take as input the graph at level-i along with the level i, and can optionally perform auxiliary computations.

The operators are applied on a given level of the hierarchy to compute the next level, starting with the input graph as the base-level. This process is repeated until a single vertex remains, or the user-specified done operator is satisfied. The resulting series of graphs form the hierarchy.

2.3.3 Creating The Hierarchy

To create a supergraph, we construct a new graph based on the specifications given by the user-operators. The number of vertices in the Supergraph is given by the number of groupings generated by the application of the Match-Operator to the input graph. The property of each vertex in the supergraph is generated by applying the vertex-reduce operator to the properties of the vertices of the input graph. The Superedges are generated whenever there is an edge between two partitions of the input graph. Since the supergraph is non-multi-edged, only a single superedge is generated for all inter-partition edges. However, each inter-partition edge from the input graph is stored in the super-edge. Similar to the supervertex's property, The superedge's property is also generated by a reduction of the properties of its corresponding input edges. Finally, the supergraph produced is pushed onto a stack of graphs representing the hierarchy, and the process is repeated.

2.3.4 Analysis of Cost of Hierarchy Creation

This section analyzes the computational complexity of general paradigm to create the hierarchy for a given algorithm. We assume a graph $G = (V, E)$, with n vertices and m edges distributed randomly and equally over $O(p)$ processing elements. We also assume a partition function $\mathcal{F}(V)$ that matches the vertices of the input graph, producing a set of matched vertices which induce the set of supervertices $\mathcal{V}$. This process is repeated until some user-defined termination condition is

```
void hierarchical_paradigm(Graph input_graph, MatchOp match,
                           VertexReducer vr, EdgeReducer er,
                           Done done,
                           Operator pre-compute,
                           Operator post-compute)

  vector<Graph> hierarchy;
  Graph g = input_graph;
  hierarchy.push_back(g);   // base-level of the hierarchy.
  int level = 0;
  while (g.size() > 1 && !done(g, level)) {
    pre-compute(g, level);
    for_each(g.vertices(), match_operator(level));

    // Creates a graph of super-vertices and super-edges
    // based on specified matchings. The super-vertices and
    // super-edges contain ids of their children, and their
    // properties are reductions of their children's
    // properties based on the provided reduce operators.
    // <Describe this step in more detail>.
    g = create_level(g, vertex-reducer(level),
                     edge-reducer(level));

    hierarchy.push_back(g);
    ++level;
  }
```

Figure 2.6: Pseudocode for the hierarchical graph paradigm.

reached. Lastly, we assume, without loss of generality (since $\mathcal{F}$ is non-injective and surjective), that the size of each succesive set of supervertices $|\mathcal{V}_i|$ decreases by a factor of $1 + \epsilon$, where $0 < \epsilon \leq 1$. Due to this, the computation performed for the input graph (G_0, the base level of the hierarchy) dominates the creation of the hierarchy. Therefore, the hierarchy will contain $O(log(n))$ levels and the total amount of computation performed to create the hierarchy is:

$$T_{Algo-Compute} = O(\frac{n+m}{p}) \tag{2.1}$$

Similarly, the cost for communication can be given by:

$$T_{Algo-Comm} = O(\frac{m}{p}) + O(p \cdot log(n)) \tag{2.2}$$

For each level, the cost associated with $\mathcal{F}$ is the cost of invocation of the user-defined match-operator $\mathcal{M}$ on each vertex. Further, the cost of creation of the next level of the graph includes the computing of properties of each super-vertex and super-edge by application of the vertex-

and edge-reducer operators (vr and er respectively) on the vertices and edges of the input graph. We also account for the invocation of the Done-operator ($\mathcal{D}$) at each level to compute if some termination condition has been reached:

$$T_{Algo} = O(\frac{n \cdot (T_{\mathcal{M}} + T_{vr}) + m \cdot T_{er}}{p} + T_{\mathcal{D}}) + O(p \cdot log(n)) \tag{2.3}$$

Similarly, the memory consumed by our hierarchical paradigm is bounded by: $O(\frac{n+m}{p})$

2.3.5 Algorithms Using the Hierarchical Paradigm

This section describes how a variety of hierarchical graph algorithms can be expressed in the hierarchical paradigm using the operators described above. In all cases, choosing appropriate operators will lead to execution of the algorithm as the hierarchy is created. The paradigm takes care of executing the operators, and creating the hierarchical graph appropriately.

Connected components:

- **Match-operator:** Neighbor with lowest id.
- **Vertex-reducer:** min(x, y) // x,y are CC ids.
- **Edge-reducer:** null.
- **Done:** $graph\_level[top].num\_edges() == 0$ // no edges remain.

The connected components algorithm assigns a unique ID to all vertices that can be reached by each other. This can be naturally be expressed hierarchically by collapsing pairs of connected vertices with each other until no connected vertices remain.

In our paradigm, the hierarchical connected components algorithm computes the lowest-id neighbor for every vertex in each level of the hierarchy. These matches are then reduced to compute the groupings for the super-vertices of the next level of the hierarchy, and the process is repeated until the top-most level of the hierarchy has no edges connecting any two super-vertices (all super-vertices are in their own connected component).

Minimum spanning tree (Boruvka's algorithm):

- **Match-operator:** Neighbor with lowest edge-weight.
- **Vertex-reducer:** null.
- **Edge-reducer:** min(x, y) // x,y are edge-weights.
- **Done:** $graph\_level[top].num\_vertices() == 1$ // single vertex remains.

The hierarchical minimum spanning tree algorithm uses Boruvka's approach to compute the result. Boruvka's algorithm is a naturally hierarchical algorithm. However, it is difficult to express in existing graph processing frameworks hierarchically, so many implementations typically use additional auxiliary data-structures such as union-find to keep track of "collapsed vertices". This is simplified using our paradigm.

In our approach, at each level of the hierarchy, each vertex chooses the lowest-weighted edge that is not a self-edge. Self edges denote loops in the sub-graph represented by the super-vertex, and are therefore not chosen. Each super-edge is assigned the minimum of the weights of its constituent edges. This process is iteratively executed until a single vertex remains, assuming a fully connected graph, or multiple unconnected vertices remain (representing a minimum spanning forest) if the graph is unconnected.

Graph partitioning (coarsening phase):

- **Match-operator:** Neighbor with highest edge-weight.
- **Vertex-reducer:** plus(x, y) // sum vertex weights.
- **Edge-reducer:** plus(x, y) // x,y are edge-weights.
- **Done:** $graph\_level[top].num\_vertices() == p$ // desired #vertices remain.

Hierarchical graph partitioning consists of three phases: (i) coarsening, (ii) partitioning, and (iii) refining. The coarsening phase collapses vertices that are "matched". Matchings are computed based on highest edge-weight vertices pairing together. This can be represented using the hierarchical paradigm. The super-graph's vertex and edge-weights are computed by summing the vertex-weights and edge-weights of the graph in the level below. The process stops when the top-level graph has the desired number of vertices (equal to desired number of partitions). After creating the hierarchy, a traditional partitioning algorithm (such as METIS) can be run on the top-level graph and the results propagated through the levels of the hierarchy in the refining phase.

Agglomerative clustering (also generalizable to other clustering):

- **Pre-Compute:** Compute nearest neighbor.
- **Match-operator:** Nearest neighbor.
- **Vertex-reducer:** centroid(x,y) // x,y are vertex coordinates.
- **Edge-reducer:** null.
- **Done:** $graph\_level[top].num\_vertices() == 1$ // single vertex remains.

Agglomerative clustering is an important algorithm used in data mining for cluster analysis. In this algorithm, each vertex starts with its own cluster and pairs of clusters consisting of nearest neighbors are merged to form super-vertices in the next level of the hierarchy. The super-vertex property is computed based on the centroids of its child vertices. The process is repeated until a single vertex remains. Different metrics and linkage criteria can be chosen to provide differently shaped clusterings.

3. THE KLA GRAPH PARADIGM – PROCESSING GRAPHS WITH BOUNDED ASYNCHRONY[*]

k-level-asynchronous (k-level-async) is a graph-processing paradigm that unifies Bulk Synchronous Parallel (BSP) [14] and asynchronous paradigms, by allowing each BSP superstep to execute up to k levels of the algorithm asynchronously. k-level-async allows users to express fine-grained vertex-centric graph algorithms. The algorithms are decoupled from parallelism and communication details, as well as from the processing of the graph, whether it is level-synchronous or asynchronous, or stored on disk or in RAM.

3.1 Existing BSP and Asynchronous paradigms

Graph algorithms have traditionally followed a BSP approach, where computation proceeds in supersteps, with communication happening between two supersteps. Each superstep's communication is guaranteed to have completed before the start of the next superstep, guaranteeing the availability of all information required for computation in the next superstep. This guarantee is enforced using a global barrier, which can quickly become expensive on large systems. Further, the BSP model is most efficient when the amount of work done in each superstep is high and uniformly distributed across processing elements, so that the overhead of the global barrier can be amortized. However, most graph topologies have long-tails, and skewed distribution, and many graph algorithms process relatively few vertices in most iterations (Figure: active-vertex distribution across iterations for various algorithms). This makes BSP inefficient for supersteps where (i) relatively little work is being done by the superstep, or (ii) if the work is poorly distributed.

To alleviate this, an asynchronous computation model was proposed. In this model, the global barriers are replaced by point-to-point local synchronizations across the edges of the graph. However, this implies no guarantees in the ordering or availability of messages required to perform

[*] The KLA model and some of the experimental results are reprinted from *proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, "KLA: A new algorithmic paradigm for parallel graph computations," 2014, pp. 27–38, Harshvardhan, A. Fidel, N. M. Amato, and L. Rauchwerger, ©2014 IEEE with permission from authors and IEEE.

computation on a given vertex. Therefore, if the number of messages required to perform the computation on a vertex is not algorithmically known a priori, the computation must be performed optimistically, which may lead to some computation having to be redone when new, more complete information is available, effectively leading to wasted work. This model is inherently better suited to algorithms and graph topologies which would have taken multiple supersteps to perform in the BSP model, or where the computation per superstep would have been exceedingly small or heavily imbalanced (e.g. all active vertices on one processor, or if $O(active - vertices) << O(p)$, leading to no effective distribution strategy). This is often the case with highly regular graphs with small average out-degrees and long diameters.

3.1.1 Tradeoffs: Asynchronous vs. BSP

As described, both the asynchronous and BSP models have tradeoffs, and neither performs well in all scenarios. While the BSP model is well suited for computationally-intensive phases, it has high global-synchronization costs in the face of stragglers and high-latency systems. On the other hand, the asynchronous model works well for hiding latency, but suffers from a high probability of performing redundant work.

3.2 The KLA Model

To balance aspects and tradeoffs of both models, we proposed the k-Level Asynchronous (KLA) model, where computation proceeds in BSP-like supersteps (KLA-Supersteps), but each KLA-Superstep may perform up to k-levels of asynchronous computation. When $k = 1$, the KLA model degenerates into the BSP model, while when k is greater than the total number of levels (d) required to complete the algorithm, KLA equates to the asynchronous model. For $1 < k < d$, the algorithm proceeds in $\lceil \frac{d}{k} \rceil$ phases. This unifies the previous models, as well as exposes intermediate levels of asynchrony. For a given execution profile, this allows the most effective use of both previous models by fine-tuning the trade-off between the cost of synchronizations and the cost of performing redundant work.

Figure 3.1 provides an intuition of how the level-synchronous, asynchronous, and k-level-async versions of the BFS algorithm shown in Figure 2.1 differ. From the user's perspective, the algorithm (expressed as vertex and neighbor operators) does not change. What does change is the *visitation logic* (Figure 3.1). This determines conditions to allow the computation to proceed on the neighbor vertex being visited. While the visit logic has been simplified in the figure to make it readable, the BFS code closely corresponds to what a user would write.

```
Visit(NeighOp update, Vertex u,
      int k_curr)
  bool active = update(u)
  if (active)
    bfs_vertex_op(u);
```

(a) Async Visit
spawn tasks

```
Visit(NeighOp update, Vertex u,
      int k_curr)
  bool active = update(u)
  if (active && k_curr % k != 0)
    bfs_vertex_op(u);
```

(b) KLA Visit
spawn tasks to depth k

```
Visit(NeighOp update, Vertex u,
      int k_curr)
  update(u);
```

(c) Level-Sync Visit
don't spawn further tasks

Figure 3.1: Visitation logic for async, KLA and level-sync behavior of BFS.

3.3 Requirements, Guarantees and Proof of Correctness

In this section, we provide a proof of correctness of the k-level-asynchronous paradigm. For an algorithm to execute using k-level-async, its neighbor-operator should be idempotent, i.e., it cannot rely on the order in which the vertex is visited and must be resilient to multiple visits. This is the same as for an asynchronous algorithm. This is required as k-level-async provides no synchronization guarantees to the algorithms neighbor and vertex operators while in the middle of a KLA-SS. However, k-level-async does guarantee that at the end of each KLA-SS, the state of the execution of the algorithm will be equivalent to executing k BSP supersteps. This can be proved using simple induction. For our proof, we use the breadth-first search algorithm as an example, though the same technique works for other graph algorithms.

Lemma 3.3.1. *At the end of the i-th KLA BFS KLA-SS, all vertices at distance $\leq i \cdot k$ from the source have been visited and labeled with the correct distances, and no vertices of distance $> i \cdot k$ from the source have been visited.*

Proof. The proof is by induction on i, the KLA-SS number. When $i = 0$, only the source has been visited and it is trivially true. Next, we assume that at the end of KLA-SS $(i-1)$, all vertices at distance $\leq (i-1) \cdot k$ from source have been visited and correctly labeled with distances, and vertices at a distance $> (i-1) \cdot k$ are unvisited.

At the beginning of KLA-SS i, only vertices at distance $(i-1) \cdot k + 1$ from source are active. The vertex-operator in Figure 2.1(a) is applied to all these active vertices and visits their neighbors (Figure 2.1(b), line 3) if they are at a distance $\leq i \cdot k$ from the source (Figure 3.1(b), line 3). During the visit, the distance to source is updated only if it decreases; hence, once the correct minimum distance is computed it will not be further updated. Moreover, since the vertex-operator visits all neighbors of active vertices, all edges in the sub-graph induced by the vertices visited during the ith KLA-SS, i.e., at distance $(i-1) \cdot k < d \leq i \cdot k$ from source, will be processed, implying that at some point during the KLA-SS, the correct distance will be computed for all vertices visited in that KLA-SS. $\square$

Corrollary 3.3.2. *KLA BFS will perform $\lceil \frac{d}{k} \rceil$ KLA-SSs, where d is the number of levels in the graph.*

Corrollary 3.3.3. *If $k = 1$, then KLA BFS visits the vertices of the graph in a level-synchronous manner.*

3.4 Finding optimal-k

The performance of KLA algorithms is dependent on the choice of a good value for k. However, it is non-trivial to determine the optimal value of k given an arbitrary input graph, machine and algorithm. Therefore, we present a technique to quickly determine an effective approximate value for k_{opt}. We also present an adaptive strategy that picks an appropriate k based on the current local topology of the graph as the algorithm progresses. This strategy is applicable in the general case. The interpolative method to approximate the value of k_{opt} is useful when executing the algorithm multiple times on the input. Our experimental results show these two techniques work well in practice.

3.4.1 Adaptive KLA

To help users obtain better performance from a single (or a few) run(s) of a KLA algorithm, or when they do not have sufficient information about the input graph, we use a simple adaptive strategy (Adaptive KLA) that dynamically varies k to process the graph. The strategy dynamically chooses the best value for the next iteration based on information from current and previous iterations.

The algorithm starts with $k = 1$ (or some user-defined start value). At the end of each KLA-SS, the value of k is doubled for the next level under the following conditions: 1) high out-degree vertices have not been discovered, 2) the penalty for asynchrony (e.g., wasted work for BFS or buffering cost for PageRank) has not exceeded a threshold, and 3) the cost of processing each vertex in the current KLA-SS is equal to or less than the cost of processing each vertex for the previous KLA-SSs. If the thresholds for either out-degree, asynchrony penalty or per-vertex processing cost have been exceeded, the k value is halved. In addition, any high out-degree vertices found in a given KLA-SS are marked for processing in the next KLA-SS to reduce the penalty for processing them prematurely. Finally, if the asynchrony penalty or processing cost exceed a maximum threshold (we empirically found 20% to work well for the machines on which we tested), the value for k is capped, so the algorithm does not suffer from greater asynchronous penalties attempting higher values. The thresholds for asynchrony penalty and processing cost can be chosen depending on the machine, such that for a machine where the cost of communication is much higher than the cost of computation, the threshold can be increased, and vice-versa.

While this technique may not provide the best performance compared to knowing *a priori* the value of k_{opt} (see Sec. 3.4.2), it is inexpensive in practice. As can be observed from Figure 8.8, it still provides substantial performance benefits compared to the level-synchronous and asynchronous paradigms by converging to an improved k value, which may then be used for subsequent runs.

3.4.2 A Model for Approximating k_{opt}

In this section we describe a model for the KLA paradigm that enables us to obtain a good approximation of the value (k_{opt}) for k which results in the lowest execution time for the algorithm based on the input graph and machine. This approach may be used when performing multiple traversals on a graph to obtain better performance than Adaptive KLA. We use BFS on an undirected, connected graph as an illustration of this model, and show, later in this section, how it applies to other algorithms.

Theoretical Model. Assume a level-synchronous BFS requires d iterations over the graph, processing active vertices in each iteration. As there is an enforced ordering guarantee, there is no wasted work in this paradigm. Therefore, for a graph with n vertices on p processors, the work done in parallel can be V_{max}, the maximum number of vertices to be processed by a processor (ideally, $V_{max} = \frac{n}{p}$). Assuming it takes time α to process each vertex of the graph on average and T_{sync} is the time required by a global-synchronization of p processors in the system, the total time is given by:

$$T^{BFS}_{Level-Sync} = \alpha \cdot V_{max} + d \cdot T_{sync} \tag{3.1}$$

On the other hand, an asynchronous BFS traversal needs only one global-synchronization for termination detection, but in the process may end up doing redundant work due to loss of ordering guarantees for messages. Assuming a function $\Psi(G)$ gives the amount of redundant work done for graph G (a method for approximating this will be discussed in detail later), the total work per processor now becomes $V_{max} + \Psi^p(G)$, where $\Psi^p(G)$ is the estimated amount of wasted work on a processor. Therefore, the total time for the asynchronous paradigm is given by:

$$T^{BFS}_{Async} = \alpha \left(V_{max} + \Psi^p(G)\right) + T_{sync} \tag{3.2}$$

Extending this to the KLA paradigm, the traversal performs $\lceil \frac{d}{k} \rceil$ KLA supersteps. Assuming the wasted work for a given k is given by the penalty-function $\Psi(G, k)$, where $\Psi(G, 1) = 0$ and

$\Psi(G,d) = \Psi(G)$, the time now becomes:

$$T^{BFS}_{KLA}(k) = \alpha\,(V_{max} + \Psi^p(G,k)) + \lceil\frac{d}{k}\rceil \cdot T_{sync} \tag{3.3}$$

The product αV_{max} represents the time taken by the algorithm to complete all useful work in parallel, and can therefore be replaced by T_{Work}:

$$T^{BFS}_{KLA}(k) = T_{Work} + \alpha \cdot \Psi^p(G,k) + \lceil\frac{d}{k}\rceil \cdot T_{sync} \tag{3.4}$$

When $k = 1$, KLA behaves the same as the level-synchronous paradigm, and for $k = d$, it behaves like the asynchronous paradigm. In the following section, we provide some insight on what causes wasted work.

Modeling the Wasted Work, $\Psi(G,k)$. Wasted work is caused by latency, where a message taking a shorter path arrives after a message through a longer path. This can be caused by two major factors: machine dependent (i.e., network latency) and graph dependent (i.e., distribution of the graph, processing order of adjacent edges of a vertex). As the exact determination of these factors is non-deterministic, we provide a model to approximate the wasted work ($\Psi_{est}(G,k)$) for a given graph in this section.

We classify the edges of an input graph into two categories: E_T, which are the edges in the output BFS tree connecting parents to their children in the BFS, and $E_{NT} = E - E_T$, which are the other edges in the graph. There exist edges in E_{NT} then, that may cause wasted work, because, if these edges are traversed before the corresponding tree edges, the target vertex and its children will need to redo the traversal with shorter paths.

We can further sub-divide E_{NT} into E_{NT_C} and E_{NT_I}. E_{NT_C} are those E_{NT} edges that cross a KLA superstep, i.e., their source vertex is in a different KLA superstep than their target vertex. E_{NT_I} are the subset of remaining E_{NT} edges that have both their source and target inside the same KLA superstep, and therefore are inside an asynchronous execution section. As edges in E_{NT_C} cross a KLA superstep, they cross a global synchronization point. Therefore, they cannot contribute to wasted work, as their traversals will reach the target vertex at the same time as the correct traversal (from the actual BFS parent with the shortest distance-from-source), as guaranteed by the global synchronization. Therefore, we can establish an estimate of the potential wasted work as the cardinality of the set E_{NT_I}:

$$\Psi_{est}(G,k) \lessapprox E_{NT\_I}(G,k) \tag{3.5}$$

We know the number of tree edges (E_T) equals $V-1$. Therefore, $E_{NT} = E-(V-1)$. Using this, we can approximate the expected number of non-tree edges inside a KLA-SS as:

$$E_{NT\_I}^{Expected}(G,k) = \frac{E-(V-1)}{d} \times (k-1) \tag{3.6}$$

From equations (3.5) and (3.6) we get:

$$\Psi_{est}(G,k) \approx \frac{E-(V-1)}{d} \times (k-1) \tag{3.7}$$

This supports our intuition. Small-diameter graphs with large out-degrees have $E >> V$ and $d = O(1)$. Therefore, even a slight increase in k may lead to large amounts of wasted work, which would negate the benefits gained from removing a global synchronization. This implies that such graphs will fair better with smaller k values ($k = 1$). Long-diameter graphs with low out-degrees have $E \approx V$ and $d = O(V)$. This implies that such graphs work best with $k = d$, as the wasted work is negligible. Other graphs such as road networks, geometric meshes, etc. tend to fall in-between, and therefore benefit the most with $1 < k < d$.

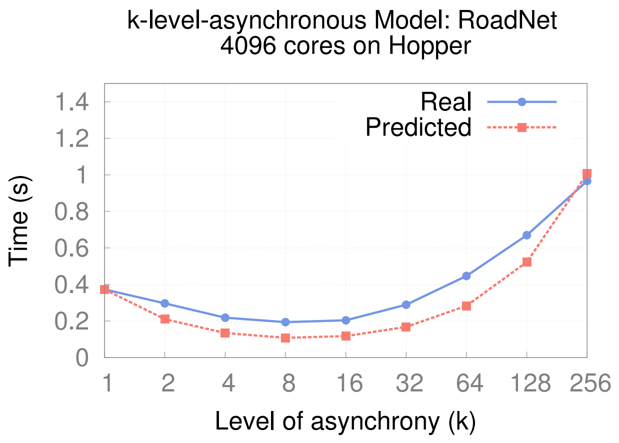
Furthermore, small-diameter graphs have a more even distribution of work, as most processors tend to be working in each superstep. Long-diameter graphs exhibit a work-load imbalance due to their low work per superstep. This imbalance can be alleviated by increasing the asynchrony for long-diameter graphs.

Other graph algorithms can be modeled in much the same way as BFS, apart from the penalty paid for asynchrony. Some algorithms (such as PageRank) may incur a penalty for buffering messages that arrive out-of-order as the asynchrony is increased, while others, such as k-core, do not incur any penalty.

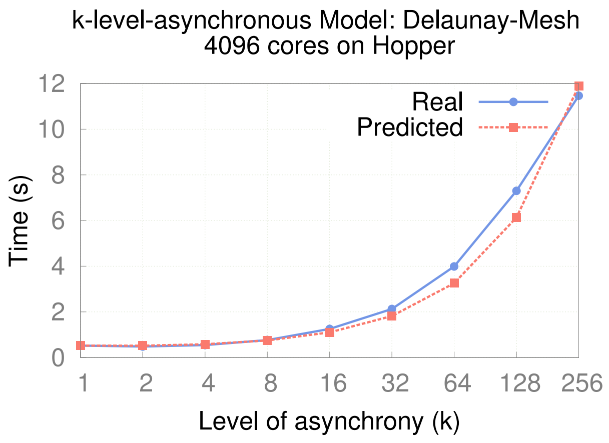
Finding k_{opt}: Estimating the Penalty-Function (Ψ_{est}) Determining Ψ exactly is expensive and only gives an upper-bound to the penalty paid for asynchrony. However, it can be closely approximated using a linear model (see Eq. 3.7). Assuming m gives an estimate of how fast the penalty increases with k, and b is the offset of the line: $\Psi_{est.}(G, k) = m \times k + b$ (both m and b depend on input-graph and machine). Two executions of the algorithm are needed to interpolate Ψ, which will provide the approximate curve of the running-time of the algorithm for varying values of k. The curve's minima can then be used to predict k_{opt}. This approach is mainly useful for application performing multiple traversals, such as in Brandes' betweenness centrality [17]. To find the equation for Ψ_{est}, we run the algorithm with $k = 1$, from which we can obtain the number of iterations (d) and the amortized cost (α) of processing a vertex. Running the algorithm a second time with $k = d$, which was found from the first execution, we can compute the penalty for a completely asynchronous execution of the algorithm ($\Psi_{est}(G, d)$), giving us the slope of the line for Ψ_{est}.

$\Psi_{est}(G, d)$ can be calculated for graph algorithms thus: 1) track the global number of visits to the input graph's vertices, 2) subtract from it the number of vertex-visits required to execute the algorithm in a BSP execution, 3) divide by the number of processors to get the average wasted work per processor.

Using these constants in the appropriate equation (Eq. 3.4), we can obtain a curve approximating the performance of the algorithm for the given input graph, for different values of k. The k_{opt} is the value for which the curve reaches its minima. We ran KLA on multiple graphs and compared the predicted vs. real k_{opt}, and found the model to approximate the trend well enough to suggest a reasonable k_{opt} value. Figure 3.2 shows two such results comparing the predicted times given by the KLA model on two different input graphs, to the actual execution times for varying k-values. This model was used to predict k_{opt} for experiments in Sections 8.5 and 8.7.



(a)



(b)

Figure 3.2: KLA Model: predicted vs. actual execution times on HOPPER for a BFS.

4. COMMUNICATION REDUCTION OPTIMIZATION – FOR SMALL-WORLD SCALE-FREE GRAPHS*

While KLA is effective in hiding the effects of a high-latency in a system, for many natural graphs, another bottleneck is effective bandwidth. Natural graphs tend to follow small-world scale-free behavior, and exhibit power-law degree distribution with small diameters. In such cases, the number of BSP iterations is often low due to the small diameter, and the work per iteration is consequently very high, due to high-degree vertices. Further, due to high-degree vertices, increasing asynchrony dramatically increases the wasted-work probability and cost. In such cases, using k=0 (i.e., the BSP model) is most effective. These graph can not be partitioned effectively due to their highly skewed degree distributions, and therefore any partitioning across a system results in a high number of cross-partition edges, which can cause scalability bottlenecks. These are no longer bound only by latency, so increasing asynchrony may not help. Improving performance and scalability in such cases requires increasing the effective bandwidth. We therefore propose a system which can improve effective bandwidth utilization by reducing the amount of bytes communicated.

In this section, we present an approach to transparently (without programmer intervention) allow fine-grained graph algorithms to utilize algorithmic communication reduction optimizations. In many graph algorithms, the same information is communicated by a vertex to its neighbors, which we coin algorithmic redundancy. Our approach exploits algorithmic redundancy to reduce communication between vertices located on different processing elements. We employ algorithm-aware coarsening of messages sent during vertex visitation, reducing both the number of messages and the absolute amount of communication in the system. To achieve this, the system structure is represented by a hierarchical graph, facilitating communication optimizations that can take into consideration the machine's memory hierarchy. We also present an optimization for small-world

* The techniques and some of the experimental results presented in this chapter are reprinted from *proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, "An algorithmic approach to communication reduction in parallel graph algorithms," 2015, pp. 201–212, Harshvardhan, A. Fidel, N. M. Amato, and L. Rauchwerger, ©2015 IEEE with permission from authors and IEEE.

scale-free graphs wherein *hub vertices* (i.e., vertices of very large degree) are represented in a similar hierarchical manner, which is exploited to increase parallelism and reduce communication. Based on these principles, we extend our framework to transparently allow fine-grained graph algorithms to utilize our hierarchical approach without programmer intervention, while improving scalability and performance. Experimental results using this approach on $131,000+$ cores show improvements of up to a factor of 8 times over the non-hierarchical version for various graph mining and graph analytics algorithms.

4.1 Out-degree and in-degree optimizations

A key pattern that arises in many parallel graph algorithms is the need for a vertex to propagate the same information to all vertices in its neighborhood. For example, consider the traditional fine-grained expression of the breadth-first search algorithm in Figure 2.1. First, the *vertex-operator* (Figure 2.1(a)) checks to see if a vertex is active (grey), and if so, it propagates its distance from the source to its neighbors using the *neighbor-operator* (Figure 2.1(b)). The neighbor-operator then updates the distance of the neighbor if needed, and marks the neighbor as active (grey) in the next iteration. A crucial observation is that the vertex-operator visits all of its neighbors with semantically identical information, which may lead to redundant communication in distributed-memory architectures. This communication pattern is present in a large class of important algorithms such as breadth-first search, betweenness centrality, connected components, community detection, PageRank, k-core decomposition, triangle counting, etc. These algorithms are used in graph mining and big-data applications where performance is critical.

The cost of memory accesses on large-scale distributed-memory systems is highly non-uniform. The communication patterns exhibited in graph algorithms executed on these systems are well-known to limit scalability. For this situation, optimizations to alleviate communication pressure can occur in several forms:

- Messages destined to the same processing element can be aggregated and combined into a single message.

- Creation of redundant messages that are a result of algorithmic redundancy (such as in breadth-first search) can be eliminated completely, and a single copy can instead be sent between processing elements.

- Bottlenecks caused by the presence of hub vertices can be mitigated by reducing incoming communication to hubs locally, and a single contribution can then be forwarded off-processor.

However, as algorithms are best expressed in a fine-grained manner that makes them oblivious to the graph structure, and as graphs themselves are represented as flat data-structures, only the first of these three optimizations is typically applied, as the knowledge of machine hierarchy is unavailable.

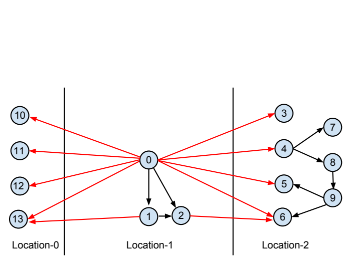
Our approach is to allow algorithms to be expressed in the natural vertex-centric manner while transparently applying communication optimizations without programmer intervention. The mechanism by which this is achieved is through the construction of a hierarchical representation of the input graph that is aligned with the machine hierarchy to identify local and non-local elements. By using this hierarchical representation and algorithm-level knowledge, we are then able to identify and transparently apply these communication optimizations for fine-grained algorithms.

4.2 Hierarchical model

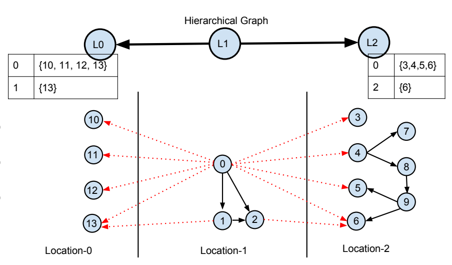
Our goal is to improve scalability and performance of graph algorithms by reducing the number and total volume of messages sent during execution. We first discuss locality-based communication optimizations for graphs in Section 4.2.1 and then follow with an additional optimization suited specifically for reducing bottlenecks due to hubs in Section 4.2.3.

4.2.1 Locality-Based Communication Optimization

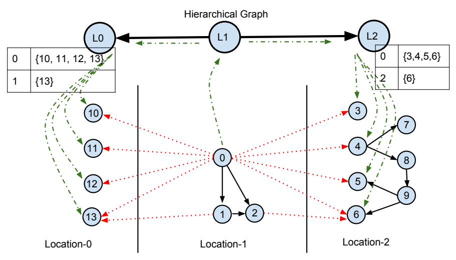
Communication in graph algorithms occurs between vertices through edges, which represent a source-target pair (s, t). For edges with the source s and with targets $t_0, t_1, \ldots$ stored on the same destination location, it is possible to aggregate messages for all such equivalent edges. Further, if all messages represent the same information, a single message can be sent to the destination



(a)



(b)



(c)

Figure 4.1: Creating a hierarchy: (a) A flat graph, with remote, inter-partition edges (red lines) and (b) its hierarchical representation with metadata. Using the hierarchy: (c) Remote communication during algorithm execution (shown in green). Hierarchical graph replaces inter-partition edges in original graph (red dashed-lines represent deleted edges) with a single super-edge between each partition (solid black lines on top-level graph). Generalizable to multiple levels of machine hierarchy.

location and then applied to $t_0, t_1, \ldots$. This concept can be applied recursively by considering a grouping of vertices and identifying same destination location pairs amongst these groupings. The same communication reduction techniques apply to all levels of this recursive process.

To enable communication reduction, we need a model of the system that captures the locality of the graph. To achieve this, we overlay the input graph on top of the machine hierarchy, thereby creating a hierarchically coarsened graph. This is described in Section 4.2.2. Once the graph is coarsened, we use a translation layer, called the hierarchical paradigm (Algorithm 2), to execute the fine-grained algorithm on the coarsened hierarchy. This is described in Section 4.2.4. Combining the fine-grained specification with the hierarchy results in a coarsening of the algorithm itself, and allows for the use of algorithm-driven communication optimizations.

4.2.2 Creating in-degree hierarchy

The transformation of the flat input graph to a hierarchical graph based on the machine-locality information is a mutating process that groups the graph's vertices into partitions based on the locality of each vertex provided by the machine hierarchy. This replaces multiple vertices in a partition, along with intra-partition edges, with a single super-vertex representing the underlying sub-graph. All edges between two partitions are replaced with a super-edge (composite edge) representing the coarsened communication between two sub-graphs. The super- vertices and super-edges form a super-graph. This process is shown in Figure 4.1 and detailed in this section.

A hierarchical graph consists of two or more levels of graphs, where the graph at level i is a super-graph of the graph at level $i-1$. The lowest level is the base-level. For building our hierarchy, we first partition the input graph into sub-graphs, where each sub-graph has a similar number of vertices, and assign each sub-graph to a processor. This partitioning can be computed with an external partitioner to reduce the edge-cut of the graph, which may improve overall performance, while the hierarchical approach will take care of the remaining cross-edges. This forms the base-level of our hierarchical representation (G_0). Next, we create a hierarchy of M levels on top, matching the machine hierarchy. For graph G_i at every level ($0 \leq i \leq M$), we create a super-graph G_{i+1}, such that each vertex in G_{i+1} represents a sub-graph partition of G_i. Super-edges are added between two super-vertices of G_{i+1} if there exist inter-partition edges of their corresponding sub-graphs in G_i. The inter-partition edges of lower-level graph G_i are then removed and their information is stored on the corresponding target's super-edge. This is done to preserve the locality of edges, as all edges that point to the same target are stored at that target vertex's location.

Our hierarchical graph consists of the base-level graph, along with one or more levels of super-graphs of the base-level graph, with super-vertices representing each vertex-partition and super-edges representing inter-partition edges (communication). The hierarchical representations obtained thus naturally expresses the machine topology. We note that creating the hierarchy is a one-time event that happens immediately following graph construction. Once the hierarchy is created, multiple algorithms can take advantage of it. We show the overhead in graph construction time in Section 8.9.7, while a bound for space overhead is provided in Section 4.3.2.

4.2.3 Distributed Hubs Optimization

Hub vertices, or hubs, are vertices with a high in- or out- degree. Many small-world scale-free graphs, such as web-graphs and social-networks, exhibit a power-law degree distribution, with most vertices connected to a few other vertices, but very few ($< 1\%$) vertices connected to an extremely large number of vertices. While our locality-based hierarchy reduces outgoing communi-

Algorithm 1 Hierarchical Graph Creation

```
Input: Graph $G_i$, int $N$
 1: if $i = N$ then
 2:     return
 3: end if
 4: Graph $G_{i+1}$ = Partition($G_i$, MachineHierarchy)
 5: for all $(u, v) \in Edges(G_i)$ do
 6:     if $Super_{i+1}(u) \neq Super_{i+1}(v)$ then
 7:         $E_i := E_i \backslash (u, v)$
 8:         $E_{i+1} := E_{i+1} \bigcup (Super_{i+1}(u), Super_{i+1}(v))$
 9:     end if
10: end for
11: Hierarchy.add($G_{i+1}$)
12: Construct($G_{i+1}, N$)
```

Algorithm 2 Hierarchical Graph Paradigm

```
Input: Graph $G_0$, Locality-Hierarchy $H$, Hubs $H_{hubs}$
 1: while active vertices $\neq \emptyset$ do
 2:     for all active vertices $v_i \in G_0$ par do
 3:         $W_i \leftarrow$ process($v_i$)
 4:         for all neighbors $u_i \in G_0$.adjacents($v_i$) do
 5:             if $H$.parent($u_i$) $=$ $H$.parent($v_i$) then
 6:                 visit-neighbor($u_i$, $W_i$)
 7:             end if
 8:         end for
 9:         for all neighbors $p_i \in H$.adjacents($H$.parent($v_i$)) do
10:             async(visit-neighbor($p_i$, $W_i$))
11:             // apply visit-neighbor recursively on children of $p_i$ who are neighbors of $v_i$
12:         end for
13:         for all hubs $h_i \in H_{hubs}$.adjacents($v_i$) do
14:             visit-neighbor($h_i$, $W_i$)
15:         end for
16:     end for
17: end while
```

cation caused by hub vertices, communication can be further reduced by specialized optimizations for high in-degree vertices.

We introduce a new approach to reduce all messages to a hub from the same processor to a single value which can then be applied to the hub. We assume that the applied operator is both associative and commutative. When a vertex tries to update (send a message to) a hub vertex, the operator is applied to a local representative for that hub. At the end of each superstep of the algorithm, the results of the local updates are flushed and applied to the original hubs. This in effect distributes the work for hubs across the system.

Construction of the distributed hubs occurs by first identifying the hub vertices using a simple scan through the graph, and then creating a super-graph where each super-vertex contains metadata about the hub vertices, and all edges to the hub are replaced by the super-edge. This metadata can only be written to and not read from, which does not require the value to be kept coherent, and works with our asynchronous update model (Section 2.2).

We handle the identification of hubs and creation of the hierarchy automatically using the degree (number of edges) of each vertex. The user may choose to provide a cutoff for this size, beyond which vertices will be treated as hubs, or they may choose to use the top k% vertices as hubs. In either case, the framework identifies the hubs so algorithm-developers do not need to account for this.

This approach is similar to recent work presented in [18], which replicates the hub vertices on other processors (called hub-representatives), allowing local vertices to read from and write to the hub vertices. However, their approach replicates data for each hub vertex, which needs to be kept synchronized, leading to extra communication and affecting scalability. Their approach also requires algorithms to be aware of the existence of these hub representatives and need to be modified to specify the synchronizing and reduction behaviors of the representatives. In contrast, our approach is framework-level, and keeps the algorithm itself agnostic to hubs. Further, our approach does not replicate vertex data.

4.2.4 Using the hierarchy

The hierarchical graph paradigm is presented in Algorithm 2. In order to execute algorithms, the paradigm proceeds in *supersteps* similar to the bulk-synchronous parallel (BSP) model [14]. However, the supersteps are executed in a manner that is aware of the machine hierarchy. Within each hierarchical superstep, the paradigm processes the active vertices in each level of the graph hierarchy iteratively. For each active vertex in the base-level graph, the results from processing it are sent to its neighbors in the same partition (lines 4-8). Any cross-partition edges for that level of hierarchy are ignored, as they will be processed by the upper-level of the hierarchy. The results from processing the active vertex are then forwarded to the super-vertex of the current partition (lines 9-12), which then transmits them via super-edges to the target super-vertices. If the *process* function in line 3 sends the same update to multiple neighbors (by calling VisitAllNeighbors, for example, as in the case of BFS, PageRank, connected components, k-core, betweenness centrality, etc.), the paradigm only creates a single copy of the update to send via the super-edges. We also provide a specialization for high-degree vertices (described later in Section 4.2.3) that uses a similar communication reduction mechanism (lines 13-15). Finally, the updates are then recursively pushed to the respective target vertices in the lower-level graphs at the target partition, whose edges were replaced by the super-edge.

The graph algorithms themselves do not change and are hierarchy-oblivious, allowing us to reuse fine-grained vertex-centric algorithms on coarsened graphs. This decouples users from the details of machine and locality exploitation.

4.3 Analysis

In this section, we describe how we exploit both the redundant nature of many parallel graph algorithms and the power-law characteristics of many input graphs, resulting in a reduction of total communication in the system.

4.3.1 Communication Volume

The hierarchical approach can reduce communication in the system by reducing the number of bytes required to update neighboring vertices. Without loss of generality, we assume a non-multi-edged graph where two vertices are connected by at most one edge, and that the graph algorithm visits each vertex (and consequently every edge, due to the non-multi-edged property). We also assume a BSP/level-synchronous model [14, 9] where an active vertex performs some computation and updates its neighboring vertices with the result of this computation. The BSP model gives the most conservative estimate for this analysis. Any algorithm that communicates more, such as in an asynchronous model with redundant work, will observe a greater communication reduction using our approach. Let us assume the size (in bytes) of this result is $r\ bytes$, and that it is uniform for all vertices. In the traditional BSP case, used in existing graph libraries such as Pregel [9], the Parallel Boost Graph Library [7, 19] and the Graph500 benchmark reference implementation [20], this will imply potentially $O(|adj(v)|)\ bytes$ being communicated, where $adj(v)$ gives adjacencies of vertex v. In fact, the amount of communication is:

$$Comm(v) = \sum_{i \in P} m_i(v) \cdot r \ \ bytes \tag{4.1}$$

Where P is the number of processors, $m_i(v)$ gives the number of adjacents of vertex v that are stored on processor i, and r is the size of the message being sent. Traditional BSP libraries employ aggregation to reduce the number of messages. However, the total number of bytes stays the same.

In many cases, for a large class of graph algorithms, such as breadth-first search, PageRank, k-core decomposition, connected components, strongly-connected components, topological sort, betweenness centrality, triangle counting, etc., the vertex sends the same information to all its neighbors. In such cases, the amount of data sent (in bytes) can be reduced using our approach. The following reduction is applicable to such algorithms. For other cases, we may not observe any reduction in the amount of data, but we will still reduce the number of messages sent. The amount of data communicated in the first case is the lowest possible given the algorithm.

Using a hierarchical approach, the vertex may only need to send the result of its computation once for every *super-edge* instead of once for every edge. Therefore, the amount of communication for a hierarchical graph can be given by:

$$Comm^H(v) = \sum_{i \in P} \begin{cases} 1 & : m_i(v) \geq 1 \\ 0 & : m_i(v) = 0 \end{cases} \cdot r \;\; bytes \tag{4.2}$$

This effectively means that for any vertex the hierarchical approach performs the least amount of communication possible (both in number of messages and total size of communication) for a given partitioning strategy, as it sends only a single result to each processor that stores any of the vertex's neighbors. Any lower communication can not guarantee the correctness of a general graph algorithm, without changing it.

The total bytes communicated across the system in the hierarchical approach for a graph G can then be given by:

$$\begin{aligned} Comm^H(G) &= \sum_{v \in V} Comm^H(v) \\ &= \sum_{v \in V} \sum_{i \in P} \begin{cases} 1 & : m_i(v) \geq 1 \\ 0 & : m_i(v) = 0 \end{cases} \cdot r \;\; bytes \end{aligned} \tag{4.3}$$

This gives the upper bound of $O(min(V \cdot P, E))$. However, we note that the worst-case upper bound is equivalent to the case of a dense graph, where every vertex is connected to every other vertex and the communication for the traditional level-synchronous approach would have otherwise been $O(V^2)$. For sparser graphs, Equation 4.3 gives a more accurate estimate. We show empirical evaluation of this communication reduction in Section 8.9.2.

4.3.2 Space Overhead

Creating a hierarchical graph adds space overhead compared to the traditional flat graph approach. In this section, we provide a bound on this overhead. The hierarchical graph G_1 uses a single vertex corresponding to each partition of the base-graph G_0, which requires $O(p)$ space overall, assuming p partitions in G_0, one per processor. Further, the number of edges in G_1 correspond to the number of partitions in G_0 that have edges between them, with one super-edge per pair of neighboring partitions in G_0. In the worst-case, there can be $O(p^2)$ super-edges in G_1 corresponding to a complete graph in G_0. Therefore the bound on the size of G_1 is $O(p^2)$ for the entire graph, or $O(p)$ overhead per-processor for using the hierarchy. Note that the metadata on super-edges does not contribute to overhead, as it replaces the deleted inter-partition edges from G_0, keeping the total size constant in this regard.

For the hub-hierarchy, every processor stores metadata for any hub vertices that have an edge to a vertex on that processor. This implies $O(|hubs|)$ storage per processor, only if there is an edge to that hub from that processor. However, since the number of such hubs is generally small, the space overhead is small in practice. Further, our algorithm for creating the hub-hierarchy is parameterized based on a user-specified lower-limit on the size of hub vertices, so the number of vertices treated as hubs can be varied to suit any space constraints on the system.

4.4 Implementation

While the hierarchical approach is generally applicable to any distributed memory graph library, for this work, we implemented our approach in the STAPL Parallel Graph Library (STAPL GL) [1] to evaluate performance, due to STAPL GL's ease of use and modification, and scalable performance [1, 2] (Section 8.3). Section 7, gives an overview of STAPL GL and describes the relevant features and extensions needed to support the hierarchical approach.

4.5 Related Work

To enable graph algorithms to scale, various different approaches have been adopted. Many implementations [20, 9], including our baseline implementation, aggregate messages being sent to a processor in order to reduce the number of messages sent. However, the *total number of bytes sent is not reduced*, but merely concatenated to form larger messages. Pregel [9] also allows users to specify a *combiner* that may be used to reduce multiple incoming messages to a given vertex to a single value per processor. However, this is not applicable to all algorithms and nor is it guaranteed to be executed in Pregel. Moreover, combining can only address reduction of fan-in, not high out-degree/fan-out vertices.

Another approach is to use ghost vertices to facilitate communication, which cache values of neighboring vertices stored on other processors. However, such vertices need to be coherent with their original vertices, and do not scale well in practice, due to storage and communication overhead of maintaining the ghosts, as shown in [21]. This approach is used by the Parallel Boost Graph Library [7], which we compare against in our experiments.

There have also been methods that propose a 2-dimensional partitioning of the graph and its edges to achieve better scalability by reducing communication cost [22]. To use such an approach, the algorithms need to be rewritten to account for a distributed edge-list and made aware of the underlying data-distribution, making this method difficult to use in practice. This method also does not reduce the message size, but does distribute the computation more evenly across partitions than 1-D partitioning. However, 2-D partitioning does not consider the locality of the target vertices of the edges being partitioned, and thus may in fact lead to more hops for the message to reach its destination, for example, one hop to get to the processor where the edge is stored, and a second hop to get to where the target-vertex of that edge is stored (as edges are not co-located in 2-D partitioning), generating extra communication. Our approach produces co-located edges in addition to lowering the computational imbalance, without user intervention.

Some approaches [21, 23, 24, 18, 25] have also proposed splitting hubs across multiple processors. PowerGraph [21], a version of GraphLab [13] designed for processing scale-free graphs uses the concept of *vertex-cuts/vertex mirroring* to split hub vertices across partitions. However, as mentioned in [21], due to maintaining the vertex-splits, their ghost vertices need to be kept synchronized across all machines it spans. This is addressed in PowerGraph by minimizing the number of processors across which the hubs are split to lower the synchronization costs. However, this limits the available parallelism for processing the hubs. Our approach is not limited by this, since we do not have ghost vertices and do not need to maintain state information across partitions. We compare our base-line with the latest available version of PowerGraph in Section 8.3. Pregel+ [25]

also uses mirroring, which reduces communication by partitioning the edges of high-degree (hub) vertices. However, this does not have much benefit in their published results, and performance actually degrades when the number of hubs is large. Our hierarchical technique, on the other hand, is applied to all vertices and consistently shows benefits, even when used for low-degree vertices.

On the other hand, [24, 18] create copies of the hubs on all processors, which may be wasteful if the hubs do not have adjacent vertices on all processors, and increases storage and communication overhead. In these cases, the hubs need to be coherent across the processors, which leads to extra communication, and a computational imbalance may still remain. This approach of splitting hubs also does not address non-hub vertices due to the overheads involved in splitting vertices and the storage requirements for replicating data. The algorithm-writers too need to be aware of the presence of such hub-representatives, and the communication between them, requiring the user to re-write their algorithm. We improve upon this work with our hubs approach which allows for local reduction of updates for hub vertices on processors where they are needed, while alleviating the need to keep them coherent. Further, it is transparent to the algorithm. This is in addition to our locality based hierarchy, and is explained in Section 4.2.3.

The authors in [25] propose a request-response paradigm where a vertex can request data from another random vertex and it will be available in the next superstep. For algorithms that need this pattern, it can reduce the amount of back traffic by not sending duplicate values to the same processor. However, this is only applicable for a restricted set of algorithms explored in their paper, and can not be applied to widely used algorithms such as traversals, PageRank, etc. It also requires the algorithm to be modified to incorporate this paradigm for the cases where it is applicable. Their results show a modest improvement in performance for such cases.

Our hierarchical approach operates at a semantic level, allowing us to reduce the information sent over the network for both outgoing and incoming edges, thereby improving scalability. This is done for all vertices that have cross-processor edges, not only for hub vertices. Since information is not replicated, and due to using an asynchronous push-model, we do not incur significant storage or communication overheads endemic to previous approaches. Crucially, due to operating at a semantic level, the algorithm itself remains unaware of the hierarchy, allowing the reuse of existing fine-grained graph algorithms as-is with the hierarchy.

5. OUT-OF-CORE GRAPH PROCESSING[*]

There exist extreme-scale data sets that do not fit into the main memory of parallel systems and require out-of-core storage and processing. Out-of-core graph processing is disk-bound, where minimizing non-sequential reads and reducing number of bytes written is critical for performance. Such applications typically also observe both high-latency and low bandwidth characteristics while processing graphs from secondary-memory. Existing systems are confined to a single shared-memory node, due the model required for fast disk-based processing not being effective in a distributed-memory setting. Such systems are also unable to minimize the number of reads required due to their model (GraphChi, for example, uses a pull model instead of a push model, leading to $O(p^2)$ reads). To address these, we use the KLA model to eliminate non-sequential disk reads and minimize writes. Further, due to the push nature of KLA, it can seamlessly extend to distributed-memory systems. We also introduce optimizations to further reduce the number of bytes written to disk.

Our approach for out-of-core processing is similar to paging, as we partition the input graph into subgraphs (logical partitions), such that each subgraph may fit in main-memory. When needed, the subgraphs are paged-in from disk to main-memory and processed. In this *asynchronous push model*, vertices of a subgraph are only processed in-memory, and updates to neighboring vertices are asynchronously pushed to the subgraphs as follows. If the subgraph containing a neighboring vertex is also in memory, the vertex is updated. If the subgraph is stored on disk, the update is written to disk and applied the next time the subgraph is loaded. The approach differs from paging, as we reflect our pages at a logical (subgraph) level, versus a non-semantic (kilobyte, megabyte) level. This allows us to take advantage of temporal locality in subgraphs.

We also implement multiple optimizations to reduce disk I/O. These optimizations include

[*] The techniques and some of the experimental results presented in this chapter are reprinted from *proceedings of the 29th International Parallel and Distributed Processing Symposium (IPDPS)*, “A hybrid approach to processing big data graphs on memory-restricted systems,” 2015, pp. 799–808, Harshvardhan, B. West, A. Fidel, N. M. Amato, and L. Rauchwerger, ©2015 IEEE with permission from authors and IEEE.

caching *hub vertices* (i.e., vertices with very large degrees) when they are written to disk, skipping graph-structure writes for unmodified graphs, and over-partitioning subgraphs to utilize the RAM to the fullest extent. The optimizations are system-level, and therefore do not involve modifying the algorithms.

This technique and its implementation is described in more detail in Section 7. The next section describes how algorithms are expressed in the asynchronous push model.

In this section, we describe our hybrid approach to processing in-memory and out-of-core graphs. While our approach is applicable to frameworks using the asynchronous push model, for this work we assume that algorithms are expressed using the KLA two-operator algorithmic specification presented in Section 2. Using this specification, our approach can utilize the vertex- and neighbor-operators to keep the algorithm unchanged from its in-memory variant, while our paradigm handles the execution (in-memory or out-of-core) and storage (which parts of the graph should be in-memory vs. on-disk).

5.1 Introduction

Over the past few years, many systems specialized for graph-processing have emerged to address the issue of processing large graphs ([7, 1, 10, 13, 9]). These systems are either designed for in-memory or disk-based graph-processing. In-memory systems, whether distributed or shared-memory, store and process the entire graph in RAM. While this can provide good performance, the size of the graph that can be processed is limited by the amount of RAM available in the system. Disk-based systems, on the other hand, are not limited by RAM availability, but sacrifice performance due to disk I/O. Recently, a hybrid system (GraphChi [12]) was proposed to allow a single-node system to process large graphs using a parallel sliding-window approach. However, this was limited to a single shared-memory node.

In this section, we propose a graph processing approach that can scale well from small memory-restricted systems to large distributed-memory machines. This RAM-disk hybrid graph-processing system provides a unified approach to utilize the available resources (RAM, disk, cores) efficiently and seamlessly. Our system decouples algorithms from the details of the machine, allowing users to write fine-grained vertex-centric algorithms, that can run efficiently without modification on different systems. We use an approach similar to *memory paging*, but applied to subgraphs, that loads subgraphs of the graph in memory, processes them, and then stores them back to disk. A crucial difference from paging is that while paging uses fixed-size pages, our approach uses partitions based on graph structure. This preserves structural locality that can reduce disk I/O. We use an *asynchronous push model* that allows paged-in vertices to be processed, while updates to their neighbors are asynchronously pushed, or deferred, until the neighbors are loaded. We also propose system-level optimizations that do not require changes to algorithms, but can reduce disk I/O. An implementation of our approach in the STAPL Parallel Graph Library [1] allows us to process large graphs on systems ranging from small-scale systems such as off-the-shelf PCs or Android tablets, to large high-end clusters. Our results show that our subgraph-paging based approach and asynchronous push model, together with optimizations, provides $3 - 12\times$ faster graph processing on a single node than the best alternative, GraphChi, and extends efficiently to multiple nodes which GraphChi cannot.

Our contributions include:

- A hybrid subgraph-paging based approach to processing large graphs, allowing both in-memory and out-of-core processing.
- An implementation that transparently allows fine-grained level-synchronous graph algorithms to benefit from our hybrid approach without code modifications.
- An experimental evaluation showing good scalability and showing improved performance over existing disk-based and in-memory graph processing frameworks on systems ranging from small Android tablets to off-the-shelf PCs to large clusters with $16,000+$ cores.

5.2 Graph Storage

We start with an input graph partitioned into p subgraphs. Each subgraph is assigned to a *location* (a single processing element), and a location can have multiple subgraphs assigned to

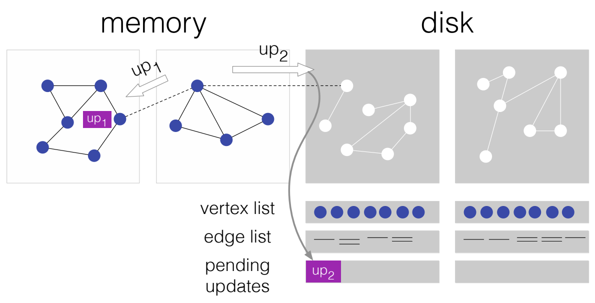


Figure 5.1: Diagram of the storage paradigm with two subgraphs in memory and two subgraphs on disk. An update on a loaded vertex is being applied in memory while an update to an un-loaded vertex is being stored in the pending updates shard.

it. The location to which a subgraph is assigned is its *home location*, which is responsible for managing the subgraph and executing requests on its vertices. Each subgraph can be present either in memory (*loaded*) or on disk (*un-loaded*), and each location can independently decide when to load and un-load which subgraphs based on memory availability.

File format for sub-graphs. Each subgraph is stored in a binary format across three shards (Figure 5.1). The first shard is the *vertex list*, which contains the vertices of the subgraph, along with any vertex data (such as level for BFS, rank for PageRank, and ID for connected components). The second shard is the *edge list*, which contains the list of edges for each vertex, in order, including any data for the edges. These are stored separately to allow for optimizations when a vertex's edges need not be loaded, for indexing into the vertex list, and also to enable us to bypass rewriting the edge list shard unless the graph structure changes (Section 5.4). The third shard, *pending updates*, stores any incoming and pending updates to vertices belonging to that subgraph. When the subgraph is loaded, the updates shard is read and all outstanding updates are applied to corresponding vertices.

5.3 Processing Out-of-Core Graphs

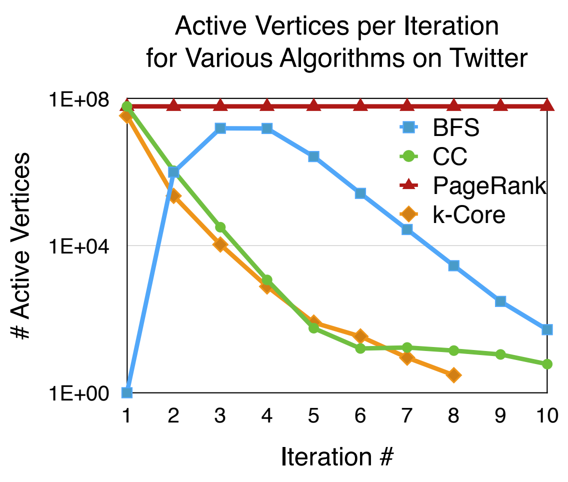
We use an approach similar to paging to process out-of-core graphs (Figure 5.3). The computation proceeds in bulk-synchronous supersteps, and for every superstep, each location loads one or more of its active subgraphs (based on available RAM) in some schedule (for our experiments, we choose a round-robin policy), processes the vertices and stores the subgraphs back to disk, including any modified state. A different subset of subgraphs is then loaded and processed, until all active subgraphs have been processed for the given superstep. The process is then repeated for the next superstep until there are no remaining active vertices (i.e., vertices that will actively compute in the current superstep).

If the machine has sufficient RAM to store the graph and metadata in-memory, all subgraphs can be kept in RAM, and no penalty is paid for disk access. If the RAM is not sufficient, the paradigm will load and unload subgraphs to allow the seamless execution of the algorithm.

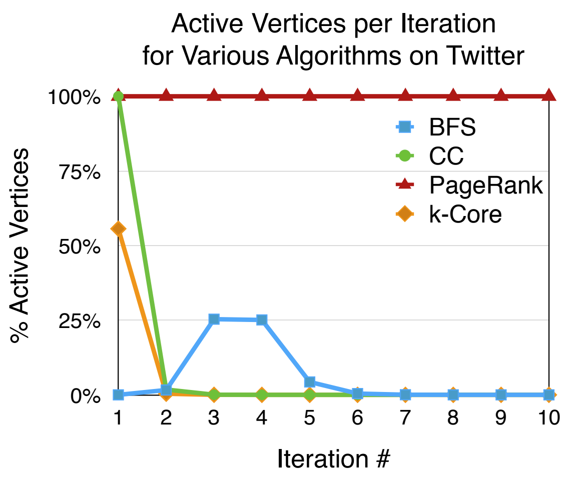
Inactive vertices or sub-graphs. Graph algorithms are often iterative, and for a large class of graph algorithms, not every vertex is actively processed in each iteration. An example of this is shown in Figure 5.2, which plots an execution profile in terms of the number of active vertices, for the first ten iterations of various representative algorithms. As can be observed, the algorithms shown, with the exception of PageRank, only process a very small fraction of their vertices in any given iteration, and only a few iterations process any significant fraction of the vertices. This provides an opportunity to skip reading and loading vertices that will not be processed.

In our approach, if a vertex's value is updated, it is marked as *active* for the next superstep. This implies that the vertex may potentially be processed in the next superstep using a vertex-operator, which can access its adjacent edges. For such vertices, it is required to load their adjacent edges. However, for *inactive* vertices in a given superstep, we can skip the loading of their adjacent edges, as it is guaranteed that such vertices will not have the vertex-operator applied to them. The inactive vertices and their values are still loaded in memory, however, to allow incoming updates to be applied to them from their neighboring vertices.

As an extension, if all vertices of a given subgraph are *inactive* for a given superstep, we skip the loading of the entire subgraph for that superstep. This is common for many graph traversals (such as BFS), which start from a single vertex, and only a few vertices are active in most iterations. This is also commonly observed in other graph algorithms (Figure 5.2). In such cases, a significant amount of time can be saved by not reading and loading entire inactive subgraphs from disk.



(a)



(b)

Figure 5.2: Variation in active vertices in the first ten iterations of various algorithms on the Twitter graph, shown as (a) count on log-scale, and (b) percentage of total vertices.

Figure 5.4(a) shows the impact of skipping inactive subgraphs (Opt1), along with the optimization of skipping inactive vertices (Opt1+2) vs. the baseline execution time for the Graph 500 benchmark input with 16 million vertices and 256 million edges on a PC with 4GB of RAM. Skipping inactive subgraphs by itself may not provide a significant improvement as opportunities of skipping entire subgraphs for Graph 500 inputs are few. However, for graphs that can be partitioned well, this can provide substantial benefits.

Updates. We use a policy which defers updates to subgraphs not present in memory to avoid unnecessary paging. Updates produced during processing of in-memory subgraphs are asynchronously forwarded to the home location of the target vertex. If the target vertex is present in RAM, the update is applied to it. If the target vertex is stored on disk, the update is written to a *pending updates* shard corresponding to the target vertex's subgraph, and applied when that subgraph is next loaded. This process is illustrated in Figure 5.1. The deferred paging policy works, as our asynchronous update model only guarantees that the effect of updates in superstep i will be visible in superstep $i + 1$, and therefore, the effects of updates need not be immediately visible. Another benefit of the push-model is that it does not have to load and read neighboring vertices'

```
for_each(superstep ∈ algorithm)
  for_each(subgraph ∈ graph)
    subgraph.load();  // load subgraph to RAM
    subgraph.apply_pending_updates();

    for_each(v ∈ subgraph.vertices())
      updates = vertex-operator(v);

      Asynchronously forward updates to home locations:
        Apply updates to vertices in in-RAM subgraphs;
        Store updates to on-disk subgraphs as pending;

    subgraph.store();  // store to disk
```

Figure 5.3: Pseudocode for the off-core graph paradigm.

states, and can simply send updates to neighbors asynchronously.

Dynamic graphs. Our paradigm supports dynamic graph operations such as adding and deleting edges or adding and deleting vertices as the computation progresses. These operations are treated in the same way as regular updates described above. Modifications to graph structures are applied directly to subgraphs loaded in RAM, or are stored to an *outstanding modifications* shard if the subgraph is stored on disk and applied immediately upon loading. If a subgraph is modified during a superstep, a dirty-bit is set to indicate the modification, ensuring that the modified subgraph will be written to disk with the updated changes.

5.4 Optimizations To Reduce Disk I/O

The majority of time for processing out-of-core graphs is spent in disk I/O. Therefore, we propose and implement several optimizations to reduce the number of bytes that need to be read from or written to disk. These optimizations are internal to the storage paradigm and therefore do not require any changes to the user algorithm.

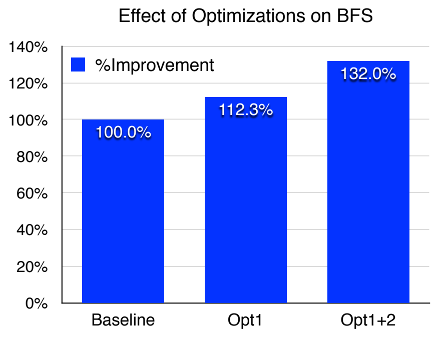
Graph structure writes. Many algorithms do not change the graph structure (i.e., they do not add or delete vertices or edges) but only update the values stored on vertices and edges of the graph. For such algorithms, we do not need to write back the structural information of the graph when storing a subgraph. In our implementation, we achieve this by separating the structure (edges) and values into separate shards (Section 5.2). The edge list shards are only updated if the graph structure was modified. The vertex list shards contain the vertex IDs and vertex values, and therefore are updated if the values change.

Multiple sub-graphs (over partitioning). Multiple subgraphs can be loaded in RAM at the same time. Hence, using smaller subgraphs allows finer-grained control, while still utilizing the available RAM to the fullest extent by loading multiple small subgraphs at once. This also increases the probability of a subgraph being inactive and therefore skipped. However, there is a trade-off between the size of a subgraph and the overhead of metadata for each subgraph.

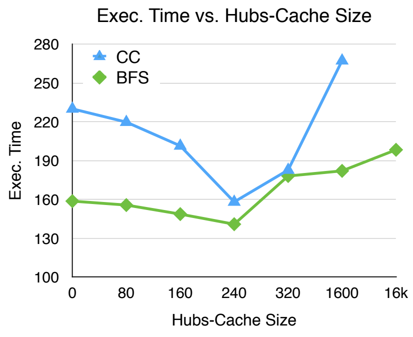
In-memory cache for hub vertices. Many social-networks and web-graphs exhibit small-world scale-free behavior with a power-law degree distribution. This implies the existence of *hub vertices* which are connected to a large number of neighbors. For such vertices, especially if they have a large in-degree, there may be multiple updates being applied to them from their neighbors in the same superstep. In such cases, if the subgraph containing the vertex is stored on disk, the updates will be written to the updates shard. These numerous writes can be avoided by using an in-memory write-back cache for such hubs. In our implementation, the hub vertices are identified at the time of graph creation (or modification) using a simple linear scan of the vertex degrees. An in-memory cache corresponding to each subgraph maps the hub vertices within the subgraph and stores its cached value.

When the subgraph is present in RAM, the cache is inactive and any updates are directly applied to the hub vertex. However, if the subgraph exists on disk, the cache is active and applies all incoming updates to the cached value. When the on-disk subgraph is next loaded, the cached value is written back to the original hub. Since the number of such hubs is usually small, the cache does not use a significant amount of resources. However, the time saved in writing, and then later reading and applying the updates can be significant. The size of the cache may be user specified, or determined from profiling the system.

Figure 5.4(b) shows the effect of cache size on algorithm performance for the connected components (CC) and BFS algorithms on a Graph500 benchmark graph. We found a value of 15



(a)



(b)

Figure 5.4: Effects of optimizations on performance: (a) Skipping inactive subgraphs (Opt1) and inactive vertices (Opt2), (b) Effect of varying size of hubs-cache for BFS and connected components (CC). Both plots run on Graph 500 input graph with 16 million vertices and 256 million edges, on a PC with 4GB RAM.

hubs per 1 million vertices to provide performance benefits of $15 - 40\%$ for a system with 4GB RAM. Increasing the cache size beyond this results in larger overhead of maintaining the cache and diminishing benefits due to the power-law degree of Graph500 inputs.

5.5 Analysis of I/O Costs

We use the I/O model described by Vitter [26, 27] to analyze the I/O costs of our approach. This model expresses the cost of an algorithm using the number of block-transfers of size B bytes from disk to main memory. An assumption in their model is that a disk can transfer a contiguous block of B bytes of data about as fast as it can transfer a single bit. Our analysis makes a worst case assumption that none of the optimizations in Section 5.4 are applied and that all vertices in the graph are active.

We assume a graph of $|V|$ vertices and $|E|$ edges, of total size N bytes, is partitioned into m subgraphs, and each subgraph of size $\frac{N}{m}$ bytes contains the subset of the graph's vertices and edges that make up the partition. Based on this, all subgraphs of a graph can be transferred using $P = \frac{N}{B}$ block transfers. We also assume p_{node} compute nodes in the system, each with local disk and with main memory of M bytes, where $\frac{N}{m} \leq M$, and that each compute node stores $\frac{m}{p_{node}}$ subgraphs on local disk. Then, the total I/O cost (C_B) of our approach per superstep of the algorithm is given by summing the cost of reading and writing for the superstep:

$$C_B(G) = C_B^{Read}(G) + C_B^{Write}(G) \tag{5.1}$$

For each superstep, in the worst case, we process every subgraph. Therefore, the total number of block transfers for reading per node is at most $\frac{P}{p_{node}}$. Similarly, for each superstep, in the worst case, we would update the neighbor for every edge in the graph, giving a total number of block transfers for writes per node of $\frac{|E|}{B \cdot p_{node}}$. Therefore:

$$C_B(G) \leq \frac{P}{p_{node}} + \frac{|E|}{B \cdot p_{node}} \tag{5.2}$$

Finally, if the algorithm performs S supersteps, the total I/O cost for processing the graph is given by:

$$C_B^{Algo}(G) \leq S \cdot (\frac{P}{p_{node}} + \frac{|E|}{B \cdot p_{node}}) \tag{5.3}$$

For comparison, as shown in [12], using the same model, GraphChi has an upper-bound of $\frac{4|E|}{B} + \Theta(P^2)$ for the I/O cost of a *single superstep*. This is significantly more than in Equation 5.2, which is linear in the number of partitions. This is due to GraphChi's parallel sliding-window approach, where each superstep is executed in P execution intervals, and each execution interval requires $O(P)$ non-sequential disk reads to load the edges from $P - 1$ sliding shards for each of the P execution intervals in that superstep, giving a quadratic complexity. We also note that as our approach supports distributed-memory machines, each with independent disks, we can reduce our I/O costs by a factor of p_{node}, whereas GraphChi is limited to a single node.

5.6 Implementation

We implemented our approach in the STAPL framework, by extending the STAPL Parallel Graph Library (STAPL GL), which previously supported in-memory graph computations only, to support out-of-core processing. The API of the STAPL GL did not change, as our technique is transparent to the user. This also allowed us to use the existing STAPL GL algorithms *without modification*. The details of this implementation is given in Section 7.2.

5.7 Related Work

In this section, we discuss some of the important existing graph processing systems, as well as theoretical work on processing out-of-core graphs.

In-memory graph libraries allow fast processing of graphs in parallel. These can be classified into two categories: distributed-memory and shared-memory. Shared-memory graph libraries such as Galois [10] and Multi-Threaded Graph Library (MTGL) [8] operate on a single node and are therefore restricted by the amount of RAM on the node, which severely limits the maximum size of the graph that can be processed. In contrast, distributed-memory graph libraries such as the Parallel Boost Graph Library (PBGL) [7], GraphLab [13] and Giraph can utilize multiple nodes to process graphs. However, even here, the maximum size of the graph that can be processed is limited by RAM. A detailed performance comparison of GraphLab, Giraph and other popular graph processing systems was shown in [28]. An in-memory system called GraphX [29] has been developed to allow multiple stages of graph-processing and pre- and post-processing to be addressed by a single framework. In their paper, the authors show GraphX to be slower than dedicated graph processing systems like GraphLab, but faster for the overall pipeline, which may include non-graph processing tasks. In Section 8.3, we compare our approach with GraphLab, PBGL, Galois and GraphChi.

Disk-based systems such as MapReduce and Hadoop can also be used to process graphs, however, the overhead of reading from and writing to disk prompted the development of more specialized graph libraries like Pregel and Giraph.

Recently, a system based on GraphLab called GraphChi [12] was developed to allow large graphs to be processed by a single-node computer by storing the graph on disk, and using a Parallel Sliding-Window approach to bring edges in memory to process them. The parallel sliding-window model and pull model implemented in GraphChi is not efficient for graph traversals, as loading the neighborhood of a single vertex requires scanning a complete memory shard. We provide a comparison of our theoretical model in Section 5.5, as well as compare our results with those of GraphChi in Section 8.10. A similar library, X-Stream [30], uses an edge-centric computational model, representing the data as a stream of edges. However, this too is limited to a single shared-memory node, and has an extra shuffle phase that takes $O(\frac{|E|}{|B|}log(P))$ extra time per iteration.

Pearce et al. [24] presented an asynchronous system for graph traversals for non-dynamic graphs on external and semi-external memory systems. However, it requires the vertex values to be in-memory (RAM), while the graph structure can be stored on disk. Due to this limitation, the realistic size of graphs is still limited, although to a much lesser extent than in the case of in-memory graph libraries.

In [26], Vitter proposes a theoretical model for external-memory graph searching using the technique of blocking. Their technique optimally uses disk blocks with replication of vertices on multiple blocks to provide fast performance for graph searching. However, due to replication, their algorithm is suited more for read-only graph algorithms, where the data for the vertex does not change. For supporting read-write algorithms, they need to update the multiple vertex replicas by bringing them in memory, the cost of which is analyzed in the paper. Their model uses a lower-level blocking and paging strategy than our subgraph-based paging. Further, their model loads the blocks on demand, while we use deferred asynchronous updates to allow better utilization of subgraphs that are already in memory.

5.8 Conclusion

We have presented a RAM-disk hybrid approach to processing large graphs that can scale from an Android tablet to off-the-shelf PCs to a large-scale distributed cluster with $16,000+$ cores. It uses an asynchronous push model in conjunction with subgraph-based paging and deferred updates to utilize available resources effectively, while decoupling algorithms from the underlying system. Our implementation extends an in-memory system (STAPL GL) to allow it to process out-of-core graphs, and shows improved performance over existing disk-based graph processing systems, as well as no penalty for execution when the system has sufficient RAM to store the entire graph in memory.

6. APPLICATIONS OF GRAPH PROCESSING

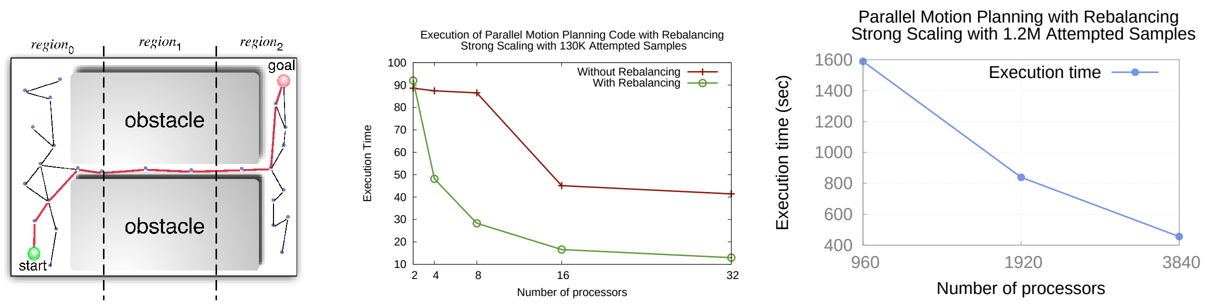
The STAPL Parallel Graph Library has been used by several researchers to implement applications in their respective domains, including Motion Planning [31], Computational Biology, Computational Geometry [32], Nuclear Physics Simulations [33, 34], and Geophysics [35]. STAPL GL's constructs and features enable application developers to write scalable and performant applications, while focusing on their problem space instead of nuances of graph processing.

In this section, we present several real-world scientific applications implemented using the STAPL Parallel Graph Library across different domains to demonstrate its suitability and general applicability for graph processing.

6.1 Motion Planning

Motion planning is the problem of finding a path for a movable object through an environment from a start to a goal configuration. Sampling-based motion planning is a probabilistic method consisting of two phases: generation and connection of samples representing valid (e.g., collision-free) points in configuration space (C-space) of the object, and querying of the roadmap for valid paths.

Jacobs et. al [31] introduced a scalable parallel application for sampling-based motion planning that subdivides the C-space into regions and constructs independent roadmaps for each region. The regions are then connected to form a single roadmap. This algorithm was implemented using STAPL GL where both the regions and roadmap are PGRAPHS. In complex environments, regions could have varying numbers of obstacles, creating regions with fewer nodes, and leading to an imbalance in computation during the connection phase. Instrumenting this real-world production application to invoke PGRAPH redistribution support on the region graph helps the application scale, as well as run faster on unbalanced inputs (Fig. 6.1) with minimal input from the application, as the application needs only provide the costs it associates with each vertex.



(a) (b) (c)

Figure 6.1: Strong scaling for a parallel motion planning application on a poorly balanced environment (a) with and without rebalancing techniques (b), at scale with rebalancing (c).

6.2 Nuclear Particle Transport

The Nuclear Particle Deterministic Transport application (PDT) accurately simulates physical systems whose evolutions depend on the transport of subatomic particles. Such simulations are extremely computationally demanding, and therefore need to rely on efficient parallel processing to obtain solutions in practical times. Parallelizing these computations is an important problem, which is targeted by the DOE's Accelerated Strategic Computing Initiative (ASCI).

One approach, the discrete-ordinates method, addresses deterministic particle transport by quantizing the simulated space into a spatial grid, and executing several *sweeps* through the grid, one for each direction of particle travel. Each sweep is a directional traversal of the spatial grid, with the actual computations being performed at the vertices of the grid. These computations factor-in energy, material properties, etc. of the simulation. The scalability and performance of this method depends on efficient parallelization of these transport sweeps. Using STAPL GL, this application was able to scale up to 384,000 cores on Sequoia, a Blue-Gene/Q machine available at the Lawrence Livermore National Laboratory, while demonstrating good parallel efficiency [34].

The core of the transport sweep is the parallel multi-traversal of a grid. Using the KLA paradigm allows the multi-traversal to scale efficiently. Further improvements in scalability can be obtained by performing a hierarchical sweep. The hierarchical sweep (pseudocode shown in Figures 6.3, 6.4, and 6.5) reads the input graph from file and creates a hierarchy on it based on the

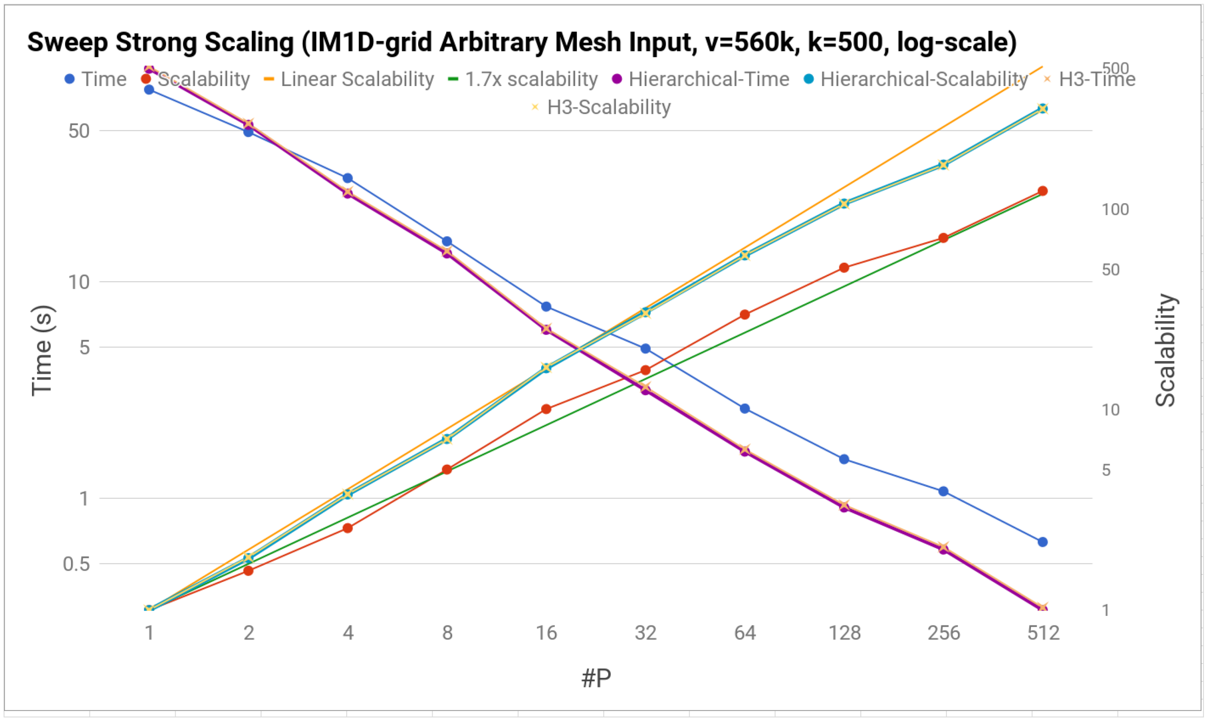


Figure 6.2: Strong scaling of parallel multi-sweep (hierarchical and non-hierarchical) on an IM1D-Grid Arbitrary Mesh Input, with 560k vertices on a Cray XK6 machine.

machine hierarchy. The Sweep itself proceeds recursively in a top-down manner – each vertex in the top-most graph of the hierarchy is visited from each sweep direction, and a recursive Sweep is spawned on its child vertices in the lower-level graphs. The results from the sweep are propagated onto the neighbors of the Supervertex.

Figure 6.2 shows the performance of running the parallel KLA Multi-Sweep (Hierarchical and non-Hierarchical) kernel, on a IM1D-Grid Arbitrary Mesh Input on a Cray XK6 machine. The hierarchical version of the sweep substantially improves upon the scalability of the non-hierarchical sweep. However, adding additional layers of hierarchy doesn't further improve the performance. This is due to the amount of communication already being small enough at $3+$ levels so as to not result in a useful tradeoff for going through the hierarchy. The aggregation of communication is $O(\frac{n}{p})$ for the second level of hierarchy, but only $O(\frac{p}{j})$ for the third level (where j is the number of cores per node). Regardless of the graph, this is typically not sufficient (since $\frac{p}{j} \ll \frac{n}{p}$) to warrant going through an extra level of the hierarchy.

```
// Initializes vertices' SWEEP-parents, SWEEP-distances, and active status.
// If the vertex is one of the sources, the distance for its traversal-ID is
// zero and it is set to active w.r.t. its own traversal-ID.
// Non-source vertices are set to inactive and their distance set to infinity.
vector<vertex_id> sweep_init_h_wf(vertex v, sources, directions, bool init_mode) {
  bool was_active = false;
  if (init_mode) {
    // For each direction, initialize the vertex's property.
    for (int i = 0; i < directions.size(); ++i) {
      auto direction = m_directions[i];
      v.property().num_incoming_edges[direction] = 0;

      for (edge e : v) {
        auto dot_product = dot_product(e.property().direction(), direction);
        if (e.target() != v.descriptor() && dot_product < 0)
            v.property().num_incoming_edges[direction]++;
      }

      if (v.property().num_incoming_edges[direction] == 0) {
        // Use the index of the direction as the SweepId.
        v.property().incoming_sweeps[i, direction] = 0;
        v.property().value[i, direction] = 0;
        was_active = true;
      }
    }
  } else {
    // Initialize the vertex's property for other sources.
    init_property_wf(m_sources, m_directions, v.property().property);
  }
  if (was_active)
    return {v.descriptor()};
}

// Indicates whether or not an edge should be visited for the given traversal.
bool filter_h_edge(traversal_id, direction, Edge e) {
  // Proceed only when the dot-product of edge direction vector and sweep
  // direction vector is greater than zero.
  if (e.cross_edge())
    return false;
  return (dot_product(e.property().get_direction(), direction) > 0);
}

// Compute new value for vertex based on incoming values, and spawn a recursive
// sweep down the hierarchy.
int update_h_property(traversal_id, direction, incoming_values,
                      vertex v, vector<GView> lower_graphs) {
  // Perform user-computation.
  auto result = user_computation(incoming_values);
  v.property().value[traversal_id, direction] = result;

  if (lower_graphs.size() > 0) {
    vector<GView> new_lower_graphs{lower_graphs[0..lower_graphs.size()-1]};
    // Perform a recursive sweep over child vertices in the lower-level graph.
    sweep_rec(v.children(lower_graph.back()), new_lower_graphs, direction, k);
  }
  return result;
}
```

Figure 6.3: Pseudocode for internal methods of the hierarchical parallel multi-sweep.

```
///////////////////////////////////////////////////////////////////////
/// Sweep Visitor.
/// Updates the target vertex with SWEEP-parent and SWEEP-distance
/// information from a particular traversal-ID, if the target vertex has
/// not been visited before, or if the target's current distance is
/// greater than the incoming distance.
///////////////////////////////////////////////////////////////////////
bool sweep_neighbor_op(vertex target, traversal_id, direction, value) {
  // If incoming value for given traversal-id, in the given direction
  // warrants updating the vertex, then add this sweep to the vertex.
  if (update_condition(traversal_id, direction, value, target.property())) {
    target.property().incoming_sweeps[traversal_id, direction] = value;
    // A vertex shouldn't be processed until the values for all incoming edges
    // for a given sweep have arrived.
    if (target.property().num_incoming_edges[direction]
        == target.property().incoming_sweeps_size[traversal_id, direction])
      return true;
  }
  //else ignore.
  return false;
}


///////////////////////////////////////////////////////////////////////
/// Sweep Work-function.
/// A vertex is visited if it is active. Active vertices update their
/// neighbors with new SWEEP distance and parent information for the given
/// traversal-ID.
/// Returns true if vertex was active, false otherwise.
///////////////////////////////////////////////////////////////////////
bool sweep_vertex_op(vertex v, GraphVisitor graph_visitor,
                     vector<GView> lower_graphs) {
  size_t traversal_id;
  bool was_active = false;

  // For each incoming sweep, compute the new value and propagate.
  for (auto element : v.property().incoming_sweeps()) {
    was_active = true;
    traversal_id = element.first;
    for (auto directional_element : element.second) {
      auto direction = directional_element.first;
      // Compute and update the vertex's property for this sweep.
      // Also return the value to propagate for this sweep.
      auto new_sweep_value =
        update_h_property(traversal_id, direction, directional_element.second,
                          v.property(), lower_graphs);
      sweep_neighbor_op uf(traversal_id, direction, new_sweep_value);
      // Propagate new sweep value to valid neighbors.
      for (edge e : v)
        if (filter_h_edge(traversal_id, direction, e))
          graph_visitor.visit(e.target(), uf);
    }
  }
  // Reset incoming sweep buffer.
  v.property().incoming_sweeps().clear();

  return was_active;
}
```

Figure 6.4: Pseudocode for vertex- and neighbor- operators for the hierarchical parallel multi-sweep.

```
void sweep_rec(GView top_graph, vector<GView> lower_graphs, vector<int> directions,
               int k)
{ graph_paradigm(sweep_vertex_op(lower_graphs), sweep_neighbor_op(), top_graph, k); }


//////////////////////////////////////////////////////////////////////
/// A Parallel Multi-Source Sweep algorithm.
///
/// Performs hierarchical multi-source sweeps on the input graph_views, starting
/// from the given directionr. The sweep from each source happens simultaneously,
/// and is identified by a unqiue sweep-Id, which is the descriptor of the source
/// vertex of that sweep. The vertex-properties are initialized by the init_prop_wf.
/// The sweep visits each vertex, and computes its new property based on the
/// provided update_prop_wf. The sweep then visits the vertex's neighbors that pass
/// the user-provided edge-flitering based on the filter_edge_wf. If the target
/// neighbor vertex should be updated based on the provided update_cond, it
/// continues the sweep, otherwise the neighbor is not updated.
/// @param graphs The vector of @ref graph_views representing a graph hierarchy.
/// graphs[0] should represent the lowest level of the hierarchy, with
/// subsequent indices building upon the i-1th graph.
/// @param k The maximum amount of asynchrony allowed in each phase.
/// 0 <= k <= inf.
/// k == 0 implies level-synchronous SWEEP.
/// k >= D implies fully asynchronous SWEEP (D is diameter of graph).
//////////////////////////////////////////////////////////////////////
void sweep(vector<GView> graphs, vector<int> directions, int k) {
  for (int i = graphs.size()-1; i >= 0; --i) {
    // Initialize the vertices and sources. Sources are direction-dependent --
    // a vertex is a source for directions where it has no incoming edges.
    auto sources = map_reduce(sweep_init_h_wf({}, directions, true),
                              join_sources_wf(), graphs[i]);
    if (sources.empty())
      return 0;
    map_func(sweep_init_h_wf(sources, directions, false), graphs[i]);
  }

  // Call actual sweep.
  sweep_rec(graphs.back(), graphs[0..graphs.size()-1], directions, k);
}

/// Driver for Sweep.
void sweep_driver(string filename) {
  using graph_t = digraph<super_vertex_property<sweep_vertex_property>,
                          super_edge_property<sweep_edge_property>>;
  using graph_view_t = graph_view<graph_t>;
  // Read graph from file.
  graph_view_t input_graph = sweep_graph_reader<graph_t>(filename);
  // Create a hierarchy.
  vector<graph_view_t> hierarchy = create_machine_hierarchy(input_graph);
  // Enumerate (x,y,z) directions for 3-D Sweep:
  vector<dimensions_t> directions = {{1,1,1}, {1,1,-1}, {1,-1,1},
                                     {1,-1,-1}, {-1,1,1}, {-1,1,-1},
                                     {-1,-1,1}, {-1,-1,-1}};
  // KLA SWEEP:
  sweep(hierarchy, directions, user_computation(), k);
}
```

Figure 6.5: Pseudocode for the hierarchical parallel multi-sweep, along with the driver for the sweep.

6.3 Seismic Ray-Tracing

Ray Tracing is a method for computing the propagation of waves or particles through a system comprised of various regions of varying characteristics, which may affect the wave's or particle's velocity through that medium, its absorption or its reflection. In Ray Tracing, the wave is modeled as a large number of *rays*, which are iteratively advanced through the system in discrete amounts. The properties of the ray, such as speed and direction, are re-computed in each iteration based on the given medium and the ray's existing characteristics. This application applies Ray Tracing to understand and model seismic wave propagation.

The application uses the wavefront construction method (WFC), which considers an entire wavefront instead of individual rays, and can adaptively control the number of rays forming the wavefront through interpolation. This improves the computational efficiency, as well as accuracy of the simulation. The wavefront is represented with a PGRAPH, while the collection of rays is stored in a map. The application iteratively steps over each ray in parallel to propagate the wavefront, computing the properties of each of its ray-segments. Experimental results from running this application on an IBM p575 cluster showed scalable parallel performance [35].

6.4 Conclusion

We presented real-world scientific applications from three different domains that use the STAPL Parallel Graph Library as their core to express computations. STAPL GL allows these applications to scale to a large number of cores with improved performance, while still maintaining expressibility.

7. IMPLEMENTATION*

The STAPL Parallel Graph Library comprises of a customizable distributed graph container (pGRAPH), and a collection of commonly used parallel graph algorithms. The library uses graph paradigms that separate algorithm design from its execution. It supports three graph processing algorithmic paradigms, k-level-asynchronous, hierarchical, and out-of-core, allowing algorithms written for STAPL GL to execute efficiently and scalably on a variety of machines and input graphs. In this section, we describe the implementation details of the pGRAPH container, as well as the various paradigms required to execute algorithms efficiently.

7.1 The pGRAPH pCONTAINER

The pGRAPH is built using the STAPL pCONTAINER framework (PCF), which provides base classes that handle issues dealing with data distribution and parallelism and allows the design of the pGRAPH pCONTAINER to focus on graph-specific concerns. The pGRAPH pCONTAINER consists of a set of base containers (bCONTAINERS) and the infrastructure to make them work together in parallel. For the pGRAPH, a bCONTAINER is a base graph data structure that exports the pGRAPH's interface. The bCONTAINER has three layers: the representation of the graph, the graph storage, and the underlying storage. The graph storage is tied to the representation, exporting an interface that allows the representation to work with the underlying storage. It provides the policies for the type of underlying storage used by the graph (e.g., vector, hash map, map) for vertices and edges. It also specifies the type for a vertex and type for an edge, along with how properties are stored on these. The underlying storage may be a sequential container unaware of parallelism that is used by the graph to store vertices and edges, or possibly another pCONTAINER.

The PCF provides a shared-object view [36] that allows users to address any element globally. pGRAPH users interact with the container by method invocation, which the framework forwards to

* The base framework presented in this chapter is reprinted from *proceedings of the 25th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, "The STAPL Parallel Graph Library," 2012, pp. 46–60, Harshvardhan, A. Fidel, N. M. Amato, and L. Rauchwerger, ©2012 with permission from Springer.

the location where the needed graph elements reside. Fig. 7.1 shows the internal base-class implementation for apply_async, which provides an example of address resolution for graph elements using asynchronous communication. The apply_async method is provided by the PCF for applying a higher-order function object on an element of the container. This may be used to implement methods such as add_edge and set_vertex_property for the PGRAPH. Internally, this forwarding is supported by a distributed directory service – which is contained within the PGRAPH – that provides a two-level lookup of the requested vertex's location. This is described below.

Shared-Object View Provided by Distributed Directory. The PGRAPH is a dynamic container, where vertices may be added and removed, and so vertex IDs need not be contiguous or even ordered. The PGRAPH uses a distributed directory to provide a shared-object view to users and abstract them from dealing with the details of distribution. While a distributed directory can increase access costs, other solutions such as centralized models (e.g., the master-slave model employed by Pregel [9]) which store the entire directory information in a single location, or replicated directory on all locations, may not scale to large systems.

In this two-level distributed directory scheme, every vertex has a *home location* associated with it, which may not be the location of the vertex, but is rather the location that stores information about the vertex's locality. It is calculated using simple closed-form solution (a hash of the vertex's descriptor), so any requesting location knows quickly and precisely where to send the request.

In this mechanism, the PGRAPH first checks if the graph vertex is local, and if so, then services the request immediately. If the vertex is not found locally, the local directory computes the *home location* of that vertex and forwards the request there. The *home location* is responsible for knowing the exact location of the vertex. In some cases, the *home location* may own the vertex itself, at which point, the requested action is performed on the vertex. However, in the case that it does not, the request is forwarded to the location that owns the vertex, where the request is serviced. As shown by Tanase et. al. [36], address resolution using asynchronous forwarding provides improved performance over a directory that determines the element's location using synchronous communication.

```
void base::apply_async(vertex_descriptor s,
                       Functor f)
if base_container.contains(s)
  base_container.apply(s, f)
else
  home = home_location(s) //hash-based lookup
  if my_location == home
    owner = directory.lookup(s)
    async_rmi (owner, apply_async(s, f))
  else
    async_rmi (home, apply_async(s, f))
```

Figure 7.1: Internal base-class implementation of `apply_async` method illustrating address resolution.

```
// asynchronous migration and redistribution:
g.migrate(vertex, location)
g.redistribute(cost_map, action_function=no_op)
```

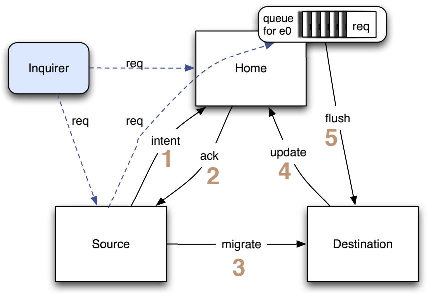


Figure 7.2: Asynchronous migration protocol for PGRAPH.

Vertex Migration. The STAPL GL provides the novel ability to migrate vertices asynchronously between locations during the execution of the program. An important property of the migration protocol is that it ensures that STAPL GL algorithms can be oblivious to the data distribution and also to any migration occurring during the execution of an algorithm.

The protocol for migration of graph elements implemented by the PGRAPH is inspired by directory-based cache coherence techniques [37] and is described in Fig. 7.2. When processing an element-migration request from a source location to a destination location, the source first informs the home location of its intent to migrate (1). The home location, upon receiving this request, marks the element as in the process of migration and creates a queue for all requests addressed to that element. It then sends an acknowledgement (2) to the source location allowing the source to then proceed to migrate the element data to the destination (3). When the destination receives the element's data, it stores it and informs the home location (4) to update its metadata to record that the destination is now the owner of that element. Finally, the home location updates its metadata and forwards all pending requests for that element to the destination location (5). If at any point during migration a location requests access to the element that is currently being migrated, the requests are forwarded to the home location for that element, where they are enqueued. The queue is flushed at the end of migration and requests are forwarded to the new location of the element.

Redistribution. As users of STAPL GL generally may be unaware of locality, STAPL GL provides a convenient way to rebalance a PGRAPH.

Redistribution of a PGRAPH requires some process for determining the new distribution. This can be user provided or it can be computed based on some cost function. For many graph-based scientific applications, a cost function (cost map) can be determined representing the expected computational costs associated with vertices and edges. In STAPL GL, such cost maps can be user provided, or if no additional information is available, uniform costs can be assumed for all elements. Given a cost map, a new partition that attempts to address the imbalance can be computed by an STAPL GL graph partitioning algorithm. Given a desired partition, each location computes the vertices that need to be migrated to itself from other locations and invokes a migrate call on those vertices. Internally, the asynchronous directory forwards the migration request to the correct location where the element is located and initiates the migration of that vertex. Fig. 7.2 illustrates how redistribution is invoked, and the protocol used in STAPL GL. STAPL GL allows application programmers to optionally provide callback functions that are invoked along with each migration call on the corresponding element to allow any action that needs to be performed during the process of migrating a single element, such as updating auxiliary data structures.

7.2 Graph Traversal Strategies

Traversal Strategies in STAPL GL are implemented using algorithmic patterns provided by STAPL that execute higher-order functions (work-functions) on elements of a view. To express a parallel graph algorithm, users choose a suitable traversal strategy and provide two operators that describe the computation in a fine-grained manner. For example, users can implement a breadth-first traversal (BFS) by providing generic BFS vertex and visitor operators (Figure 7.3(b, c)) to the traversal strategy. Both operators are generic and oblivious to the strategy, allowing separation of algorithm from performance by choosing the appropriate strategy based on the input and system.

Figure 7.3 shows the pseudo-code to demonstrate how the level-synchronous, asynchronous, and KLA versions of BFS differ. From the user's perspective, the BFS vertex and visit operators for the algorithm do not change (Figure 7.3(b,c)), and neither does the driver of the algorithm (Figure 7.3(a)). What does change, however, is the VISIT function (Figure 7.3(d-f)), which determines the conditions to allow the computation to proceed on the target vertex being visited.

A KLA BFS results by providing the generic BFS operators (Figure 7.3(b,c)) to the traversal strategy, using the KLA VISIT function (Figure 7.3(e)). Lines 3-7 in Figure 7.3(a) can be replaced by a call to the KLA traversal strategy, which has the same effect. The vertex-operator returns true if it is active for a vertex, and false otherwise. The algorithm terminates when all vertices are inactive (all invocations of vertex-operators return false). The requirements for the KLA BFS visitor operator are the same as for the asychronous BFS visitor operator – it cannot rely on the order in which the vertex is visited and must be resilient to multiple visits.

Hierarchical Paradigm in SGL. The hierarchical graph paradigm allows the execution of vertex-centric algorithms on hierarchical graphs, as it is a drop-in replacement for the standard graph paradigm, as seen in Figure 2.1(c), such that existing vertex and neighbor operators need not be modified to take advantage of the hierarchy, or even be aware of it. Different graph algorithms can take advantage of the hierarchical paradigm simply by swapping the call to *kla_paradigm* with *hierarchical_paradigm* and providing it the hierarchical graph. This swap is encapsulated in STAPL GL's *execute* method and the *execution policy*.

Support for Out-of-Core Processing. To support out-of-core (disk-based) graph processing, we extend STAPL GL, which supports in-memory graph computations, to allow storing the graph structure to disk. The graph structure stored in the PGRAPH distributed container consists of one or more base containers per location. The subgraphs are stored in the base container, which we allow to be stored in-memory or on-disk. The base container also contains an in-memory hubs-cache, if the optimization is enabled. To allow for loading and storing subgraphs to disk, we extended the PGRAPH base containers to provide serializing and deserializing capabilities, and implemented the ability to read and write shards. The asynchronous forwarding of updates is processed by the two-level distributed directory that maps vertices to their home-locations.

The API of the STAPL GL does not change, as our technique is transparent to the user. This also allowed us to use STAPL GL algorithms *without modification*.

```
void BFS(Graph graph, vertex source, int k)
  source.color = GREY;
  GraphView active_vertices(graph, source)
  while(active_vertices.size() != 0)
    active_vertices
      = reduce(map(bfs_op, active_vertices, k), logical_or())
    global_fence();
```

(a)

```
bool bfs_op(vertex v)
  if (v.color == GREY)          // Active if GREY
    v.color = BLACK
    map_func (Visit(bfs_visitor_op(v.dist+1)), v.neighbors())    // Visit neighbors
    return true                 // vertex was Active
  else  return false            // vertex was InActive
```

(b)

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

(c)

```
// Level-Sync Visit -- don't spawn further tasks
Visit(visitor vis, vertex u, int k_current)
  vis(u)
```

(d)

```
// KLA Visit -- recursively spawn tasks to depth k
Visit(visitor vis, vertex u, int k_current)
  if (vis(u))                // if u was updated
    if (k_current % k != 0) // iff inside async section
      bfs_op(u)     // recursively call work-function on u
```

(e)

```
// Async Visit -- recursively spawn tasks
Visit(visitor vis, vertex u, int k_current)
  if (vis(u))              // if u was updated
    bfs_op(u)       // recursively call work-function on u
```

(f)

Figure 7.3: (a) The fine-grained BFS algorithm, with (b) BFS vertex operator and (c) BFS visitor operator. Also shows how changing the visit-functions between level-sync (d), async (f) and k-level-async (e) visit functions changes the behavior of the algorithm from async to KLA to level-sync, using BFS as an example algorithm.

8. RESULTS*

This section presents experimental results studying the scalability and performance of STAPL GL and its parallel graph algorithms, and compares them to existing graph libraries and implementations. We also study the performance of various input graphs on different platforms, and demonstrate the need for a single unifying paradigm, which can deliver the best performance for various combinations of platforms and input graphs.

8.1 Experimental Setup and Input Graphs

Experimental studies were performed on HOPPER – a Cray XE6 machine at the National Energy Research Scientific Computing Center (NERSC) with 153,216 total cores, of which 98,304 cores were available to us. Results reported were averaged over multiple runs to obtain a suitable confidence interval. The compiler used for the experiments was gcc (version 4.7.2) with the `-O3` optimization flag. In all experiments, a location represents a single core, and the terms may be used interchangeably.

KLA was evaluated on representative graphs from different domains, ranging in size from a few million to tens of billions of vertices and edges (Table 8.1). These include the Graph 500 benchmark input [20], Google and Twitter web-graphs, various road-networks (TX, PA, US, EU and Synthetic) [38, 39], and geometric graphs (Delaunay meshes, geometric meshes and torii) [38, 39]. Input graphs use the default distribution provided by the input files.

* Some results presented in this chapter are reprinted from: "KLA: A new algorithmic paradigm for parallel graph computations," Harshvardhan, A. Fidel, N. M. Amato, and L. Rauchwerger ©2014 IEEE; "An algorithmic approach to communication reduction in parallel graph algorithms," Harshvardhan, A. Fidel, N. M. Amato, and L. Rauchwerger ©2015 IEEE; "A hybrid approach to processing big data graphs on memory-restricted systems," Harshvardhan, B. West, A. Fidel, N. M. Amato, and L. Rauchwerger ©2015 IEEE; "The STAPL Parallel Graph Library," Harshvardhan, A. Fidel, N. M. Amato, and L. Rauchwerger; with respective permissions.

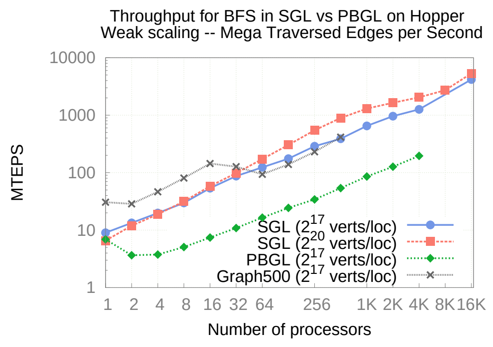
Name	Description	V, E	Source
G500-N	Graph500 Benchmark Input	2^N, 16×2^N	Graph 500
Google	Google Web-Graph	916K, 5.1M	SNAP
Twitter	Twitter Web-Graph	62M, 1.2B	SNAP
RoadNet-TX,PA	Road-networks (TX, PA)	(1.4M,3.8M), (1M,3M)	SNAP
RoadNet-US,EU	Road-networks (US, EU)	(24M,58M), (50M,54M)	Dimacs-9
RoadNet-Synthetic	Synthetic Road-network	9.63B, 10.2B	Generated
Delaunay	Delaunay Triangulation Graph	16M, 50M	Dimacs-10
RGG	Geometric Graph	16M, 132M	Dimacs-10
Torus	Toroid Mesh	$2.25M \times p$, $9M \times p$	Generated

Table 8.1: Characteristics of input graphs.

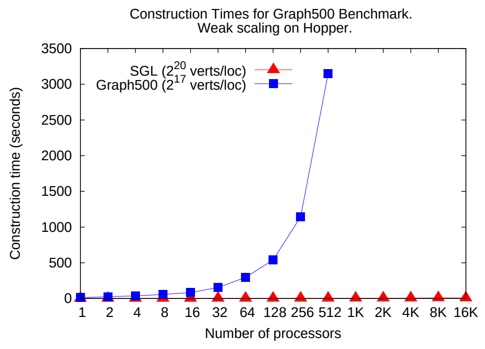
8.2 Graph 500 Benchmark

To test the performance and scalability of STAPL GL and compare it with existing libraries, we implemented the Graph 500 benchmark [20], which simulates typical graph-processing workloads on internet-scale webgraphs, for STAPL GL. Ours was a naive implementation of BFS, using the level-sync BFS, for a fair comparison with the benchmark reference implementation and other libraries. The traversal strategy handles details of aggregation and communication, thus giving higher performance even with simple algorithmic implementations. We compare our implementation with PBGL, MTGL and the Graph500 reference implementation in MPI, as well as with various other distributed and shared-memory graph libraries.

Fig. 8.1(a) shows a weak-scaling plot comparing the scalability of the Graph500 benchmark implementation, PBGL and STAPL GL. Both the benchmark and PBGL suffered from memory bottlenecks and poor scalability. While the Graph500 reference implementation was able to construct the graph, it crashed during the execution of the algorithm on 1,024 cores and PBGL was unable to run the algorithm beyond 4,096 cores. The time to build the graph from the input specification is shown in (Fig. 8.1(b)). The benchmark uses synchronous operations to globally shuffle the input edges to maintain contiguous access through the edge-list, whereas STAPL GL is able to asynchronously add edges. We also observed that STAPL GL scaled better than both the benchmark and PBGL. The benchmark generates a large number of small messages while executing the



(a)



(b)

Figure 8.1: Comparison of MTEPS (a) and construction times (b) for PGRAPH, PBGL and Graph 500 ref. implementation. Note log-scale y-axis for (a)

algorithm, which overloads the MPI buffers of the machine. STAPL GL messages are aggregated by combining messages being sent to the same location, as well as buffering them to send fewer messages of larger size, which helps achieve better performance – as is more evident when going off-node (24 cores). This performance trend is also observed when executing other algorithms, such as PageRank (Google's web-ranking algorithm), connected-components and k-core, on the benchmark input graph.

8.3 Comparison with in-memory libraries

We study our in-memory performance and compare it with existing in-memory graph libraries using the Graph500 benchmark. Our approach is the only one among the libraries that can perform out-of-core computations, however, our comparisons with both shared-memory and distributed-memory systems still show competitive in-memory performance with no penalty paid for disk-access when sufficient RAM is available.

Figure 8.2 shows the execution times of various libraries normalized to the time taken by our approach implemented in the STAPL GL. We note that the STAPL GL performs comparably to shared-memory libraries, and scales better and is faster than other distributed-memory graph libraries such as the Parallel Boost Graph Library (PBGL) [7] and GraphLab/PowerGraph [13]. While our approach is $2\times$ slower than the Graph500 benchmark implementation [20] up to 16 cores, once off-core, our approach scales better. Our approach is initially slower as the benchmark implementation uses a raw array of integers to store the graph, and as such does not have the overhead of a generic library.

As shown in Figure 8.2, and in Figure 8.3 on up to $16,384$ cores, our approach outperforms other libraries in distributed memory. PBGL uses ghost-vertices, which limits its scalability as the ghost cells need to be kept synchronized. GraphLab uses a pull-model, where each vertex synchronously reads the values from its neighbors, which does not allow deferred updates as in our asynchronous push model, affecting performance. In contrast, using an asynchronous push-model can scale effectively to higher core-counts.

Finally, our approach is also seen to compare favorably with shared-memory libraries. Green-Marl [40] is a domain specific language (DSL) for graph computations in which the DSL compiler is able to avoid the overhead of a generic library and is around $1.5 - 2\times$ faster than our approach in shared memory. However, as the provided implementation is limited to shared-memory only, it cannot run beyond 16 cores (1 node) on our Cray XE6m. Galois [10] is another shared-memory graph library, that has performance comparable to our approach.

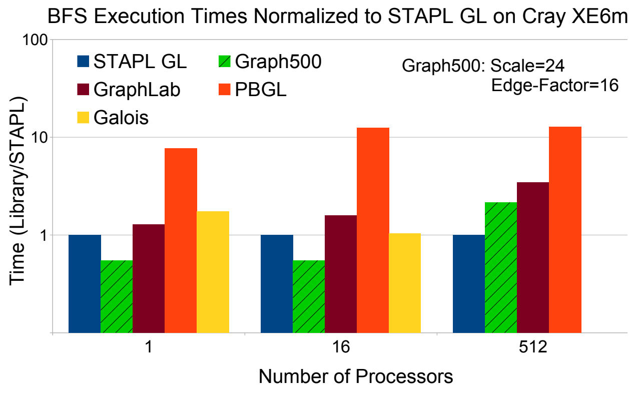


Figure 8.2: Execution times of Graph500 on various graph libraries normalized to STAPL GL on CRAY XE6m. Shared-memory graph libraries (Galois, GreenMarl) are shown to 16 cores.

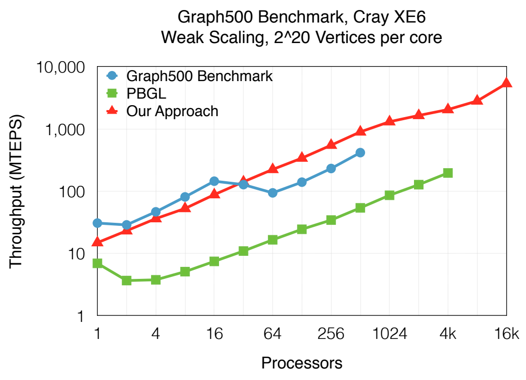
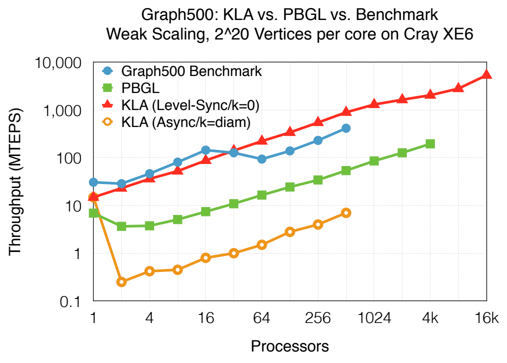
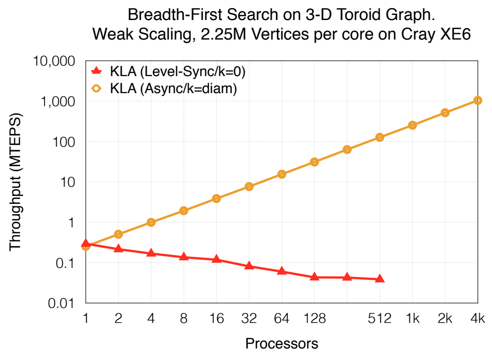


Figure 8.3: Weak scaling of STAPL GL on the Graph500 benchmark input on CRAY XE6 with 2^{20} vertices per core. Y-axis shows throughput in Mega-Traversed Edges per Second (MTEPS) in log-scale.



(a)



(b)

Figure 8.4: Scalability (Throughput) of BFS on (a) Graph500 benchmark (max. $|V|$=16B, $|E|$=256B) and (b) torus (max. $|V|$=9.2B, $|E|$=37B).

8.4 KLA Experiments

In this section, we demonstrate the need for a single unifying paradigm with the ability to parametrically control asynchrony. We evaluate the performance of KLA as compared with traditional level synchronous and asynchronous paradigms for a variety of algorithms on various real-world and synthetic graphs.

8.4.1 Level-Synchronous vs. Asynchronous BFS

To demonstrate the performance differential between level-synchronous and asynchronous paradigms, we compared our implementation of an async BFS to its level-sync variant on a torus graph and the Graph 500 Benchmark graph (shown in Figure 8.15). These represent two extreme cases for long and short diameter graphs. We use $k = 1$ to emulate level-synchronous behavior and $k = n$ to emulate asynchronous behavior for KLA BFS. In both cases, the algorithm remains the same and only the value of k need vary to suit the input.

The Graph 500 input graph has a short diameter (<16) and vertices with very high out-degrees. For this graph, async BFS performs poorly due to the large number of messages flooding the system and the large amount of wasted work for revisited vertices. The level-sync BFS performs well in this case due to the input graph's low diameter, which implies fewer global synchronization. The torus graph, on the other hand, represents a worst-case scenario for parallel BFS scalability – the algorithm is essentially serialized due to the topology of the torus, which limits the available parallelism. In this scenario, we observed that async BFS performed much better than level-sync BFS, due to the absence of synchronization-points (Figure 8.15 (b)). These experiments demonstrate that the topology of the graph can significantly affect the performance characteristics of algorithms, and pairing the correct paradigm to the topology can result in a significant performance benefit.

8.5 Evaluation of KLA BFS

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

Graph Structure and k_{opt}. To investigate the correlation between the structure of a graph and the choice of k for the KLA algorithm, we ran the KLA BFS on a graph while deforming its shape to vary the diameter from very large (circular chain, $d \approx 128k$) to very small (random graph, $d \approx 8$). We obtained graphs with different diameters by allowing any given vertex to randomly select and connect only to its $\pm m$-closest neighboring vertices, varying m from 2 (circular chain)

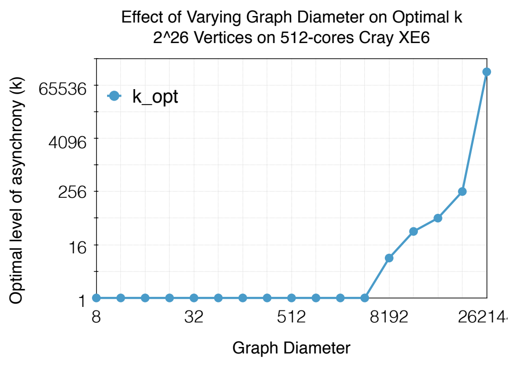
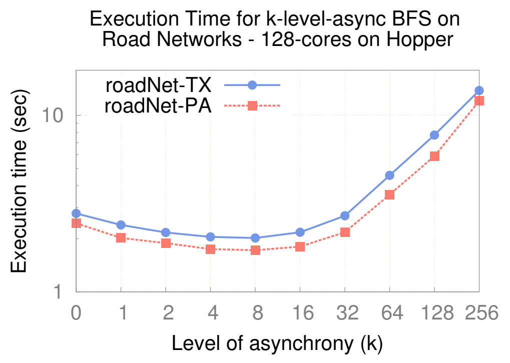


Figure 8.5: Modeling KLA: Study of effects of graph diameter on the optimal k parameter.

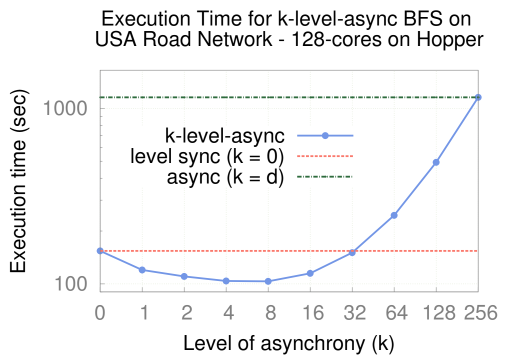
to n (random network). This is similar to the approach described by Watts and Strogatz [41]. We ran KLA BFS on each deformation for varying values of k, and observed which k was best suited for the given diameter. Figure 8.5 shows the value of k that gives the best performance for input graphs of varying diameter. This indicates that while a value of $k = 1$ better suits small-diameter graphs, there are also classes of graphs with intermediate diameters for which k_{opt} is between 1 and d. Road network graphs, analyzed in the next section, are one such class of graphs, along with scientific meshes. It is for such graphs that KLA provides improved performance, while maintaining the performance for other inputs and providing a single unifying algorithmic interface.

Road Network Graphs. KLA BFS can be used to accelerate finding routes on a road-map and navigation, as it is well suited to road network graphs due to their relatively long diameters. We obtained road networks for the state of Texas and Pennsylvania (Figure 8.6(a)), and the entire U.S.A. (Figure 8.6(b)). For each of these graphs, the level-synchronous variant, with $k = 1$ proved expensive due to high number of synchronizations. However, owing to the amount of re-done work and the number of messages generated, the purely asynchronous version was also slow. In fact, the best performance was obtained with $1 < k < d$, where the computation was $\approx$27-33% faster than the fastest traditional paradigm for all three inputs.

Additionally, we generated a synthetic road network by stitching together multiple copies of the European road network, which provided us an input with 9.63 billion vertices and 10.2 billion edges. This input was large enough to allow for a meaningful strong-scaling study to 10^5 cores.



(a)



(b)

Figure 8.6: Performance of KLA BFS on road networks for (a) TX, PA and (b) USA on HOPPER.

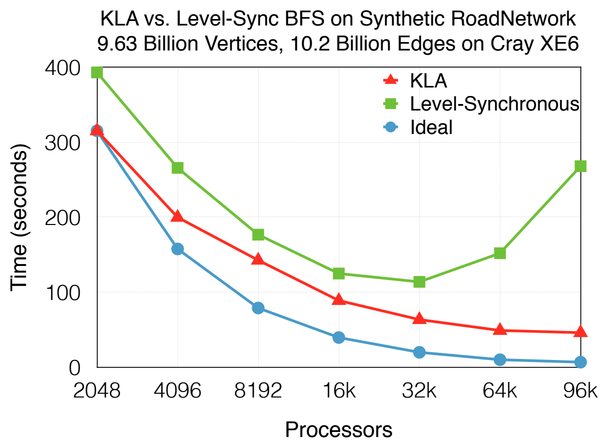


Figure 8.7: Performance of KLA vs. Level-Sync BFS on synthetic road network, 98,304 cores on HOPPER.

Figure 8.7 compares the scalability of KLA BFS with level-synchronous BFS on this road network. The level-synchronous BFS scales to 32,768 cores, but not beyond. The KLA BFS is able to scale better until 98,304 cores (the maximum available cores). It yields faster running times due to better balancing synchronization costs, which become increasingly expensive for large core counts, with wasted work. This experiment demonstrates that traditional paradigms do not always scale to a large number of cores, where KLA can improve scalability and performance significantly.

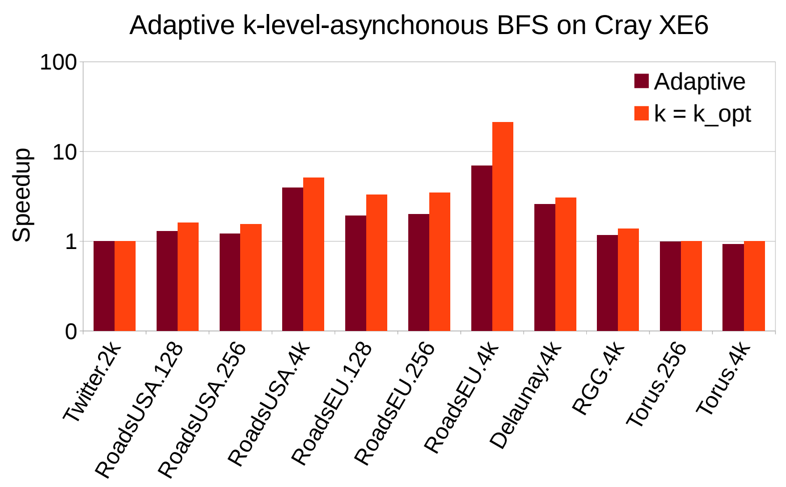


Figure 8.8: KLA BFS speedup with k_{opt} and Adaptive KLA on various graphs on HOPPER.

8.6 Evaluation of Adaptive KLA BFS

For many applications, the optimal value of k may not be known for a particular input graph. Our Adaptive KLA technique provides improved execution times adaptively based on the graph's topology when users do not have sufficient information about the input graph.

We ran the Adaptive KLA BFS on input graphs from various domains, including road networks (Europe, USA), web graphs (Twitter) and geometric graphs (Delaunay, Random Geometric Graphs (RGG), Torus). We compared the speedup obtained using KLA BFS with k_{opt} to the Adaptive KLA algorithm for each graph. We also ran the level synchronous and asynchronous BFS on these graphs, and used the faster version's running-time as a baseline for calculating the speedup of both KLA BFS and Adaptive KLA BFS. Figure 8.8 shows the speedup obtained for various graphs over the best traditional paradigm. We observe that KLA with optimal k (k_{opt}) can provide improved performance over traditional paradigms for some input graphs. Due to the small girth of these graphs, not all processors are active in each iteration (this is especially true for higher processor counts) and those processors end up waiting at the global synchronizations in the level-synchronous approach. In contrast, the asynchronous approach may re-visit many vertices, increasing the run-time. KLA allows the traversal to proceed enough so as to recruit more processors in the iteration and yet controls the depth of asynchrony to avoid too many redundant visits, allowing for better scalability and performance. We also observe that Adaptive KLA can improve the performance, although the overhead of adaptivity (as it needs a few iterations to converge) and monitoring the iterations (for wasted work, etc.) keeps it from achieving the same level of performance as KLA with k_{opt}.

8.7 Evaluation of Other Types of Algorithms

In this section, we evaluate the performance of three important graph mining and graph analytics algorithms: k-core, PageRank and SSSP. We also compare the Δ-stepping SSSP algorithm with its KLA variant. We run the three algorithms on a variety of input graphs from different domains (road networks, scientific and geometric meshes, and web-graphs) and present their speedup (relative to level-synchronous versions in case of k-core and PageRank, relative to non-KLA Δ-stepping for SSSP) in Figure 8.9.

k-core. For k-core (Figure 8.9), we expect the fully asynchronous setting for k to perform fastest. While this is the case for most inputs, graphs with high out-degrees can cause a large number of messages to flood the network, leading to network congestion. In such cases, using the level-sync setting ($k = 0$) provides the best performance, as is observed for the Google web-graph. The benefit provided by KLA for k-core is the ease of algorithm tunability to obtain the best performance for any given input at runtime.

PageRank. PageRank incurs a penalty of buffering for increased asynchrony. However, as the cost of buffering was much lower than the cost for communication and global synchronization on our system, PageRank also saw good performance gains from the KLA paradigm (Figure 8.9).

Δ-stepping. We also compare an implementation of Δ-stepping SSSP in KLA with non-KLA Δ-stepping SSSP in Figure 8.9. The results presented show the speedup of the KLA version over the non-KLA version for the best values of Δ and k. We observe from these experiments that, as expected, both variants of Δ-stepping exhibit similar performance for different input graphs, as the KLA paradigm is a generalization of the Δ-stepping algorithm [2].

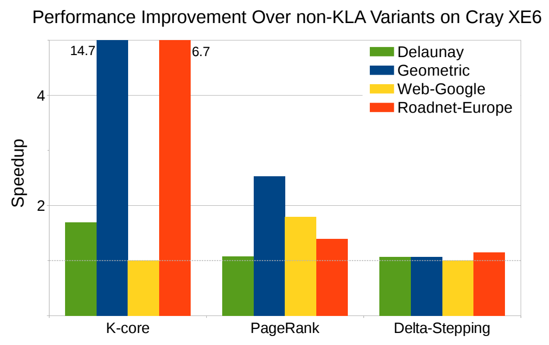


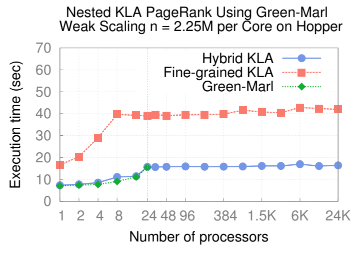
Figure 8.9: Evaluation of k-core PageRank and SSSP on 16,384 cores on HOPPER.

8.8 KLA with Other Graph Frameworks

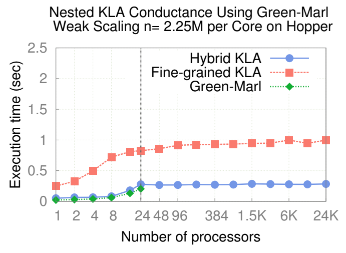
KLA with Green-Marl. KLA algorithms may also be called recursively in nested sections. We describe such computations using a tuple of k values. Each value in the tuple represents the k for each algorithm being executed in that level of nesting. The arity of the tuple denotes the number of levels of nesting in the algorithm. If the algorithm in the nested section is sequential or non-KLA, the k value for it is ignored ($k = x$). For example, using a shared-memory library for intra-node computation and KLA for inter-node computation, the tuple would be $\langle x, k_1 \rangle$. Figure 8.10 demonstrates the benefit of using nested-parallel execution in the Hierarchical Traversal Strategy paradigm. It takes advantage of an optimized shared-memory library (Green-Marl) for processing partitions of the graph on a shared-memory node and uses KLA version of the algorithm on the upper-level to extend it across distributed nodes in the machine.

In our experiments, we ran two different algorithms – PageRank and cut-conductance – on a mesh graph using the Green-Marl [40] implementation within the node and KLA off-node, and compared them with their single-level KLA implementations in STAPL GL.

KLA in Galois. In addition to the implementation provided for STAPL GL, we illustrate the portability of the KLA technique by applying it within the Galois [10] parallel framework. Galois' execution model relies on shared-memory worklists that determine task scheduling of algorithms. A suite of worklists are provided by the library, including a bulk-synchronous worklist that con-



(a)



(b)

Figure 8.10: Nested KLA with Green-Marl for (a) PageRank and (b) conductance, up to 24,000 cores.

tains two queues and processes them one after another, and an asynchronous worklist that contains several queues that may be processed in any order, but are prioritized by some metric (e.g. vertex-level).

We modified Galois' bulk synchronous worklist to process vertices in the same KLA-SS asynchronously and to only synchronize after k levels of asynchrony, effectively creating a KLA worklist for Galois. Figure 8.11 shows the results of executing KLA BFS in Galois for various input graphs. As with KLA in STAPL GL, we see improvements for input graphs with long diameters, whereas short diameter graphs perform best with level-synchronous execution. In certain cases, the KLA worklist is able to achieve more than 90% speedup over the best of the level-synchronous and asynchronous worklists.

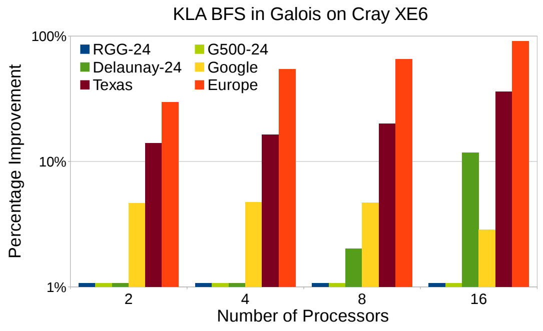


Figure 8.11: Speedup of KLA BFS in Galois on various graphs relative to the fastest version of Galois (bulk synchronous or asynchronous) on HOPPER.

8.9 Hierarchical Communication Reduction

Having extablished a base-line performance of our library's non-hierarchical mode, in this section, we evaluate the performance of our hierarchical approach as compared with the traditional (flat) approach for a set of important graph mining and graph analytics algorithms. We also compare our base-line performance with other graph libraries.

8.9.1 Improving Work Imbalance

As the number of processors increases, the graph partition can become imbalanced in the amount of edges each processor has to process. This is further aggravated by scale-free graphs where the hub-size also increases drastically with the size of the graph. This results in the processor storing a high-degree vertex performing more work processing its outgoing edges, which negatively affects scalability and performance. Our hierarchical approach alleviates this by reducing the number of inter-partition edges for each vertex to a single super-edge, and distributing the work of applying updates more evenly as the updates are now applied on the target location instead of the high-degree source.

We show the effect of our hierarchical approach on the work imbalance across processors for different input sizes of the Graph500 input graph at varying processor-counts in Figure 8.12. The flat partitioning has severe work imbalance which gets worse as the problem and machine size is scaled. The locality-based hierarchical approach is able to reduce this imbalance significantly, while also reducing the rate at which the imbalance grows (demonstrated by the diverging trend-lines). The locality-based hierarchical approach in combination with the hubs-based hierarchy further reduces the imbalance significantly and slows down the growth even more. This, as we will observe in the next section, improves scalability and performance through a better load-balanced execution, even while being decoupled from the algorithm.

8.9.2 Applications

In this section, we first demonstrate the effectiveness of our approach at scale ($12,000+$ and $130,000+$ cores on Figures 8.13, 8.14) on two different large-scale machines, and then dive down

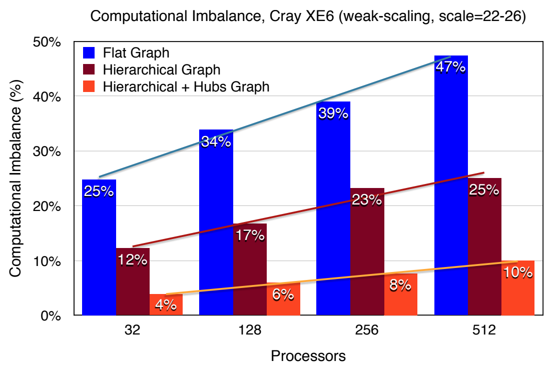
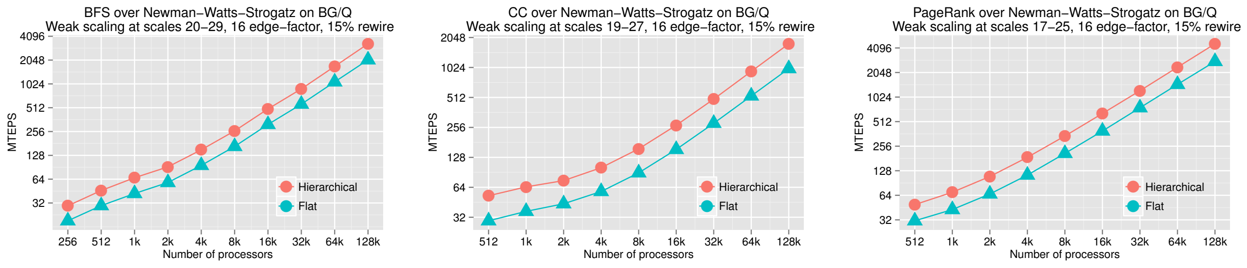


Figure 8.12: Comparison of work-imbalance (number of edges) for flat and hierarchical graphs for Graph500 inputs (weak-scaling) on a CRAY XE6m, along with trendlines showing growth rates.



(a) (b) (c)

Figure 8.13: Throughput of (a) BFS, (b) connected components and (c) PageRank on Watts-Strogatz input on BGQ, **131,072 cores**. At scale, the hierarchical approach has an improvement of 1.59x, 1.78x and 1.64x over the flat approach, respectively.

to lower core-counts to observe trends and explain our results. The performance benefit of our approach depends on the computation/communication cost ratio. For instances where the communication becomes a bottleneck, our approach benefits greatly. This is dependent on three things: the input graph, the algorithm and the system. A denser graph is harder to partition effectively, and therefore will result in cross-edges with a higher probability, resulting in higher communication costs. On the other hand, algorithms such as betweenness centrality perform heavier computation than algorithms such as breadth-first search, and can therefore hide the communication overhead better, while some systems, such as the IBM Blue Gene/Q, have slower processors and faster networks to achieve the same effect. Figure 8.13 shows the performance of various algorithms on a

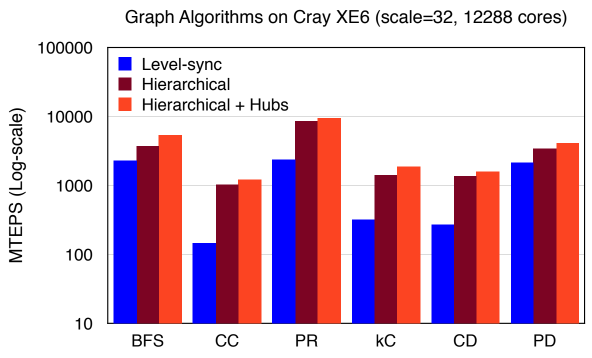
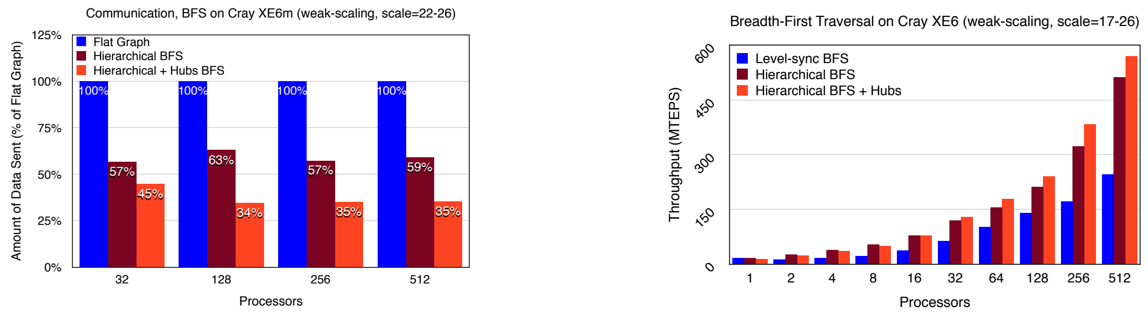


Figure 8.14: Scalability (Throughput, log-scale) of various algorithms using hierarchical approach. Graph500 input graph with 4 billion vertices and 64 billion edges at **12,288 cores**.

Watts-Strogatz small-world network on up to **131,072 cores**. At scale, the hierarchical approach is able to see improvements of $1.59\times$ to $1.78\times$ over the traditional flat algorithm. Our experiments are designed to evaluate our approach on a wide range of important graph algorithms that are representative in their class, as well as different systems.

Performance at Scale. We ran breadth-first search (BFS), connected components (CC), PageRank (PR) and k-core decomposition (KC) algorithms at scale on **12,288 cores** on a Cray XE6 machine to demonstrate the performance benefits of the hierarchical approach at scale. Our results (Figure 8.14) show a $2.35\times$ performance improvement for breadth-first search, a $7.26\times$ to $8.54\times$ improvement for connected components, a $3.6\times$ to $3.95\times$ improvement for PageRank, and a $4.43\times$ to $5.84\times$ improvement for k-core decomposition. We attribute these improvements to lower communication and better load-balance provided by the hierarchical approach, which improves scalability, as we will demonstrate in this section. Compared to the IBM Blue Gene/Q system, the performance benefits on the Cray XE6 are more substantial using our approach. This is due to the Blue Gene system having slower cores and a faster network, which lowers the computation/communication ratio compared to the Cray XE6. When the ratio is higher (due to a slower interconnect, for example), our approach yields better benefits.



(a) (b)

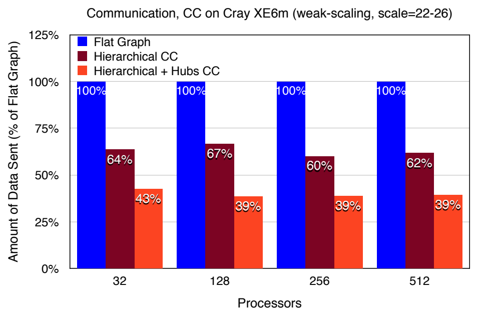
Figure 8.15: Communication reduction using hierarchical approach and (b) Scalability (Throughput) of BFS on Graph500 benchmark.

8.9.3 Fundamental Algorithms

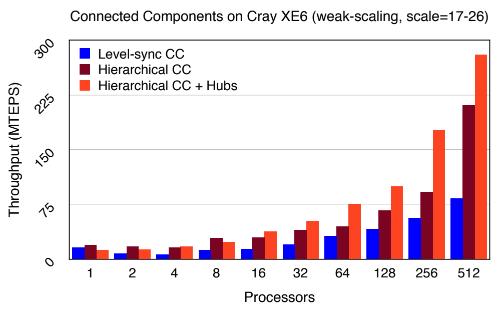
Graph500 BFS. We evaluated the Graph500 benchmark application that simulates data-intensive HPC workloads. The benchmark performs breadth-first traversals of the input graph from multiple source vertices. The Graph 500 input graph simulates social networks and web-graphs and exhibits small-world scale-free behaviour, i.e., it has a short diameter (< 16 hops) and vertices with very high out-degrees. These high out-degree vertices (hubs) cause scalability bottlenecks due to communication.

Figure 8.15(a) compares the number of bytes communicated in the base-line (flat) approach with that in our hierarchical approach. As can be observed, the hierarchical approach is able to significantly reduce the number of bytes sent over the network by $2.7\times$ to $3.3\times$. This directly translates to the performance of the algorithm (Figure 8.15(b)), where we observe a $1.8\times$ to $2.1\times$ improvement over the base algorithm. For example, at 512 cores, BFS on the Graph 500 input graph communicated (sent/received) $33.87\ GB$ of data across the system in the base (flat) case. This was reduced to $12.56\ GB$ for our hierarchical approach and then further to $10.3\ GB$ for the hierarchical with hubs approach. With the network sending only a third of the data, performance improved.

Connected Components. Connected components [42] (CC) has a heavier communication pattern than breadth-first traversal, and therefore, we expect our hierarchical strategy to provide a higher



(a)



(b)

Figure 8.16: Communication reduction using hierarchical approach and (b) Scalability (Throughput) of Connected Components on Graph500 input.

speedup versus the traditional paradigm. Figure 8.16 shows this to be the case. As we increase in scale, the benefits of our approach become more evident. At 512 cores, the total bytes communicated was reduced from 111.9 GB in the base-case to 68.91 GB for our hierarchical approach, to 44.1 GB for the hierarchical with hubs approach, a reduction of $1.6\times$ to $2.6\times$ respectively (Figure 8.16(a)), providing a $2.54\times$ to $3.38\times$ speedup over the base-case (Figure 8.16(b)).

8.9.4 Graph Mining Algorithms

PageRank. PageRank [15, 16] is an important algorithm used to rank web-pages on the internet in order of relative importance. As an example of an iterative random walk algorithm, each vertex calculates its rank in iteration i based on the ranks of its neighbors in iteration $i-1$ and then sends out new ranks to its neighbors for the next iteration. The algorithm terminates when either a certain number of iterations have been reached or the ranks have converged.

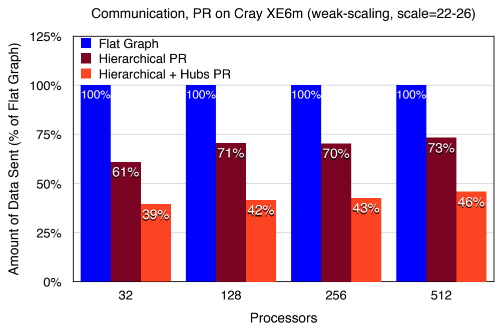
PageRank exhibits even heavier communication than connected components, due to all vertices being active (and communicating over all edges) in every iteration of the PageRank algorithm, making it a worst-case scenario for communication. Our evaluation of this algorithm in Figure 8.17 shows a communication reduction of $1.36\times$ to $2.18\times$ (from 711 GB for flat to 521 GB for hierarchical and 326 GB for hierarchical with hubs), corresponding to a $2.54\times$ to $2.73\times$ speedup over the base-case at 512 cores, which shows the worst-case communication is substantially improved.

k-core Decomposition. A k-core of a graph G is a maximal connected sub-graph of G in which all vertices have degree at least k. The k-core algorithm is widely used to study clustering and evolution of social networks [43]. It is also used to reduce input graphs to more manageable sizes while maintaining their core-structure. The typical parallel algorithm iteratively deletes vertices with degree less than k until only vertices with degree greater than or equal to k exist. k-core has a different communication pattern than either of BFS, CC or PageRank. Here too, the hierarchical approach is able to provide benefits over the base case (Figure 8.18). At 512 cores, our approach is able to reduce the total bytes communicated across the system from $48.8\ GB$ for the base-case to $21.4\ GB$ for the locality-based hierarchy to $11.2\ GB$ for the hierarchy with hubs, a reduction of $2.3\times$ to $4.3\times$, leading to a speedup of $1.2\times$ to $1.9\times$.

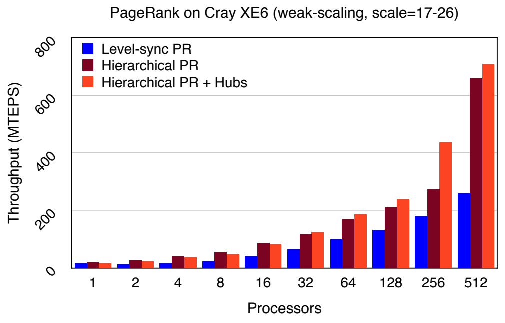
Community Detection. Community Detection is an important application that is widely used to detect groups or clusters in social networks. We use a modularity maximization algorithm to label vertices to their assigned communities. The application iteratively assigns the most frequently occuring label in the local neighborhood of each vertex, until the global quality of the labeling can no longer be improved. Figure 8.19 shows the improvement in performance obtained by using our hierarchical approach. We note that while the locality-based hierarchy provides a significant improvement in performance, the addition of the hub-based hierarchy does not further improve the performance appreciably. This is due to the fact that, as described in Section 4.2.3, the hub-based hierarchy relies on reducing updates on hubs to a single value on each location. However, for community detection the labels of the entire neighborhood is needed, leading to a minimal reduction for the hub-hierarchy.

8.9.5 Application of Traversals

Betweenness Centrality. Betweenness Centrality is an important graph-mining algorithm that is used to identify vertices with large influence in a network. For example, it is used to understand the social influence of users on Facebook and Twitter. Our implementation uses Brandes' algorithm [17], which performs forward and backward traversals of the graph to compute the number of shortest paths passing through each vertex. This application directly benefits from our hierarchical approach, as seen in Figure 8.19. Due to the large number of traversals involved, the actual

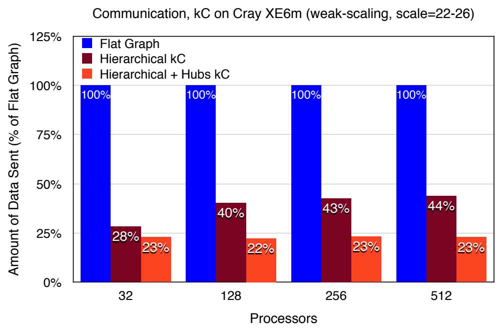


(a)

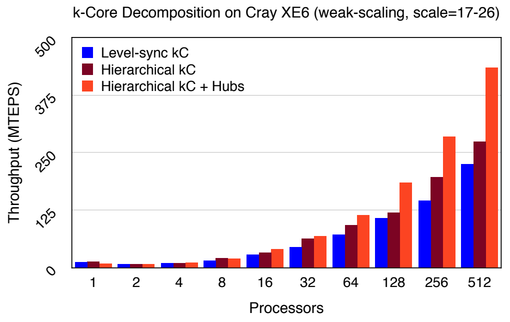


(b)

Figure 8.17: Communication reduction using hierarchical approach and (b) Scalability (Throughput) of PageRank on the Graph500 input graph.



(a)



(b)

Figure 8.18: Communication reduction using hierarchical approach and (b) Scalability (Throughput) of k-core on Graph500 input.

saving in time is significant. For example, on 512 cores, the execution time of the application was reduced from $1,635.93$ seconds to $1,094.63$ seconds, an improvement of 50%.

Pseudo-Diameter. Pseudo-Diameter is a graph metric used in network analysis to understand the structure of networks. It iteratively traverses the graph from a source, and selects the farthest vertex as the source for the next traversal, until the distance from a source to its farthest vertex can not be increased. Being a direct application of multiple traversals, improvements (Figure 8.19) mirror those in BFS.

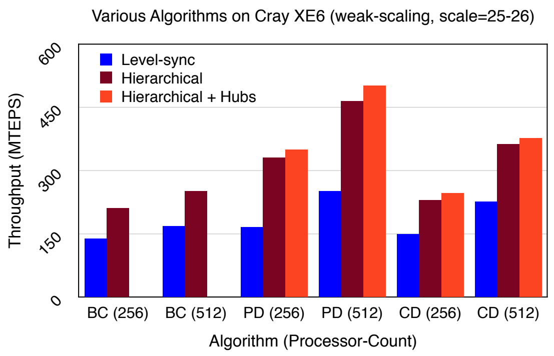


Figure 8.19: Throughput of Betweenness Centrality (BC), Pseudo-Diameter (PD), and Community Detection (CD) on Graph500.

8.9.6 Other Graphs

We also evaluate our approach on the extreme cases of Erdos-Renyi graphs, toroidal meshes and the real-world Twitter social-network graph. A 2D toroidal mesh is the worst-case scenario for our hierarchical approach, as the maximum out-degree of any vertex in the mesh is 4, and when partitioned correctly, the number of cut-edges is minimized. We evaluated our approach on this graph to show that even though we do not expect to improve performance, we add no overhead for such cases either. Figure 8.20 shows the performance of a BFS traversal of a toroidal mesh with 16 million vertices and 64 million edges, where the overhead of using hierarchies is less than 3%.

On the other hand, an Erdos-Renyi random graph [44] is the best-case for our approach. An Erdos-Renyi network tries to connect each vertex of a graph with every other vertex with a probability p ($p = 1$ leads to a complete graph). We generate an Erdos-Renyi network of 1 million vertices and 5.5 billion edges (a probability $p = .5\%$) to show the benefit of our hierarchical approach when the connectivity of the input graph is high. This is shown in Figure 8.20, where the hierarchical approach is $42.5\times$ faster.

To evaluate how the performance improvement varies with graph connectivity, we measured the performance improvement of our approach on a Watts-Strogatz random network for varying rewire probabilities. The Watts-Strogatz model [41] produces random networks with the small-world properties such as short average path lengths and a high degree of clustering. The model starts with a regular ring lattice with every vertex connected to its k nearest neighbors on either side, forming a ring-like structure. Thereafter, for each vertex, each of its edges is 'rewired' with a probability β, such that the target of the edge is selected with uniform probability from the remaining vertices, avoiding self-loops and duplicate edges. By varying β, one can generate graphs that are ring-like and do not exhibit small-world properties (for small values of β), to small-world networks with short path-lengths and high degree of clustering (as β approaches 1).

Figure 8.21 plots the speedup of our approach over the base-line as the rewire-probability is varied from 0 to 1 for various processor counts. While our approach has a 25% overhead on the BG/Q machine for rewire-factor $\leq 3\%$, it shows an improvement over the base-line for all other rewire-factors ($3\% - 100\%$). The initial overhead is due to the graph being well-partitioned with extremely few cut-edges. In this case, there is not much communication to reduce, and we just measure the overhead of going through the hierarchy. It is higher for the BG/Q machine than the Cray XE6 (3% for torus in Figure 8.20, and we measured 1.5% for Watts-Strogatz with 0% rewiring) as the processors are much slower and the network is extremely fast, so the effect is more pronounced. This can be eliminated by profiling the system using this example to compute the number of cross-edges beyond which the hierarchical approach should be used. As the algorithms remain the same, this can be done cheaply at runtime.

We can observe that the improvement over the base-line increases as the small-world behavior increases. Small-world graphs are harder to partition well and produce a large number of cross-partition edges, which result in heavy communication, thus limiting performance. Our approach alleviates this, improving performance. This is evident even in machines where the computation/communication ratio is low (i.e. better network, slower processors), such as the BG/Q. For machines with (comparatively) slower networks and faster processors, the improvement is even better, as observed in experiments on the Cray XE6 (Figure 8.14 and others).

Finally, we also evaluate our approach on the Twitter social network from 2010. This network has 65 million vertices and 1.2 billion edges, and has large hub vertices corresponding to popular users who have many followers. The presence of large hubs leads to poor scalability, which can be

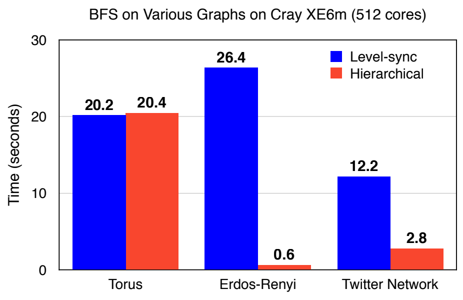


Figure 8.20: Running times of BFS on various input graphs using flat and hierarchical approaches.

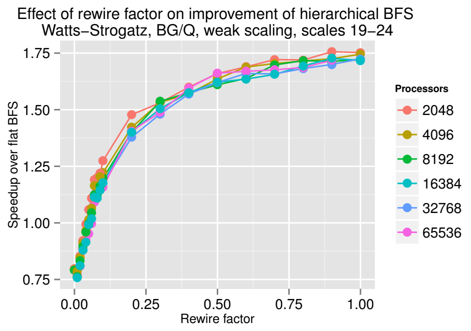


Figure 8.21: Improvement of the hierarchical approach on BFS for varying rewiring-probability of a Watts-Strogatz graph.

effectively addressed by using the hierarchical approach. As a result of communication reduction from $21\ GB$ to $8.2\ GB$ and a better load-balance, our approach provides a speedup of $4.36\times$ (Figure 8.20) over the flat approach.

8.9.7 Overhead of Hierarchy Creation

In order to run hierarchical algorithms, the input graph needs to be converted to its hierarchical representation on the machine. This is a one-time process following graph construction, after which multiple algorithms can take advantage of the hierarchy. Figure 8.22 compares the time to construct the flat graph to the time to construct the hierarchical graph. The time to generate the hierarchical graph includes the time to create the flat graph. Hierarchical graph creation overhead is proportional to the number of cross-processor edges in the graph. Therefore, for graphs with few

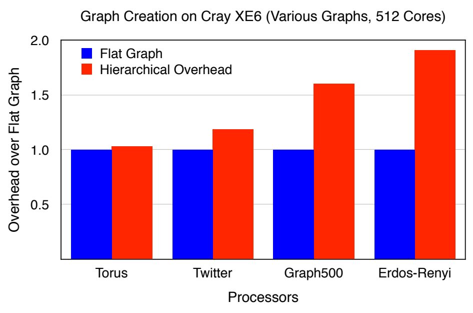


Figure 8.22: Comparison of total construction times of flat and hierarchical graphs for various inputs.

cross-processor edges, such as a toroidal mesh, the hierarchy construction adds negligible overhead, while for graphs with a large number of cross-processor edges, such as a dense Erdos-Renyi graph, the hierarchy creation overhead is larger (Figure 8.22). Correspondigly, the benefit of the hierarchy is also significantly larger for the Erdos-Renyi graph vs. the torus (Figure 8.20). As an example, on 512 cores, the graph construction time for 67 million vertices and 2.1 billion edges was 8.5 seconds, while creating the hierarchy added 5.2 seconds to that time. A comparison showing the running time of a *single run* of various algorithms on hierarchical and flat graphs is shown in Figure 8.23, along with the time required to create the hierarchies, to show the overhead. For example, a PageRank algorithm using the traditional approach takes 68.7 seconds, while a PageRank using the hierarchical approach uses 39.7 seconds. To amortize the cost of hierarchy creation, BFS would need to be run 4 times, however, in many use-cases, such as betweenness centrality and pseudo-diameter, BFS is usually executed multiple times from different sources, allowing the hierarchy creation to be ammortized, even for a single execution of the algorithm (Figure 8.23). Other algorithms, such as community detection, PageRank, and k-core decomposition also observe significant overall benefits even in a *single* execution, including the cost of hierarchy creation, as shown in Figure 8.23. We note that the hierarchy is algorithm-agnostic and only depends on graph structure, and not the metadata (vertex/edge properties), and can therefore be used with different algorithms once created.

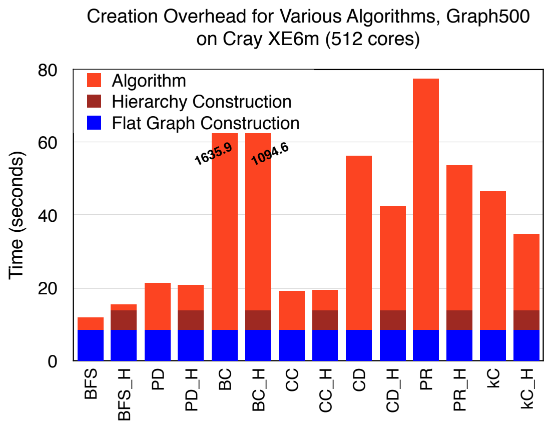


Figure 8.23: Comparison of graph construction and algorithm execution times of flat and hierarchical graphs for Graph500 input.

8.10 Out-of-Core Performance

In this section, we evaluate the performance of our approach on various platforms, and compare it with existing disk-based libraries. Our approach is evaluated on a set of important graph mining and graph analytics algorithms, and a variety of inputs. The input graphs include the Graph 500 benchmark inputs [20], a benchmark for data-intensive and graph applications, as well as real-world graphs available to us.

Our experiments were run on multiple platforms – a Cray XE6 machine with 153,216 cores (Hopper), and a smaller Cray XE6m machine with 576 cores available to us. In addition, we evaluated our technique on two 4-core PCs with 4GB RAM and 16GB RAM (respectively), both with 7200 RPM hard disk drives, and a 4-node commodity cluster with two quad-core processors per node. Finally, we ran on an EXYNOS 4 quad-core tablet running Android 4.0, containing an ARM Cortex-A9 with 1GB of DDR2 main memory and a 16GB MMC card for storage. Graphs were distributed as specified by their input files, using a block distribution of the vertices.

Single Node Performance. We compared STAPL GL's single node out-of-core performance with GraphChi, a graph-processing system designed to solve large graph problems on a single node efficiently. GraphChi is a sister project of GraphLab that focuses on allowing the processing of large graphs on a single-node computer using a parallel sliding-window technique. In [12], the authors show how GraphChi has comparable performance, even on a single node, to existing frameworks, such as Spark, Hadoop and GraphLab running on much larger machines.

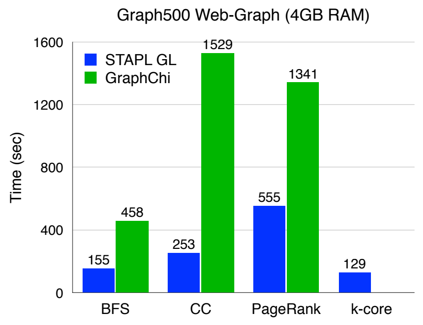
GraphChi partitions the input graph into p partitions and processes the partitions in a sliding-window. However, much like in GraphLab, GraphChi's pull-model reads neighboring vertices' values, which may require $p - 1$ reads for p partitions in the worst-case. Furthermore, as noted in [12], the technique used in GraphChi is unable to take full advantage of available RAM, as their model is not adaptive to the resources.

Figure 8.24(a) shows the out-of-core performance of various graph algorithms in the STAPL GL and GraphChi on a 4-core PC with 4GB RAM. For GraphChi, we ran the experiments on different configurations and explored the space of parameters to choose the fastest configuration for comparison. As can be observed, STAPL GL's push-model provides $2.5-6\times$ faster performance than GraphChi's pull-model. Further, our approach allows better scalability with increasing RAM sizes. Figure 8.24(b) shows the same graph on a 4-core PC with 16GB RAM. Increasing the RAM and resources allows a $4\times$ or better performance increase for the STAPL GL. However, GraphChi is not able to optimally utilize all available RAM, and therefore, STAPL GL is able to perform $4 - 12\times$ better with increased resources. GraphChi's k-core algorithm was designed to work with matrix inputs, and did not accept edge-list or adjacency-list representations, so thus we were unable to obtain comparative results for it. We evaluated our performance on two real-world graphs. Figure 8.28 shows performance of the STAPL GL on Twitter and Friendster social-networks on a PC with 4GB RAM.

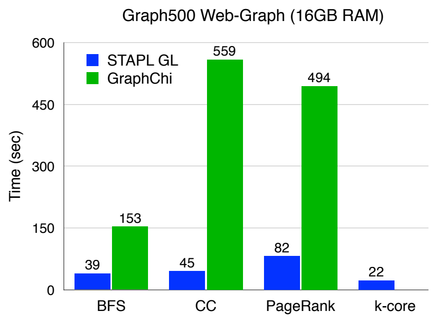
We studied the behavior of our approach with respect to memory size in Figure 8.25 for breadth-first search and PageRank on the Graph500 graph. As shown, total time to completion decreases as we increase memory size. In fact, at the largest memory size, where the graph fits in memory, our approach performs the same as the in-memory STAPL GL variant. When the available memory is restricted, a majority of the subgraphs are present on disk. Therefore, a higher number of updates to vertex values generated during computation have to be written to disk, increasing the load and store times. As the available memory increases, larger portions of the graph can fit in memory, allowing more updates to happen in RAM. When the entire graph fits in memory, all updates happen in RAM, resulting in negligible overhead due to loading and storing. We observe this behavior for PageRank in Figure 8.25(b), where there is a substantial improvement in run-time from 4 GB to 8 GB, which is significantly larger than the benefit in BFS (Figure 8.25(a)). This is due to a larger volume of updates generated by PageRank, and thus written to disk for the out-of-core case, compared to BFS's fewer updates from only a subset of the graph's vertices per superstep, as shown in Figure 5.2. Due to this, the amount by which an increase in memory improves performance is both input and algorithm dependent.

Android Tablet Performance. To further test our approach with limited RAM, we also tested on an Android tablet with 1GB RAM, using two cores. We use this as an extreme example to test the performance of our method, and not necessarily as a viable graph-processing system. Even so, we were able to process the Graph500 input graph using our approach, as seen in Figure 8.26. The time taken to execute the algorithms was much higher than on a PC, due to the different architecture of the processor, lower clock-speed, lower RAM and far slower disk speed (the storage used was a low power MMC card). The size of the input graph was also limited due to the capacity of the memory-cards available. However, this experiment demonstrates that our approach can scale well to systems with limited memory as compared to the size of the input graph. We were unable to get GraphChi or other graph processing systems to run on our Android tablet.

Multi-Node and Cross-Platform Performance. As we use asynchronous forwarding and each subgraph is managed individually only by its home location, the technique naturally extends to multiple nodes. Figure 8.27 shows the execution of multiple algorithms on a distributed-memory system with a graph that does not fit fully in RAM. As we increase the number of nodes available, larger portions of the graph can fit in RAM and more processors are available, allowing faster execution. When the graph fits completely in RAM (on 8 nodes), no penalty is paid for disk access, and the fastest execution time can be observed.

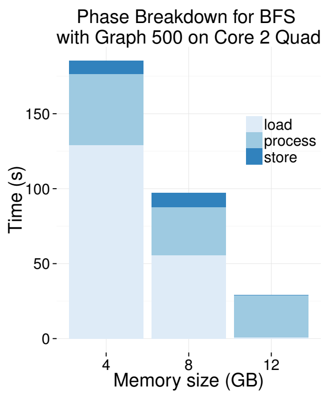


(a)

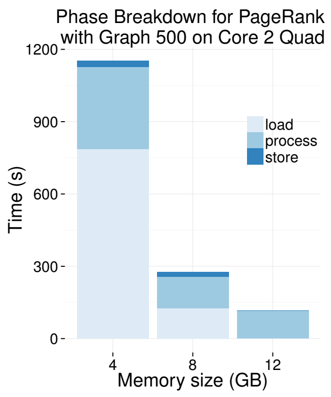


(b)

Figure 8.24: STAPL GL running time (various algorithms) vs. GraphChi on the Graph500 benchmark input with 16 million vertices, 256 million edges, running on PC with (a) 4GB RAM and (b) 16GB RAM.



(a)



(b)

Figure 8.25: Breakdown of times for different phases (loading, storing and processing) for (a) BFS and (b) PageRank with respect to memory size.

We also tested our approach across platforms to demonstrate its effective use of resources. Figure 8.29 shows the Graph500 input graph scaling from an Android tablet to a Cray XE6m machine with 128 cores. Figure 8.30 shows the Twitter input graph on a PC, a cluster of 4 compute nodes, and a Cray XE6m machine on 512 cores. Our framework is able to accommodate processing these large graphs, while adapting well to the resources available on each machine. In this sense, the program is able to scale with the machine without user-code modification.

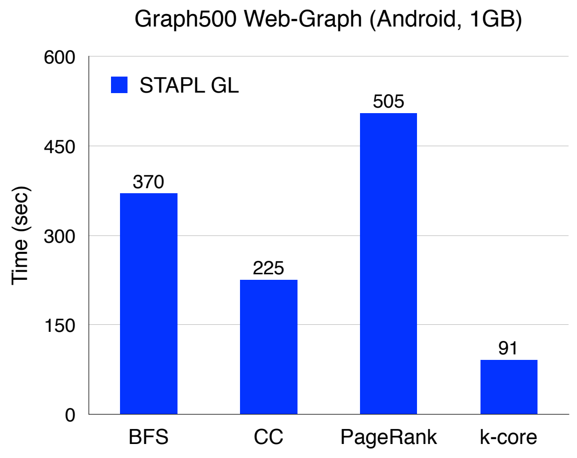


Figure 8.26: STAPL GL running time (various algorithms) on the Graph500 benchmark input on an Android tablet with 1GB RAM, 4 million vertices, 64 million edges.

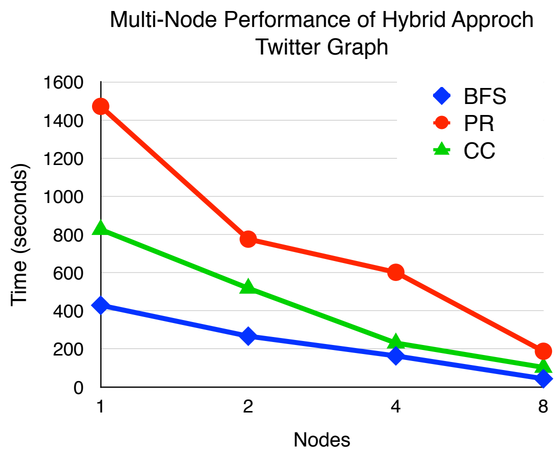


Figure 8.27: STAPL GL running time for BFS, PageRank (PR) and connected components (CC) on the Twitter input on multiple nodes of Cray XE6.

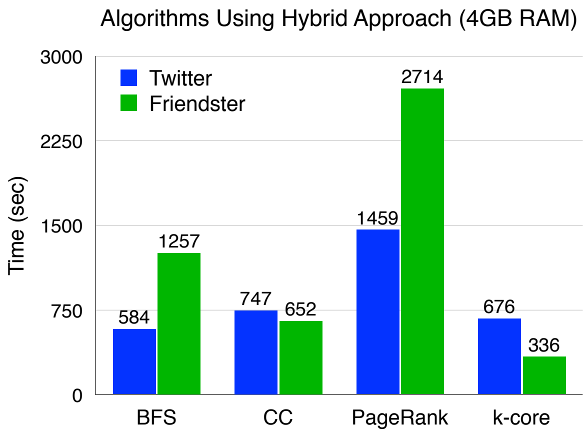


Figure 8.28: STAPL GL running time (various algorithms) on 4GB PC on the Twitter graph with 65 million vertices, 1.2 billion edges, and the Friendster graph with 118 million vertices, 2.6 billion edges.

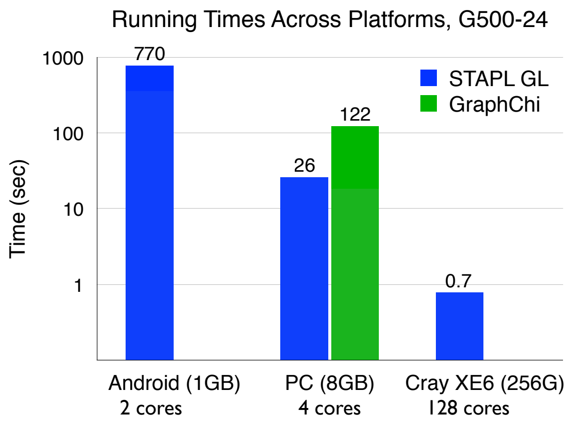


Figure 8.29: STAPL GL running time across platforms on the Graph500 input graph with 16 million vertices, 256 million edges.

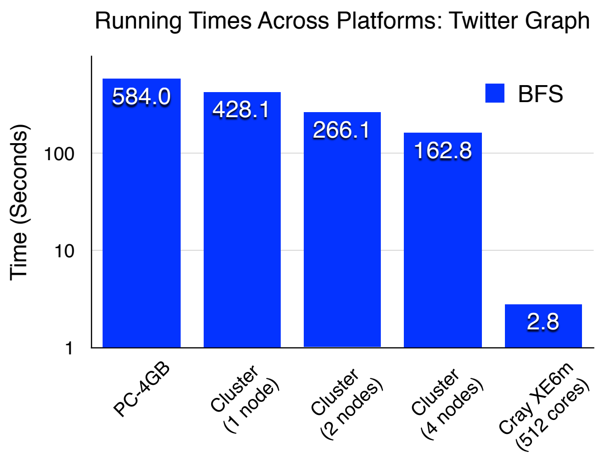


Figure 8.30: STAPL GL running time across platforms on the Twitter input graph with 65 million vertices, 1.2 billion edges.

9. SUMMARY

This work focuses on the STAPL GL, a parallel graph library that allows natural expression of graph algorithms while providing scalable performance and portability. An implementation based on our vertex-centric asynchronous push model displayed improved performance and scalability vs. existing parallel graph processing libraries and hand-written implementations of benchmarks, while being expressive to use.

Our model allows decoupling of the algorithm from its execution, allowing us to choose the best suited execution paradigm for a given combination of input graph, algorithm and system. To take advantage of this, we proposed three novel paradigms to address the challenges in the field of large-scale graph processing. The first paradigm unified the two existing graph-processing approaches, namely BSP and asynchronous, while extending them to allow fine-grained control over the amount of asynchrony in the system in order to balance the cost of global synchronizations with redundant work. This helped hide the latency in large systems. The second paradigm utilized algorithmmic redundancy and the knowledge of the machine model to reduce the amount of communication for small-world graphs, thereby increasing the effective bandwidth of the system. The third paradigm introduced subgraph-paging, which together with the asynchronous push model allowed efficient out-of-core processing of large graphs by eliminating random disk-reads and minimizing random disk-writes.

Experimental studies on large-scale systems and a variety of graphs show our baseline implementation to perform better than existing graph processing systems, while using the novel paradigms can improve performance by $8\times$ to $10\times$.

The STAPL Parallel Graph Library has also been used to successfully implement real-world scientific applications such as Seismic Ray-Tracing, Nuclear Particle Transport, and Motion-Planning and Protein-Folding, all of which have demonstrated good scalability and performance.

REFERENCES

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

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

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

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

[5] M. Adams and E. Larsen, “Fast iterative methods for discrete-ordinates particle transport calculations,” *Progress in nuclear energy*, vol. 40, no. 1, pp. 3–159, 2002.

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

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

[8] J. W. Berry, B. Hendrickson, S. Kahan, and P. Konecny, “Software and algorithms for graph queries on multithreaded architectures,” *Parallel and Distributed Processing Symposium, In-*

ternational, vol. 0, p. 495, 2007.

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

[10] M. A. Hassaan, M. Burtscher, and K. Pingali, "Ordered and unordered algorithms for parallel breadth first search," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, (New York, NY, USA), pp. 539–540, ACM, 2010.

[11] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, (New York, NY, USA), pp. 135–146, ACM, 2013.

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

[13] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning and data mining in the cloud," pp. 716–727, 2012.

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

[15] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Comput. Netw. ISDN Syst.*, vol. 30, no. 1-7, pp. 107–117, 1998.

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

[17] U. Brandes, "A faster algorithm for betweenness centrality," *Journal of Mathematical Sociology*, vol. 25, pp. 163–177, 2001.

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

[19] N. Edmonds, J. Willcock, and A. Lumsdaine, “Expressing graph algorithms using generalized active messages,” in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP ’13, (New York, NY, USA), pp. 289–290, ACM, 2013.

[20] “The graph 500 list.” `http://www.graph500.org`, 2011.

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

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

[23] I. Hoque and I. Gupta, “Lfgraph: Simple and fast distributed graph analytics,” in *Proceedings of Conference on Timely Results in O.S.*, pp. 1–17, 2013.

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

[25] D. Yan, J. Cheng, Y. Lu, and W. Ng, “Effective techniques for message reduction and load balancing in distributed graph computation,” in *WWW*, 2015.

[26] M. H. Nodine, M. T. Goodrich, and J. S. Vitter, “Blocking for external graph searching,” in *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS ’93, (New York, NY, USA), pp. 222–232, ACM, 1993.

[27] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter, “External-memory graph algorithms,” in *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA ’95, (Philadelphia, PA, USA), pp. 139–149, Society for Industrial and Applied Mathematics, 1995.

[28] Y. Guo, M. Biczak, A. Varbanescu, A. Iosup, C. Martella, and T. Willke, “How well do graph-processing platforms perform? an empirical performance evaluation and analysis,” in *Proc. International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 395–404, 2014.

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

[30] A. Roy, I. Mihailovic, and W. Zwaenepoel, “X-stream: Edge-centric graph processing using streaming partitions,” in *Proceedings of the Symposium on Operating System Principles*, pp. 472–488, 2013.

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

[32] M. Ghosh, S. L. Thomas, M. M. Aguirre, S. Rodriguez, and N. M. Amato, “Motion planning using hierarchical aggregation of workspace obstacles,” in *Proc. IEEE Int. Conf. Intel. Rob. Syst. (IROS)*, IEEE, 2016.

[33] W. D. Hawkins, T. Smith, M. P. Adams, L. Rauchwerger, N. Amato, and M. L. Adams, “Efficient massively parallel transport sweeps,” *Transactions of the American Nuclear Society*, vol. 107, 2012.

[34] W. D. Hawkins, T. S. Bailey, M. L. Adams, P. N. Brown, A. J. Kunen, M. P. Adams, T. Smith, N. M. Amato, and L. Rauchwerger, "Validation of full-domain massively parallel transport sweep algorithms," *Transactions of the American Nuclear Society*, vol. 111, 2014.

[35] T. K. Jain, "Parallel seismic ray tracing," tech. rep., 2013.

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

[37] D. Culler, J. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1st ed., 1998. The Morgan Kaufmann Series in Computer Architecture and Design.

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

[39] "9th dimacs implementation challenge.." `http://www.dis.uniroma1.it/challenge9/`, 2013.

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

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

[42] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations," in *Proceedings of the IEEE International Conference on Data Mining*, pp. 229–238, IEEE, 2009.

[43] J. I. Alvarez-hamelin, A. Barrat, and A. Vespignani, “Large scale networks fingerprinting and visualization using the k-core decomposition,” in *Advances in Neural Information Processing Systems 18*, pp. 41–50, MIT Press, 2006.

[44] P. Erdos and A. Renyi, “On random graphs. i,” in *Publicationes Mathematicae Debrecen*, pp. 290–297, 1959.