Cache and NUMA Optimizations in A Domain-Specific Language for Graph Processing

by

Mengjiao Yang

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2018

Author .....
Department of Electrical Engineering and Computer Science
May 25, 2018

Certified by .....
Julian Shun
Assistant Professor
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chairman, Masters of Engineering Thesis Committee

Cache and NUMA Optimizations in A Domain-Specific Language for Graph Processing

by

Mengjiao Yang

Submitted to the Department of Electrical Engineering and Computer Science
on May 25, 2018, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

High-performance graph processing is challenging because the sizes and structures of real-world graphs can vary widely. Graph algorithms also have distinct performance characteristics that lead to different performance bottlenecks. Even though memory technologies such as CPU cache and non-uniform memory access (NUMA) have been designed to improve software performance, the existing graph processing frameworks either do not take advantage of these hardware features or overcomplicate the original graph algorithms. In addition, these frameworks do not provide an interface for easily composing and fine-tuning performance optimizations from various levels of the software stack. As a result, they achieve suboptimal performance.

The work described in this thesis builds on recent research in developing a domain-specific language (DSL) for graph processing. GraphIt is a DSL designed to provide a comprehensive set of performance optimizations and an interface to combine the best optimization schedules. This work extends the GraphIt DSL to support locality optimizations on modern multisocket multicore machines, while preserving the simplicity of graph algorithms. To our knowledge, this is the first work to support cache and NUMA optimizations in a graph DSL.

We show that cache and NUMA optimizations together are able to improve the performance of GraphIt by up to a factor of 3. Combined with all of the optimizations in GraphIt, our performance is up to 4.8x faster than the next fastest existing framework. In addition, algorithms implemented in GraphIt use fewer lines of code than existing frameworks.

The work in this thesis supports the design choice of a compiler approach to constructing graph processing systems. The high performance and simplicity of GraphIt justify the separation of concerns (modularity) design principle in computer science, and contribute to the larger effort of agile software systems development.

Thesis Supervisor: Julian Shun
Title: Assistant Professor

Acknowledgments

I would like to express my gratitude to many people without whom this thesis would not have been possible.

First and foremost, I would like to thank my thesis advisor, Prof. Julian Shun, for patiently guiding me through my master's studies. Julian introduced me to parallel computing through MIT's performance engineering class (6.172), where my interest in building faster software systems originated. I was fortunate enough to be a TA for his class on graph analytics (6.886), where I drew from the breadth of his knowledge and acquired a deeper understanding of various aspects of graphs. I am very grateful to Julian for answering all my questions, sharing with me many of his research ideas, encouraging me to think about theoretical graph problems, and even helping me improve my writing skills.

I would also like to thank Yunming Zhang and Prof. Saman Amarasinghe for mentoring me on this project. Yunming Zhang initially proposed integrating cache and NUMA optimizations in GraphIt, and worked closely with me to turn these ideas into reality. I thank Saman for pointing out high-level directions that my project could take, reminding me of the bigger picture, and giving me advice on presenting GraphIt to other people.

Next, I would like to express my appreciation for other members of the Commit group. I am thankful to Vladimir Kiriansky for the crash courses on various topics related to computer memories; to Fredrik Kjolstad for our discussions on the relationship between graph computation and tensor algebra; and to Charith Mendis and Stephen Chou for tips on thesis writing and for their perspectives on being a Ph.D student.

In addition, I want to thank Prof. Frans Kaashoek and Prof. Robert Morris for equipping me with the systems knowledge this project required through their Operating Systems (6.828) and Distributed Systems (6.824) courses. I thank Prof. Martin Rinard for telling me to work hard and for the valuable life advice he has offered. I am also grateful to my undergraduate advisor, Prof. Patrick Winston, for reminding me of the importance of academic research and encouraging me to reach out to other professors for research opportunities.

Finally, I want to express my utmost gratitude to my wonderful parents, Xi Yang and Wenjuan Zhao, for always treating me as their equal and encouraging me to think critically at a very young age. Without their endless love and support, I would not have become the person I am today.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Graph analytics is widely used in solving many large-scale real-world problems. For example, friend suggestion can be modeled as triangle enumeration on a social network graph [16], traffic planning is equivalent to path finding in a road network [32], and cancer prediction can be tackled by subgraph matching in a protein-protein interaction network [7]. Many real-world graphs, however, are large in size and irregular in structure, imposing a major challenge on modern hardware with its memory constraints. While designing programs that are aware of caching and non-uniform memory access (NUMA) is crucial for obtaining high performance, utilizing them in graph processing can be tedious and error-prone. A number of graph processing frameworks have implemented NUMA optimizations [18, 38, 43, 46], but they tend to trade programmability for performance and are often not flexible enough to consistently perform well on different input graphs and algorithms.

1.1.1 Large and Irregular Graph Structures

The largest publicly available graph—the hyperlink web graph from the 2012 Common Crawl—has 3.5 billion nodes (webpages) and 128 billion edges (hyperlinks) and takes up 540 GB memory [27]. There are even larger proprietary graphs with trillions

of edges from Facebook and Yahoo [10, 36]. Since most graph algorithms perform little computation, memory latency has been identified as the major performance bottleneck [3]. Being able to efficiently address memory is the key to achieving high performance on large graph datasets.

Aside from graph size, irregular graph structures introduce a significant challenge to achieving high performance in graph algorithms. Many real-world graphs have a power-law distribution where a small fraction of vertices are adjacent to a large number of edges. Hence the amount of work at each vertex can be drastically different. This makes task scheduling among threads as well as balanced graph partitioning difficult.

1.1.2 Existing Memory Technologies

Caches refer to small memories residing on or close to the CPU and hence able to operate much faster than the main memory. Programs with good spatial and temporal data locality can effectively make use of this mechanism and reduce memory latency. Graph processing, however, exhibits poor locality, as there can be an edge connecting any two vertices; naively traversing all the neighbors of a vertex can result in a high cache miss rate. Prior experiments show that CPUs are often stalled on high latency DRAM (direct random access machine) accesses [3]. Methods such as graph reordering [33, 40] and variants of cache blocking [4, 44] have been proposed to reduce cache misses by changing graph layouts.

In addition to cache locality, node locality on NUMA machines also significantly impacts system performance. It is widely known that the speed of modern CPUs is much faster than the speed of memory devices. To increase the memory bandwidth and to reduce the latency of local memory access, the non-uniform memory access (NUMA) design of distributed shared memory (DSM) has been proposed [1, 30]. Multi-socket systems with NUMA support use separate memory in each socket, allowing shared memory access to scale proportionally to the number of processors [26]. However, this scalability does not come for free. Data migration between memory banks is expensive. Remote memory accesses, depending on the architecture, can take

2 to 7.5 times longer than local accesses [12]. While small graphs can be replicated on each processor's local memory, large graphs need to be partitioned and processed separately on each processor to increase the ratio of local to remote memory accesses. A handful of NUMA-aware graph processing frameworks have shown a considerable speedup over NUMA-oblivious systems [18, 38, 43, 46].

1.1.3 Graph Processing System Limitations

Existing NUMA-aware graph frameworks generally trade programmability for performance, mainly because NUMA optimizations involve low-level details of memory management and thread placement. As a result, graph algorithms in these frameworks are often accompanied by hardware-specific optimizations, complicating the overall application logic. For instance, Gemini [46], a distributed graph processing framework optimized for NUMA systems, has to manually implement NUMA-aware work-stealing because the schedulers of existing parallel computing languages such as Cilk [8] and OpenMP [11] are NUMA-oblivious. This kind of low-level complexity generalizes to other performance optimizations as well. Grazelle, a NUMA-aware shared-memory framework with edge list vectorization [18], includes hundreds of lines of assembly code, which severely hurts its readability.

The impact of NUMA optimizations in existing graph processing frameworks is not consistent across different algorithms and input graphs. Figure 1-1 shows the 24-core performance of Gemini, Grazelle, and Ligra (the first high-level shared-memory framework for parallel graph traversal) [34]. Despite employing many additional optimizations, Grazelle is only the fastest on about half of the algorithm-graph combinations, and Gemini is only the fastest on 1/5 of the combinations. Surprised by these low fractions, we investigated further and found that NUMA optimizations can sometimes have a negative impact on the work-efficiency and parallelism of a graph program, and should therefore be adjusted according to the algorithms and input graphs. Gemini and Grazelle do not provide this flexibility, nor do they support other known graph traversal optimizations such as cache blocking and flexible frontier data structures.

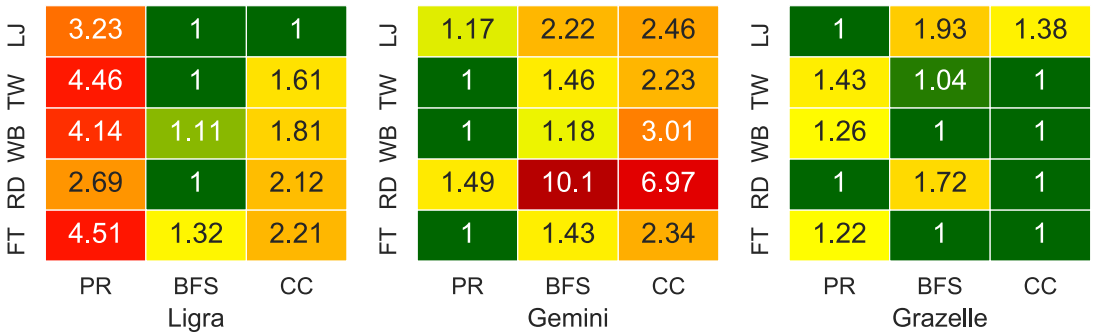

Figure 1-1: A heat map showing the running time slow downs of Ligra, Gemini, and Grazelle compared against the fastest framework among the three. The experiments are conducted on a dual-socket system with a total of 24 cores. The x-axis shows the algorithms: PageRank (PR), breadth-first search (BFS), and connected components (CC). The y-axis shows the graphs: LiveJournal (LJ), Twitter (TW), WebGraph (WB), USAroad (RD), and Friendster (FT). Lower number (green) is better. The entries with number 1 are the fastest.

1.2 Contributions

In this thesis, we explore ways to improve the performance of shared-memory graph processing using existing memory technologies. More specifically, we study the impact of cache and NUMA optimizations on a diverse set of graph traversal algorithms and input data. In addition, we explore various ways to carry out these optimizations and determine the best implementation. We also analyze the trade-offs made by cache and NUMA optimizations in terms of how they affect memory locality, work-efficiency, and parallelism.

Although these memory optimizations have been employed in some of the existing graph processing systems, Figure 1-1 shows that they can negatively impact performance when generalized to a broader set of algorithms and graphs. To improve the flexibility of these optimizations when applied across multiple algorithms and graphs, we expand the work of Zhang et al. on building a domain-specific language (DSL) for graph processing that separates algorithms from performance optimizations [45]. More specifically, we integrate cache and NUMA optimizations into the GraphIt compiler to enable flexibility in these optimizations. A compiler approach also allows these optimizations to be easily combined with other algorithm and data structure optimizations. To our knowledge, we are the first to enable cache and NUMA optimizations in a DSL. Our experiments show that cache and NUMA optimizations

improve the performance of GraphIt by 3 times. Combined with other optimizations, GraphIt can be as much as 4.8 times faster than the fastest existing framework.

1.3 Thesis Organization

This thesis is organized as follows:

In Chapter 2 (Background), we first introduce common graph traversal algorithms and the inefficiencies in their memory access patterns. We then give an overview of the NUMA memory design and its performance implications. We also introduce the infrastructure of the GraphIt compiler. The work in this thesis serves as an extension to the GraphIt compiler.

In Chapter 3 (Cache Blocking in GraphIt), we present our C++ implementation of cache blocking and explain how to automatically generate cache-optimized traversal algorithms in GraphIt.

In Chapter 4 (NUMA-Awareness in GraphIt), we present our C++ implementation of NUMA-aware graph algorithms, the additional optimizations that we attempted, and the integration of NUMA-awareness in GraphIt.

Chapter 5 analyzes the performance impact of cache blocking and NUMA-aware graph processing with respect to the version of GraphIt without these optimizations. In addition, it compares locality-optimized GraphIt to other graph processing frameworks. This chapter also discusses various trade-offs made by cache and NUMA optimizations.

Finally, Chapter 6 summarizes our work and points out potential directions for future research.

Chapter 2

Background

2.1 Notation

We denote a directed graph by $G = (V, E)$, where V represents the set of vertices and E represents the set of directed edges. A single vertex in the graph is denoted by v. $|V|$ and $|E|$ denote the number of vertices and the number of edges. $E_{dst<-src}$ denotes an incoming edge to the *dst* vertex and $E_{src->dst}$ denotes an outgoing edge from the *src* vertex. $N^-(v)$ denotes the set of in-neighbors and $N^+(v)$ denotes the set of out-neighbors of vertex v. We allow G to be divided into s subgraphs: $G_0, G_1, \ldots, G_{s-1}$, and denote $G_i = (V_i, E_i)$ to be one of these subgraphs. E_i denotes the edges in G_i, and V_i denotes the *home vertices* of G_i. Home vertices are the source vertices in the compressed sparse row (CSR) format and the destination vertices in the compressed sparse column (CSC) format. These sparse formats consist of an edge array and an offset array. The offset array stores the start and end edge-array index for the neighbors of each vertex [37]. The C++ code presented in Chapter 3 and Chapter 4 uses g to denote the original CSR graph and sg to denote a subgraph.

We define `writeMin(addr, val)` to be an atomic instruction that updates the memory at `addr` if `val` is less than the original value stored at that memory location. `parallel_for` is a keyword for initiating a parallel construct for statically or dynamically assigning a group of worker threads to the range specified so that each thread is responsible for processing a certain portion of the loop.

2.2 Graph Traversal Algorithms and Framework

This thesis focuses on memory-bound graph traversal algorithms that perform simple CPU computations. These algorithms often operate on a small number of vertices (the active frontier) during each iteration. Traversals can use the top-down approach (push), the bottom-up approach (pull), or the hybrid of the two introduced by Beamer et al. [2]. In the push direction, vertices on the active frontier propagate data to their neighbors and activate the neighbors accordingly. In the pull direction, inactive vertices check to see if their neighbors are on the active frontier, pull data from the active neighbors, and activate themselves. The hybrid approach switches traversal direction according to the size of the active frontier. Ligra generalizes this direction optimization to all the graph algorithms studied in this thesis.

This section first studies the memory access patterns of six such algorithms: PageRank, breadth-first search, connected components, single-source shortest path, PageRank-Delta, and collaborative filtering. Without special blocking or reordering, these algorithms often incur $O(|E|)$ random memory accesses, which leads to poor cache locality. In addition, accesses from processors in one NUMA node to memories in another NUMA node are high in latency and low in bandwidth. This section first briefly describes each algorithm and points out their memory inefficiencies (namely, random accesses that incur cache misses and remote memory accesses). We then introduce Ligra's edge traversal abstraction, which simplifies the implementation and optimization of these algorithms. GraphIt uses the abstraction that Ligra provides as a foundation for various optimizations.

PageRank is an algorithm first used to compute the relative importance of web-pages [9]. During every iteration, nodes in an input graph $G(V, E)$ update their ranks based on the ranks and outgoing degrees of their neighbors. Figure 2-1 shows an example implementation of PageRank where nodes pull the updates from their neighbors. Note that reading $N^-(v)$ on Line 2 and writing to `new_rank` on Line 3 incur (potentially cross-socket) $O(|V|)$ sequential memory access, and reading `ranks` and `out_degree` of ngh on Line 3 (highlighted in red) incurs $O(|E|)$ random memory

```
1  parallel_for (v ∈ V) {
2    for (ngh ∈ N⁻(v)) {
3      new_rank[v] += ranks[ngh] / out_degree[ngh];
4    }
5  }
```

Figure 2-1: The edge traversal code for one iteration of PageRank in the pull direction with frequent random memory access to ranks and out_degree highlighted in red.

access, because a node can have any other node as its neighbor. Note that, as an optimization, `ranks[`ngh`] / out_degree[`ngh`]` can be refactored into a loop that computes the value once per vertex so that the division is performed $O(|V|)$ instead of $O(|E|)$ times.

2.2.1 PageRank

2.2.2 Breadth-First Search

Breadth-first search is a graph traversal algorithm that visits all vertices in a graph in the order of their distance from the source vertex. Figure 2-2 shows the pseudocode for breadth-first search in the pull direction where unvisited vertices iterate through their incoming neighbors to find a parent that is on the active frontier. Reading the `frontier` array on Line 4 (highlighted in read) incurs random memory accesses as ngh can be any vertex. Unlike other algorithms, breadth-first search does not have to traverse all the edges in the pull mode; once a child vertex finds an active parent, the execution breaks out of the inner loop on Line 7, resulting in much lower random

```
1  parallel_for (v ∈ V) {
2    if (parent[v] < 0) {
3      for (ngh ∈ N⁻(v)) {
4        if (frontier[ngh]) {
5          parent[v] = ngh;
6          next_fronter[v] = 1;
7          break;
8        }
9      }
10   }
11 }
```

Figure 2-2: The edge traversal code for one iteration of breadth-first search in the pull direction with frequent random memory access to the frontier highlighted in red. All entries of the parent array are initialized to -1.

read traffic.

2.2.3 Connected Components

Connected components finds clusters in a graph where the vertices are connected. It can be implemented using label propagation, where unique IDs are assigned to vertices at the start. During each iteration, a vertex sets its ID to be the smallest of its neighbors' IDs and its own ID. The program terminates when all vertices stop updating their IDs. The resulting IDs array represents the components discovered; vertices in the same component are labeled with the same ID, and vertices in different components are labeled with different IDs. Figure 2-3 shows the edge traversal logic of label propagation. Reading the IDs array on Line 4 leads to frequent random memory accesses. Line 5 also reads IDs, but the corresponding entry should have been brought into the cache from the read on Line 4. Reading `frontier` on Line 3 can incur random memory access when $|V|$ is sufficiently large, but since `frontier` can be represented using a bit-vector, reading `frontier` causes much lower memory traffic than reading the IDs array.

The `frontier` array represents an active set of vertices whose IDs have changed during the previous iteration, so that the algorithm can operate only on the active set of vertices on each iteration. Maintaining such an active set introduces some random memory accesses, but can effectively filter out many random reads of the IDs array.

The result of label propagation is deterministic, as vertices in the same component will eventually converge to the same ID. However, the process of updating IDs is non-deterministic, as a vertex could see its neighbor's ID before or after that neighbor

```
1  parallel_for (v ∈ V) {
2    for (ngh ∈ N⁻(v)) {
3      if (frontier[ngh]) {
4        if (IDs[v] > IDs[ngh])
5          IDs[v] = IDs[ngh];
6      }
7    }
8  }
```

Figure 2-3: The edge traversal code for connected components using label propagation in the pull direction with frequent random memory access to the ID array highlighted in red.

updates its own ID for that iteration. Optimizations that affect the shared-memory communications between vertices could have a negative impact on the convergence rate.

Label propagation usually works well on power-law graphs with smaller diameters. On graphs with larger diameters and more regular degree distributions (e.g., road network graphs), parallel union-find [35] could perform much better. This thesis only focuses on optimizing the label propagation algorithm.

2.2.4 Single-Source Shortest Paths

Single-source shortest paths computes the shortest distance from a source vertex to each other vertex in the graph. There are two families of algorithms to solve this problem: label-setting (e.g., Dijkstra's algorithm [15]) and label-correcting (e.g., the Bellman-Ford algorithm [5]). Figure 2-4 shows the main edge traversal logic of the Bellman-Ford algorithm in the push direction. A frontier is used to keep track of the vertices whose distance from the source vertex changed during the previous iteration. Active vertices are examined during each iteration and their neighbors' distances are updated using the atomic instruction `writeMin` (Line 4). This operation results in random memory accesses. The Bellman-Ford algorithm traverses each edge at least once, causing enough memory traffic for cache and NUMA optimizations to be effective. Traversal can happen both in the push and pull directions.

```
1  parallel_for (v ∈ V) {
2    if (frontier[v]) {
3      for (ngh ∈ N+(v)) {
4        if (writeMin(&SP[ngh], SP[v] + ngh.weight))
5          next_frontier[ngh] = 1;
6      }
7    }
8  }
```

Figure 2-4: The edge traversal code for one iteration of Bellman-Ford single-source shortest paths in the push direction with frequent random memory access to the shortest path (SP) array highlighted in red. All entries of SP are initialized to infinity.

```
1  parallel_for (v ∈ V) {
2    for (ngh ∈ N⁻(v)) {
3      if (frontier[ngh])
4        new_rank[v] += ranks[ngh] / out_degree[ngh];
5    }
6  }
```

Figure 2-5: The edge traversal code for one iteration of PageRank-Delta in the pull direction with frequent random memory access to the frontier and the vertex data highlighted in red.

2.2.5 PageRank-Delta

PageRank-Delta is a variant of PageRank where only the vertices whose ranks have changed by more than a threshold are considered active for the next iteration. Traversal can happen in both the pull and push directions. Figure 2-5 shows PageRank-Delta in the pull direction. Line 3 checks if a source vertex is active before pulling its contribution. This check causes random memory accesses to read `frontier` in addition to the random reads of `ranks` and `out_degree`. However, since this check filters out many non-active vertices, the read traffic on Line 4 is significantly reduced. Cache and NUMA optimizations are expected to have less impact on PageRank-Delta compared to PageRank, as not all vertices are active during each round.

2.2.6 Collaborative Filtering

Collaborative Filtering is an algorithm widely used in recommender systems [31]. The main idea is to express each item and each user's rating as a combination of some latent features and then to perform Gradient Descent on these features. The

```
1  parallel_for (v ∈ V) {
2    for (ngh ∈ N⁻(v)) {
3      double estimate = 0;
4      int K = 20;
5      for ( int i = 0; i < K; i++)
6        estimate += latent_vec[ngh][i] * latent_vec[v][i];
7      double err = rating - estimate;
8      for ( int i = 0; i < K; i++)
9        error_vec[v][i] += latent_vec[ngh][i] * err;
10   }
11 }
```

Figure 2-6: The error vector computation for Collaborative Filtering in the pull direction with frequent random memory access to the latent vectors highlighted in red.

prediction of a user's rating on a particular item depends on the similarity between the user's features and the item's features. Collaborative Filtering can be modeled as a bipartite graph of users and items. As shown in Figure 2-6, an error vector is computed in $O(|E|)$ work and is later used to update the latent vectors. `K` is the dimension that we use for the features. Here we omit the code for updating latent vectors which takes $O(|V|)$ work. Random memory accesses can occur on Line 5 and Line 8 of Figure 2-6.

2.2.7 The Edge Traversal Abstraction

Ligra is a graph processing framework for shared memory focusing on graph traversal algorithms including the ones mentioned above. It provides a `VertexSubset` data structure to represent a subset of vertices in a graph, a `VertexMap` abstraction to apply vertex operations on vertices in a `VertexSubset`, and an `EdgeMap` abstraction to apply edge traversals on edges whose either or both ends are in a `VertexSubset`. These primitives simplify the implementations of many graph traversal algorithms, as these algorithms have the common pattern of iterating through (a subset of) vertices and their neighbors. The underlying implementation of a `VertexSubset` (i.e., sparse or dense) and the traversal direction (i.e., push or pull) depend on the size of the `VertexSubset` [2, 34]. We use `SparsePush` to denote traversals that iterate over the outgoing neighbors of each vertex on the frontier, and update the neighbors' values. `DensePull` iterates over the incoming neighbors of all vertices in a graph, and updates a vertex's own value. `DensePush` loops through all vertices and checks if each one is on the frontier instead of only looping over frontier vertices as in `SparsePush`. GraphIt also supports hybrid schedules such as `SparsePush-DensePush` and `SparsePush-DensePull`.

GraphIt has an algorithm language that is used to specify the high-level logic of graph traversal algorithms, as well as a scheduling language that is used to combine and fine-tune performance optimizations. GraphIt adopts Ligra's abstractions and lets programmers define two functions—`UpdateEdge` and `UpdateVertex`—in the algorithm language. The programmer can use these two functions to specify the op-

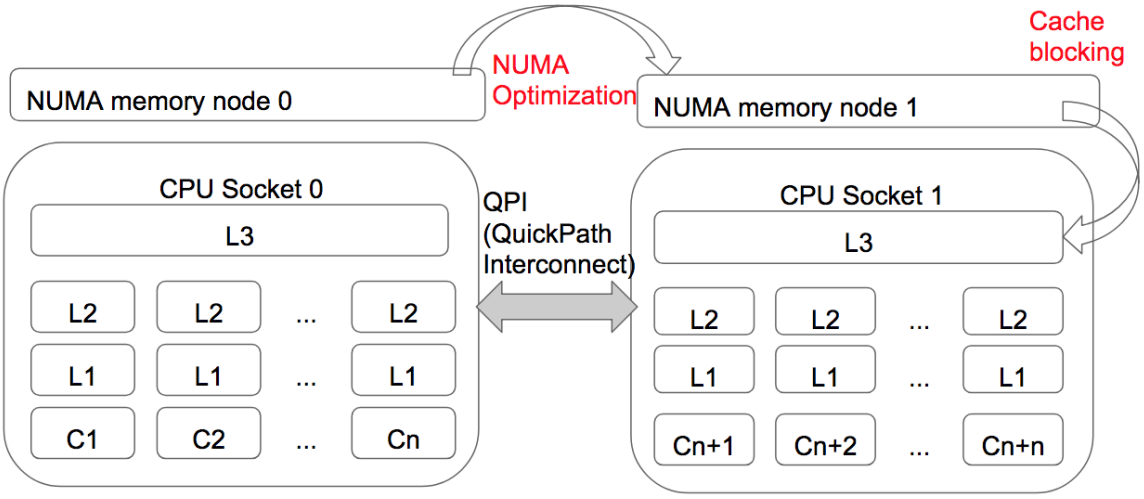

Figure 2-7: An illustrative example of the NUMA memory hierarchy. The boundaries of the two kinds of optimizations covered in this thesis (cache blocking and NUMA optimization) are marked in red.

erations on a single vertex/edge. GraphIt exposes a library function, `EdgeSetApply`, to iterate through the edges to perform `UpdateEdge` on each edge. Optimizations can be specified using GraphIt's scheduling language. These optimizations will be automatically generated as a part of `EdgeSetApply`, preserving any algorithmic simplicity.

2.3 NUMA Characteristics

Figure 2-7 shows an example of the NUMA memory hierarchy on a two-socket system. Each socket contains n cores and a memory node that is local to that socket. The link connecting the two sockets shaded in grey is the Intel QuickPath Interconnect (QPI) link. Memory accesses going through the QPI link take longer than the ones that can be satisfied locally [24]. While cache blocking optimizations, as highlighted in Figure 2-7, minimize the local DRAM accesses by fitting the desired data in the L3 cache, NUMA optimizations minimize the remote DRAM accesses by allocating the frequently accessed data local to the requesting cores.

This section first introduces three common NUMA allocation policies, and then

summarizes the NUMA characteristics of our system by collecting latency measurements from a micro-benchmark. We measure the running time ratios of the algorithms described in Section 2.2 using only remote or local memory.

2.3.1 NUMA Allocation Policies

When there is no explicit NUMA-aware memory control mechanism in place, Linux defaults to the *first-touch* policy where a page is allocated on the memory node local to the process that first uses that page. Under Linux's lazy allocation, first-touch happens when a memory location is first read or written rather than during the `malloc` call. This policy generally works fine in the absence of irregular memory access. However, a mismatch between allocation threads and processing threads can lead to poor performance. The first-touch policy can be especially harmful when graph loading is single-threaded. The burden of making sure that allocation threads and processing threads are from the same socket falls on the developers.

Alternatively, the `libnuma` library and the `numactl` control command allows interleaved allocation where memory is allocated in a round-robin fashion on the set of nodes specified. This policy balances memory access times among the cores, and generally improves performance on NUMA-oblivious graph processing frameworks such as Ligra [34] and Galois [28]. However, with the prior knowledge of the graph structure and the algorithm, one can explicitly confine memory accesses to be mostly NUMA-local.

NUMA-local processing requires allocating memory on a specific node or interleaved on a specific subset of nodes, and placing threads that access a memory region onto the same NUMA node as the memory region itself. NUMA-aware graph processing frameworks such as Polymer, Gemini, GraphGrind, and Grazelle all use this customized allocation strategy [18, 38, 43, 46].

```
1  int64_t *src = (int64_t *)malloc(arr_size * sizeof(int64_t));
2  parallel_for (i = 0; i < arr_size; i++) {
3    src[i] = i;
4  }
5  size_t *indices = (size_t *)malloc(iters * sizeof(size_t));
6  parallel_for (i = 0; i < iters; i++) {
7    indices[i] = (rnd ? hashInt(i) : i);
8  }
9  start_timer();
10 parallel_for (i = 0; i < iters; i++) reduction (+:sum) {
11   size_t index = indices[i];
12   sum += src[index];
13 }
14 end_timer();
```

Figure 2-8: The micro-benchmark that allocates, fills, and reads an array. Only reads are timed. Potential random memory access is highlighted in red. The `hashInt` function takes an integer and produces a hash value in a pseudorandom manner.

2.3.2 Micro-benchmark

We implement a micro-benchmark that allocates an array much larger than the system's last level cache (LLC). We measure the time it takes to sequentially or randomly read `iters` number of elements. Figure 2-8 shows the C++ code for this micro-benchmark. An array of random indices is generated on Line 7 if the `rnd` flag is set to `true`. Lines 10-13 sequentially read the index array and perform a random or sequential read on the `src` array.

Table 2.1 shows the total running times of reading a 2 GB array locally and remotely with sequential and random access patterns. We use `numactl -N` to control on which CPUs to execute the micro-benchmark and `numactl -m` to control on which NUMA nodes to allocate the memory. Note that random accesses still require sequential reads of the index array whose time is included in the measurements. Random memory access on average takes 5 times longer than sequential memory access, while remote memory access takes 3 times longer than local memory access.

To verify that our observations on the micro-benchmark also apply to graph al-

	Sequential Reads	Random Reads
Local Access Time (s)	0.56	2.93
Remote Access Time (s)	1.86	8.71

Table 2.1: The total running times in seconds of sequential and random memory reads of a 2 GB array under local and remote allocation policy using cores from one socket.

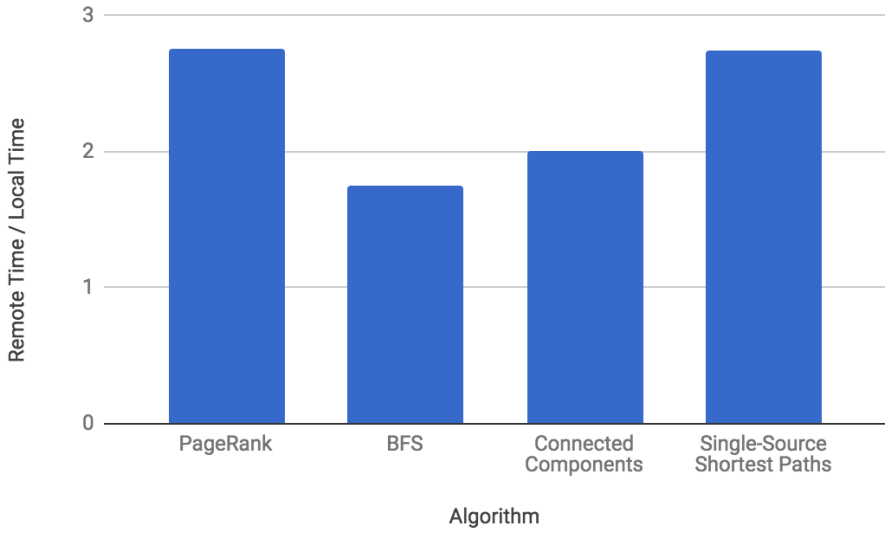

Figure 2-9: The ratios of various algorithms' running times using remote vs. local memory on the Twitter graph.

gorithms described in Section 2.2, we measure the running times of each algorithm using only remote vs. local memory. Their ratios measured on the Twitter graph are shown in Figure 2-9. Unsurprisingly, algorithms using only local memory are 1.8 to 2.8 times faster than algorithms using only remote memory.

2.4 The GraphIt Compiler

The work in this thesis contributes to GraphIt, an ongoing effort to develop a simple and high-performing DSL for graph processing. Simplicity in GraphIt is achieved through separating the high-level graph traversal logic from the algorithm-specific, graph-specific, and hardware-specific optimizations. High performance is achieved through the creation of an expressive scheduling language that allows the programmer to compose and fine-tune various optimizations to find the optimal schedules for a particular algorithm running on a specific graph and hardware. The GraphIt compiler has three major components: a front-end that scans and parses the traversal algorithms and optimization schedules provided by the programmer, a mid-end that interprets the schedules via multiple lowering passes, and a back-end that generates high-performance C++ code with the specified optimizations. This section gives a

Apply Scheduling Functions	Descriptions
`program->configApplyDirection(label, config);`	Config options: **SparsePush**, DensePush, DensePull, SparsePush-DensePull, SparsePush-DensePush
`program->configApplyParallelization(label, config, [grainSize], [direction]);`	Config options: **serial**, dynamic-vertex-parallel, static-vertex-parallel, edge-aware-dynamic-vertex-parallel, edge-parallel
`program->configApplyDenseVertexSet(label, config, [vertex set], [direction]);`	Vertex set options: **both**, src vertex set, dst vertex set Config Options: **bool-array**, bit-vector
`program->configApplyNumSSG(label, config, numSegments, [direction]);`	Config options: **fixed-vertex-count** or edge-aware-vertex-count
`program->configApplyNUMA(label, config, [direction]);`	Config options: **serial**, static-parallel, dynamic-parallel
`program->fuseFields({vect1, vect2, ...});`	Fuses multiple arrays into a single array of structs.

Table 2.2: GraphIt's Scheduling functions. The default option for an operator is shown in bold. Optional arguments are shown in [ ]. If the optional direction argument is not specified, the configuration is applied to all relevant directions. We use a default grain size of 256 for parallelization.

brief overview of each of the three compiler components.

2.4.1 Front-End Schedules

GraphIt reuses the front-end of Simit, a DSL for physical simulation [20], to handle parsing, tokenizing, and semantic analysis. GraphIt's scheduling language exposes a family of C-like function calls with the `configApply` prefix followed by the name of a specific optimization and function arguments. For example, `configApplyNUMA` enables NUMA optimization and `configApplyDirection` enables the traversal direction optimization. Table 2.2 shows the front-end scheduling API with a complete set of optimizations that GraphIt supports, including the cache and NUMA optimizations implemented as a part of this thesis.

The front-end reads in these C-like `configApply` functions and constructs a `schedule` object that contains three types of information: the physical layout of the vertex data (i.e., an array of structs or a struct of arrays), the edge traversal optimizations (e.g., traversal direction and cache/NUMA optimizations), and the frontier data structures (i.e., sliding queue, boolean array, or bit-vector). The mid-end lowering passes heavily rely on information stored in this `schedule` object.

2.4.2 Mid-End Lowering

The mid-end consists of a few optimization passes which transform the `schedule` object into a mid-end intermediate representation (MIR) that is used by back-end code generation. In particular, the `ApplyExprLower` pass is mainly responsible for transforming edge traversal optimizations including traversal direction (push or pull), deduplication (enabled or disabled), the frontier data structure (sliding queue, boolean array, or bit-vector), and whether the traversal happens in parallel. As we will see in Chapter 3 and 4, the work in this thesis enables the `ApplyExprLower` pass to transform cache optimizations and adds an additional pass (`MergeReduceLower`) to transform NUMA optimizations.

These mid-end lowering passes generally perform one or both of the following two tasks: gathering any global information associated with this pass into the MIR context, and modifying the local properties of existing MIR nodes. An MIR node is an abstract structure which can represent an expression, a statement, a type, the domain of a for loop, a function declaration, or a variable declaration. For example, `EdgeSetApplyExpr` is an MIR node that stores node-local information such as whether a traversal should enable deduplication. GraphIt borrows the idea of the visitor design pattern, where an additional visitor class is used to implement the appropriate specializations in each MIR node. Each of the lowering passes constructs an MIR visitor or rewriter, gets a list of functions from the MIR context, and follows the visitor pattern to iterate through the functions and modify the corresponding MIR nodes according to the `schedule` object obtained from the front-end processing.

2.4.3 Back-End Code Generation

After the mid-end lowering passes have gathered optimization information in the MIR context and the MIR nodes, the MIR nodes are visited again for code generation. When the `EdgeSetApplyExpr` node is visited, for example, the compiler back-end generates C++ code with OpenMP or Cilk parallel primitives if the `is_parallel` flag in the `EdgeSetApplyExpr` node has been set to true.

Chapter 3

Cache Blocking in GraphIt

In this chapter, we present the integration of cache blocking in the GraphIt compiler. Cache blocking [29, 41, 42] is a technique developed for sparse matrix-vector multiplication (SpMV) to reduce instances of DRAM access by dividing the matrix into blocks that fit in cache. We apply this technique to graph processing by partitioning a graph into subgraphs whose vertex data can fit in the last level cache (LLC). We first develop hand-optimized C++ implementations with cache blocking and then enhance the GraphIt scheduling language and compiler to automatically generate the code. Section 3.1 explains our C++ implementation of cache blocking in detail, and Section 3.2 demonstrates how the cache-optimized programs are generated by the GraphIt compiler.

3.1 Implementation

3.1.1 Graph Partitioning

As in Polymer [43], Cagra [44], and Gemini [46], we partition the graph topology data (the offset array and the edge array in CSR format) by source or by destination, based on the traversal direction. Vertices are first divided into s disjoint regions $(R_0, R_1, \ldots, R_{s-1})$, which correspond to the range of source vertices in the pull mode or destination vertices in the push mode. In the pull mode, incoming edges $E_{dst<-src}$

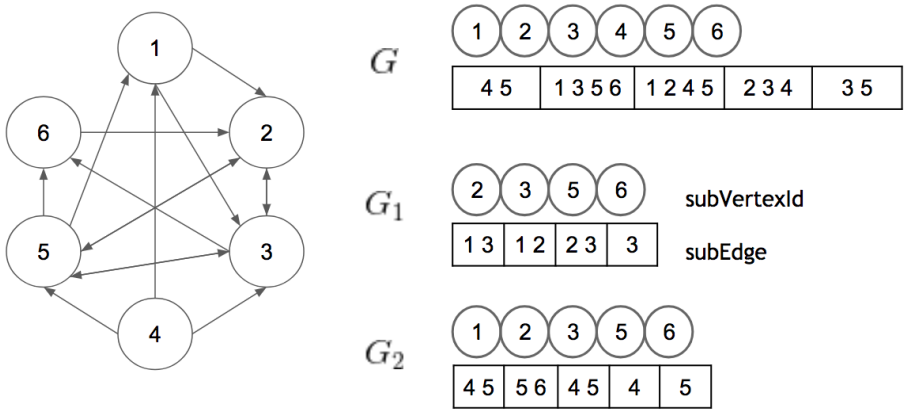

Figure 3-1: An example of the graph-partitioning strategy that GraphIt uses. G is partitioned in the pull direction. Subgraph G_1 is assigned source vertices 1–3, and G_2 is assigned source vertices 4–6. `subVertexId` represents the home vertices on each subgraph. Note that vertex 2, 3, 5, and 6 are duplicated across the two subgraphs.

sorted by *dst* are assigned to subgraph G_i if $src \in R_i$. In the push mode, outgoing edges $E_{src->dst}$ sorted by *src* are assigned to subgraph G_i if $dst \in R_i$. We then allocate three arrays for G_i to store this subgraph's topology: `subOffset`, `subVertexId`, and `subEdge`. `subOffset` stores the offsets into `subEdge`, as in the traditional CSR format. `subEdge` only stores edges within R_i. `subVertexId` maps the index of `subOffset` to the actual vertex IDs in the original graph. Figure 3-1 visualizes this partitioning strategy.

When processing a subgraph G_i, we only iterate through vertices that are present in G_i. A vertex can be present in multiple subgraphs since its incident edges are partitioned. We define the duplication factor, α, to be the ratio of the total number of home vertices across all subgraphs over the vertex count in the original graph ($\alpha = \frac{\sum_{i=0}^{s} |V_i|}{|V|}$). α increases as the number of subgraphs s increases. The value of s controls the range of random memory accesses, which decreases as s increases. Intuitively, s should be as small as possible while ensuring the vertex data associated with range R_i can fit into the LLC. We use $s = 16$ on Twitter and WebGraph, and $s = 30$ on Friendster. Note that this value depends on the cache size and can be fine-tuned. We also eliminate zero-degree vertices when constructing the subgraphs to reduce the total work. There is another advantage to this partitioning strategy: if

the edge array of the original CSR is sorted, the partitioned edge arrays will still be sorted as we sequentially distribute the edges among the subgraphs. This preserves any existing locality.

3.1.2 Subgraph Processing

In blocked subgraph processing, all threads process one subgraph together at a time (no parallelism across subgraphs) to maximize parallelism. As a result, there is no need for additional synchronization, atomic instructions, or intermediate buffering. For example, PageRank in the pull direction performs blocked reads on the `contrib` array but writes directly to the global `new_rank` array. Since writes happen after all reads and each thread writes to a different location, there are no write conflicts. However, the writes to `new_rank` can incur random memory access, as `dst` is not continuous in each subgraph. An alternative approach would be to sequentially write the new ranks from each subgraph into a subgraph-local buffer and merge the buffers in a cache-aware fashion at the end of each iteration, as described in [44]. This approach requires the start and end index of each merging block to be precomputed and does not perform well on all graphs according to our experiments. Hence we choose not to include this optimization in our final implementation.

Figure 3-2 shows PageRank's `EdgeSetApply` function in the pull direction with cache blocking enabled. We associate two library functions—`getNumSubgraphs(label)` and `getSubgraph(subgraphId)`—with the original graph g, and a `numVertices` field with a subgraph sg which is equivalent to $|V_i|$. Lines 1-2 iterate through the subgraphs

```
1  for (int subgraphId = 0; subgraphId < g.getNumSubgraphs("s1"); subgraphId++) {
2    auto sg = g.getSubgraph(subgraphId);
3    parallel_for (int localId = 0; localId < sg.numVertices; localId++) {
4      int dst = sg.subVertexId[localId];
5      for (int ngh = sg.subOffset[localId]; ngh < sg.subOffset[localId+1]; ngh++) {
6        int src = sg.subEdge[ngh];
7        new_rank[dst] += contrib[src];
8      }
9    }
10 }
```

Figure 3-2: The C++ implementation for PageRank's `EdgeSetApply` function with cache blocking where blocked reads are highlighted in red.

associated with the edge traversal labeled "s1". Lines 3-7 iterate through vertices and edges of a subgraph and update the global `new_rank` array. Note that reading `contrib[src]` (highlighted in red) is blocked but writing to `new_rank[dst]` is not. We expect the C++ compiler to create a temporary variable for `new_rank[dst]` to accumulate the sum for a destination vertex and only to perform one random write at the end rather than actually writing to `new_rank[dst]` during every iteration of the inner loop. Therefore, reading `contrib` happens $O(|E|)$ times, whereas updating `new_rank` only happens $O(\alpha|V|)$ times. The overall running time is dominated by the read traffic, which our approach is designed to optimize.

3.2 Compiler Integration

3.2.1 Scheduling Language Interface

We extend GraphIt's scheduling language to support `configApplyNumSSG(label, config, numSegments, [direction])` where `SSG` stands for segmented subgraphs. `label` identifies the `EdgeSetApply` function to which cache blocking is applied. `config` specifies the partitioning strategy of subgraphs (e.g., edge-based or vertex-based), although we currently only support vertex-based approaches. `numSegments` corresponds to the number of subgraphs into which to partition the original. `direction` corresponds to whether cache blocking is applied to the push or pull direction during graph traversal (defaults to pull). When `configApplyNumSSG` is present, the number of subgraphs and direction information are stored in the front-end `schedule` object for later use in the middle-end lowering and back-end code generation. Graph partitioning is implemented in the runtime library in the same way as described in Section 3.1.1. Two library functions, `BuildPullSubgraphs` and `BuildPushSubgraphs` are exported to build the subgraphs.

3.2.2 Schedule Lowering and Code Generation

During the `ApplyExprLower` pass, we check to see if `configApplyNumSSG` is present among the input schedules, and if so, store the number of subgraphs (specified by the input `numSegments`) into the `EdgeSetApplyExpr` node (defined in Section 2.4.2) so that it can be retrieved when the `EdgeSetApplyExpr` node is visited during the code generation phase. We generate the call to `BuildPullSubgraphs` or `BuildPushSubgraphs` in the body of the `main` function as a part of the setup procedure. The resulting subgraphs are stored along with the original graph so they can be accessed during edge traversals. The code generation phase checks the number of subgraphs stored in the `EdgeSetApplyExpr` node, and generates the cache-blocking version of the `EdgeSetApply` function if the number of subgraphs is greater than one.

Chapter 4

NUMA-Awareness in GraphIt

This chapter presents the integration of NUMA optimizations in GraphIt. By carefully partitioning an input graph into subgraphs and binding the subgraphs and their processing threads to the same NUMA node, GraphIt significantly reduces cross-socket memory accesses. Section 4.1 describes the steps involved in manually optimizing graph algorithms to reduce remote accesses. Section 4.2 presents additional optimizations that we attempted but did not integrate into the compiler for various reasons. Section 4.3 explains how we enhance the GraphIt compiler to automatically generate the hand-optimized programs presented in Section 4.1.

4.1 Implementation

4.1.1 Subgraph Allocation

After partitioning an input graph into a set of subgraphs the same way as in cache blocking (Section 3.1.1), we want to allocate each subgraph (the subOffset, subVertexID, and subEdge arrays) once and avoid migrating a subgraph from one NUMA node's memory to another. To do so, we bind each subgraph to a specific socket using `numa_alloc_onnode(size_t size, int node)` provided by the `libnuma` library. Here `size` is dependent on the number of vertices and edges of a particular subgraph, and `node` is set to be `subgraphId` % `numSockets`. In this way, subgraphs are bound

to NUMA nodes in an interleaved fashion. The `numa_alloc_onnode` function is relatively slow compared to the `malloc` family of functions; hence, we only perform these allocations once at the beginning when we build the subgraphs. Memories allocated using `numa_alloc_onnode` need to be freed by `numa_free` at the end of processing.

4.1.2 Thread Placement

The major implementation challenge that we faced in making graph algorithms NUMA-aware was in explicitly placing threads onto the same NUMA node as the subgraph that these threads are responsible for processing. Even though the `taskset` family of Linux commands and the `numa_run_on_node` family of library calls from `libnuma` provide an interface to control processor affinity at our desired level, they do not take into account the dynamic task scheduling mechanism provided by Cilk and OpenMP. Cilk and OpenMP, on the other hand, do not support NUMA-aware scheduling or work stealing (i.e., stealing from workers on the same NUMA node first). Existing NUMA-aware graph processing systems have spent a substantial amount of effort to overcome this limitation. For example, Polymer uses `pthread`, a do-it-yourself low-level concurrency platform that requires programmers to marshal arguments to thread creation, which greatly complicates the underlying Ligra framework [43]. Gemini computes the start and end index of every task, manually assigns tasks to workers, and implements NUMA-local work-stealing [46]. GraphGrind makes changes to the Cilk runtime system to achieve NUMA-aware task scheduling [38]. Inspired by the systems above, we explore three ways to implement NUMA-aware thread placement using existing concurrency platforms without making changes to the platforms themselves.

On-the-Fly Thread Placement. The easiest way to make sure that remote memory accesses are minimized is to bind a spawned thread on-the-fly to the same NUMA node as the subgraph it is about to process. This only requires a one-line change (Line 5 of Figure 4-1) from the cache-optimized PageRank. `get_socket_id` on Line 3 is a library function to retrieve the predetermined socket on which a subgraph should be processed. The cost of `numa_run_on_node`, however, turns out to be too high when

```
1  for (int subgraphId = 0; subgraphId < g.getNumSubgraphs("s1"); subgraphId++) {
2    auto sg = g.getSubgraph(subgraphId);
3    int socketId = sg.get_socket_id();
4    parallel_for (int localId = 0; localId < sg.numVertices; localId++) {
5      /* Place the thread to the same socket as the subgraph. */
6      numa_run_on_node(socketId);
7      int dst = sg.subVertexId[localId];
8      for (int ngh = sg.subOffset[localId]; ngh < sg.subOffset[localId+1]; ngh++) {
9        int src = sg.subEdge[ngh];
10       local_new_rank[socketId][dst] += contrib[src];
11     }
12   }
13 }
```

Figure 4-1: The C++ PageRank code for binding threads to sockets on-the-fly to avoid remote memory access.

invoked every time a worker thread is spawned, outweighing the benefit of reduced cross-socket memory access.

Static Range Assignment. One workaround to avoid on-the-fly thread placement overhead is to statically bind threads to sockets prior to processing (the same way that subgraphs are allocated). The downside of this approach is that the processing range of each thread will also have to be static. Figure 4-2 shows the version of NUMA-aware PageRank using this approach. More specifically, on our system, with 48 threads and 2 NUMA sockets, threads 0-23 are bound to socket 0 and 24-47 to socket 1 during initialization. Lines 10-11 of Figure 4-2 compute the start and end index of this thread's task based on its thread ID and the total numbers of threads and sockets in the system. This approach does not allow intra-socket work-stealing since the start and end index are statically computed. Our experiments show that this implementation results in a 60% speedup on synthetic graphs (i.e., uniform [17] and kronecker graphs [22]), but there is a 50% slowdown on the Twitter graph because each thread is doing a different amount of work, which leads to load imbalance. The overall performance is dominated by the straggler threads.

Dynamic Affinity Control. Even though OpenMP does not provide a locality-aware work-scheduling mechanism, it has the important `PLACES` abstraction: the environment variable `OMP_PLACES` describes how threads are bound to hardware. Like `taskset`, `OMP_PLACES` specifies on which CPUs the threads should be placed, with the main differences being that `OMP_PLACES` can be specified using an abstract name and

```
1  threads = numa_num_configured_cpus();
2  sockets = numa_num_configured_nodes();
3  threadsPerSocket = threads / sockets;
4
5  #pragma omp parallel num_threads(threads) {
6    int threadId = omp_get_thread_num();
7    int socketId = threadId / threadsPerSocket;
8    auto sg = g.getSubgraphFromSocket(socketId);
9    int verticesPerThread = sg.numVertices / threadsPerSocket;
10   int startIndex = (threadId % threadsPerSocket) * verticesPerThread;
11   int endIndex = startIndex + verticesPerThread;
12
13   for (int localId = startIndex; localId < endIndex; localId++) {
14     int dst = sg.subVertexId[localId];
15     for (int ngh = sg.subOffset[localId]; ngh < sg.subOffset[localId+1]; ngh++) {
16       int src = sg.subEdge[ngh];
17       local_new_rank[socketId][dst] += contrib[src];
18     }
19   }
20 }
```

Figure 4-2: The C++ PageRank code for computing the start and end index for each worker thread to avoid on-the-fly thread placement.

integrates well with other OpenMP library calls such as `proc_bind`. The abstract names for the values of `OMP_PLACES` can be one of the following:

- `threads`: each place corresponds to a single hardware thread
- `cores`: each place corresponds to a physical core
- `sockets`: each place corresponds to a single NUMA socket

`OMP_PLACES` specifies the granularity of thread placement when combined with the `proc_bind` directives for parallel processing. The `proc_bind` clause specifies the affinity policy of a pool of worker threads and can take in one of the following values as input:

- `spread`: spreads threads among places as far away as possible
- `close`: iterates through places and assigns one thread to one place in order
- `master`: places the child threads into the same place as the parent thread

Figure 4-3 and Figure 4-4 illustrates the `spread` and the `close` placement strategy. To make edge traversals NUMA-aware, we set `OMP_PLACES = sockets`. The code for PageRank using `OMP_PLACES` and `proc_bind` to control processor affinity is shown in Figure 4-5. Line 1 enables nested parallelism in OpenMP. Line 3 spawns a number of threads equal to the number of sockets using the `spread` or `close` affinity policy, so that each thread is placed into a different socket. Each of these threads iterates

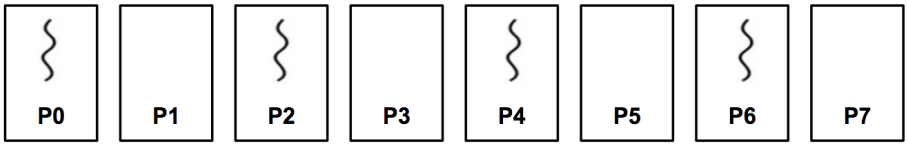

Figure 4-3: A visualization of the `spread` placement strategy where 4 threads are spread among 8 places (P0,. . . ,P7) as far away from each other as possible.

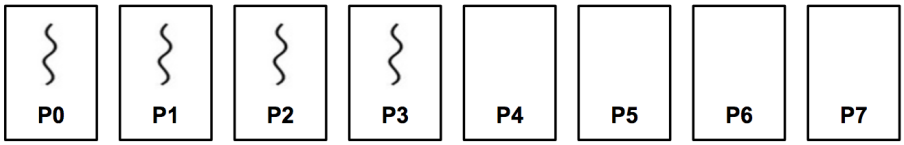

Figure 4-4: A visualization of the `close` placement strategy where 4 threads are assigned to 8 places (P0,. . . ,P7) in order.

through the subgraphs on that socket (Lines 5-7), and spawns a number of threads that is equal to the number of processors on that socket. These newly spawned threads are bound to the same socket as their parent using the `master` affinity policy (Line 9). Lines 10-15 dynamically schedule the pinned threads to process subgraphs on the same socket as the threads themselves. Note that `parallel_for` from Figure 3-2 is replaced by `#pragma omp for` since `proc_bind` is an OpenMP functionality. Unlike the on-the-fly thread placement strategy where each thread is assigned to a socket based on its thread ID, `proc_bind`, using the `master` policy, directly confines the affinity of processors in a socket, avoiding unnecessary thread migration. By not having to statically assign processing ranges to workers, dynamic affinity control allows work-stealing within a subgraph, and therefore has a better performance.

4.1.3 Broadcast and Merge Phases

Unlike with cache blocking, where only a single thread can write to a specific entry of the global `new_rank` array at once, NUMA optimizations can result in two or more threads updating a vertex's data simultaneously because subgraphs on different sockets are processed in parallel. If this is not a benign race, we need to either use atomic instructions to update the global array directly, or to buffer updates locally on each socket and merge the result at the end of each iteration. We found that the

```
omp_set_nested(1);
int numPlaces = omp_get_num_places();
#pragma omp parallel num_threads(numPlaces) proc_bind(spread) {
  int socketId = omp_get_place_num();
  for (int i = 0; i < (g.getNumSubgraphs("s1") + numPlaces - 1) / numPlaces; i++) {
    subgraphId = socketId + i * numPlaces;
    auto sg = g.getSubgraph(subgraphId);
    int threadsPerSocket = omp_get_place_num_procs(socketId);
    #pragma omp parallel num_threads(threadsPerSocket) proc_bind(master) {
      #pragma omp for schedule(dynamic, 1024)
      for (int localId = 0; localId < sg.numVertices; localId++) {
        int dst = sg.subVertexId[localId];
        for (int ngh = sg.subOffset[localId]; ngh < sg.subOffset[localId+1]; ngh++) {
          int src = sg.subEdge[ngh];
          local_new_rank[socketId][dst] += contrib[src];
        }
      }
    }
  }
}
```

Figure 4-5: The C++ code for PageRank's `EdgeSetApply` with NUMA optimization using dynamic affinity control.

overhead of atomic instructions outweighs the benefits from NUMA optimizations. Hence, we create socket-local buffers (e.g., `local_new_rank` in Figure 4-5) to store the partial updates from each subgraph and merge them in the end. The implementation of the merge phase of PageRank is shown in Figure 4-6. Note that the inner loop serially iterates through all the sockets. Therefore, the total amount of work from the merge phase increases as the number of sockets increases.

Since `new_rank` is stateful across iterations, the entries of `new_rank` need to be copied into each `local_new_rank` before traversing the edges. We call this the broadcast phase. Note that the cost of the broadcast phase is also proportional to the number of sockets.

```
parallel_for (int n = 0; n < |V|; n++) {
  for (int socketId = 0; socketId < omp_get_num_places(); socketId++) {
    new_rank[n] += local_new_rank[socketId][n];
    local_new_rank[socketId][n] = 0;
  }
}
```

Figure 4-6: The merge phase of PageRank with NUMA optimizations.

4.2 Additional Optimizations

This section describes two additional optimizations that we implemented in an attempt to further improve the performance of NUMA-aware graph algorithms. Inter-socket work-stealing is designed to overcome workload imbalance in highly skewed graphs. Sequential buffered writes and cache-aware merge are our attempts to further reduce random memory access during edge traversal and the merge phase. The benefit of these optimizations turned out to be small, and so we did not integrate these two optimizations into the compiler. However, this could be done in future work.

4.2.1 Inter-Socket Work-Stealing

Figure 4-5 shows that threads in one socket only process the subgraphs allocated on that socket. In the case where one socket finishes executing all of its subgraphs, the threads in that socket will be busy waiting rather than helping other sockets process their subgraphs. Previous work has shown that perfect CPU load balance is hard to achieve no matter what graph-partitioning strategy is used [38]. Our graph-partitioning strategy described in 3.1.1 is locality-preserving, but does not provide any guarantees on load balance.

We implement coarse-grained inter-socket work-stealing at the subgraph level to accommodate any load imbalance from graph partitioning. We store the subgraphs in `unprocessedSubgraphs`, an array of linked lists indexed by `socketId`. The `getSubgraph` function is modified to take in `socketId` as an argument, and returns the next subgraph on that socket's queue or steals a subgraph from the next socket if its own queue is empty. These operations are performed using atomic instructions. The C++ implementation of the `getSubgraph` function is shown in Figure 4-7.

4.2.2 Sequential Writes and Cache-Aware Merge

One other issue with NUMA optimizations using dynamic affinity control is that updating `local_new_rank[socketId][dst]` after the inner-most loop finishes executing

```
1  SubGraph *getSubGraph(int socketId) {
2    /* Try socketId's queue first. If empty, try other sockets */
3    auto sg = getHeadOrNull(socketId);
4    if (sg)
5      return sg;
6    int victimId = (socketId + 1) % numSockets;
7    while (victimId != socketId) {
8      sg = getHeadOrNull(victimId);
9      if (sg)
10       return sg;
11     victimId = (victimId + 1) % numSockets;
12   }
13   return NULL;
14 }
15
16 SubGraph *getHeadOrNull(int socketId) {
17   auto head = unprocessedSubgraphs[socketId];
18   while (head) {
19     auto next = head->next;
20     if (__sync_bool_compare_and_swap(&unprocessedSubgraphs[socketId], head, next))
21       return head;
22     head = unprocessedSubgraphs[socketId];
23   }
24   return NULL;
25 }
```

Figure 4-7: The helper functions to retrieve the next unprocessed subgraph. A socket's local queue is checked first before stealing from another socket. The atomic instruction on Line 20 is necessary since more than one thread can be accessing a socket's queue at the same time.

still incurs socket-local random writes, as `dst` is not guaranteed to be continuous in each subgraph. One way to make sure that writes to `local_new_rank` are sequential is to compress `local_new_rank` from size $|V|$ to size $|V_i|$. Then we can use `localId` instead of `dst` to index into the `local_new_rank` array and the writes will be sequential.

The downside of having a `local_new_rank` whose size is different from the global `new_rank` array is the increased overhead during the merge phase: the merge step needs to iterate through each V_i, translate the `localId` to the actual destination ID, and collect the new values in the global `new_rank` array. Writing to the `new_rank` array incurs many (cross-socket) random memory accesses. We implement the *cache-aware merging* technique introduced by Zhang et al. [44]. Instead of writing all of G_i's `local_new_rank` entries to `new_rank` at once (which has a large random access range), we write a small range of vertices from each subgraph before moving on to writing the next range. The range is determined in a way that vertices within that range can fit into the L1 cache. As a result, random DRAM accesses can be greatly

reduced.

4.3 Compiler Integration

4.3.1 Scheduling Language Interface

We extend the scheduling language interface of GraphIt to support `configApplyNUMA(label, config, [direction])` where `label` identifies the `EdgeSetApply` function to which NUMA optimizations are applied. `Config` controls whether or not inter-socket work-stealing is enabled (not fully supported yet). `ConfigApplyNUMA` is only valid when `configApplyNumSSG` is also configured, as NUMA optimizations require graph partitioning. When `configApplyNUMA` is present, the NUMA flag in the front-end `schedule` object is set to `true`, and the direction information is stored for later use. As with cache blocking, `direction` indicates whether the subgraphs should be built in the pull or push direction. The library functions `BuildPullSubgraphs` and `BuildPushSubgraphs` are extended to take an additional default argument: `numa_aware`. If `configApplyNUMA` is present in the specified schedules, `numa_aware` will be set to `true`, and subgraphs will be bound to specific sockets as described in Section 4.1.1.

4.3.2 Merge-Reduce Lowering Pass

We add a `MergeReduceLower` pass in the middle-end of the GraphIt compiler to perform the following tasks:

- Extract the reduction operator for the merge phase (e.g., `+=` in PageRank and `MIN` in label propagation).
- Add the `socketId` argument to the declaration and invocation of the `updateEdge` function.
- Modify the write target in `updateEdge` from global vertex data array to local buffers.

In the `MergeReduceLower` pass, we first create an `ApplyExprVisitor` following the visitor patterns introduced in 2.4 to revisit the `EdgeSetApplyExpr` expression

node. The `EdgeSetApplyExpr` node stores the information related to the edge traversal, such as the identifiers for the `updateEdge` function and the target vertex data array. We extend the `EdgeSetApplyExpr` node to store an auxiliary data structure—`MergeReduceField`—and we extract the type and variable name of the vertex data array and the reduction operator into this data structure. Since the `updateEdge` function is available during the `MergeReduceLower` pass, we also modify its declaration to take the `socketId` as an additional argument. While visiting the `EdgeSetApplyExpr` node, we create another visitor, `ReduceStmtVisitor`, to visit the `ReduceStmt` statement. We modify the field name of the left-hand-side expression (the target vertex data array) by appending `local_` to the front of the original array name when visiting `ReduceStmt`, so that the `updateEdge` function writes to the local buffers instead of the global vertex data array.

4.3.3 Code Generation

During code generation, we again generate the function call to build subgraphs in the generated `main` function. In addition, we also generate the allocation and deallocation for all socket-local buffers. When visiting the `EdgeSetApplyExpr` node, we look at its `MergeReduceField`, and generate the merge phase using the stored operator. We modify the generated C++ code to use the dynamic affinity control logic as specified in Section 4.1.2.

Chapter 5

Evaluation

5.1 Experimental Setup

5.1.1 Datasets

We evaluate the effect of cache and NUMA optimizations on graphs whose vertex data does not fit into the LLC. Table 5.1 lists our input datasets and their corresponding sizes. LiveJournal [13], Twitter [21], and Friendster [23] are three social network graphs with power-law degree distributions. Friendster is special because its number of edges does not fit into a 32-bit signed integer. USAroad [14] is a mesh network with small and undeviating degrees. The WebGraph [27] is from the 2012 Common Crawl. The Netflix dataset and its synthesized expansion (Netflix2x) [6, 25] are only

Dataset	Number of Vertices	Number of Edges	Average Degree
LiveJournal (LJ) [13]	5 M	69 M	14
Twitter (TW) [21]	61.6 M	1.47 B	24
WebGraph (WB) [27]	101 M	2.04 B	20
USAroad (RD) [14]	24 M	58 M	2.4
Friendster (FT) [23]	124.84 M	3.6 B	29
Netflix (NX) [6]	0.5 M	198 M	396
Netflix2x (NX2) [25]	1 M	792 M	792

Table 5.1: The number of vertices, edges, and average degrees of the input graphs. M stands for millions. B stands for billions. The number of edges of LiveJournal and Friendster are doubled as they are undirected. All other graphs are directed.

used to evaluate collaborative filtering.

5.1.2 Algorithms

We evaluate the performance of GraphIt and other frameworks on six graph traversal algorithms: 20 iterations of PageRank (PR), breadth-first search (BFS), connected components (CC) with synchronous label propagation, single-source shortest paths (SSSP) with frontier-based Bellman-Ford algorithm, 10 iterations of PageRank-Delta (PRDelta), and collaborative filtering (CF). The details of each algorithm can be found in Section 2.2. The experiments for BFS and SSSP choose 10 starting points and take the average. SSSP uses `SparsePush-DensePush`. BFS, CC, and PRDelta use `SparsePush-DensePull`. PR and CF use `DensePull`.

5.1.3 Hardware Configuration

We use a dual-socket system with Intel Xeon E5-2695 v3 CPUs and 12 cores each for a total of 24 cores and 48 hyper-threads. The system has 128GB of DDR3-1600 memory and 30 MB last level cache on each socket, and runs with Transparent Huge Pages (THP) enabled.

5.2 Cache Blocking Performance

5.2.1 Overall Speedups

Figure 5-1 shows the speedups from cache blocking over the existing best schedules for each algorithm. Figure 5-2 shows the factors by which cache misses are reduced. Note that the performance speedup is highly correlated to the reduction of cache misses. We apply cache blocking when the frontier is dense, and other schedules are kept the same as before. As a result, cache optimizations help with traversals in the `DensePush` direction for single-source shortest paths and in the `DensePull` direction for all the other algorithms. Cache blocking improves the performance of PageRank, connected components, single-source shortest paths, PageRank-Delta, and

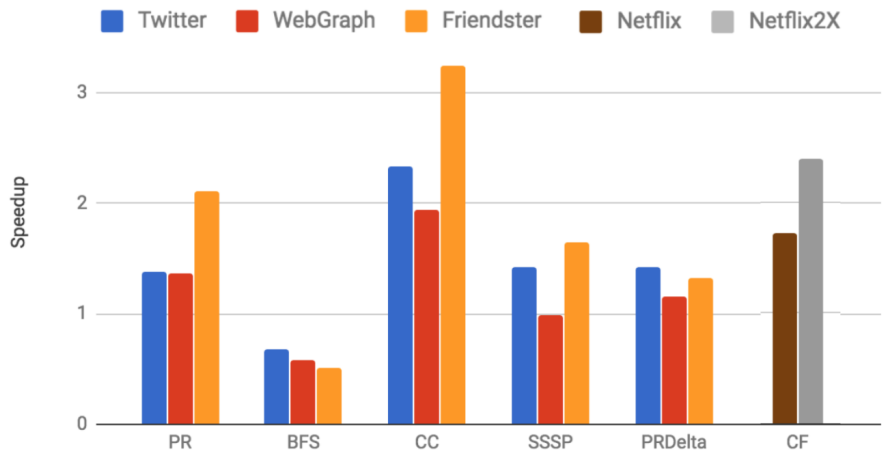

Figure 5-1: The speedups from applying cache blocking to the six algorithms. Twitter and WebGraph are partitioned into 16 subgraphs. Friendster is partitioned into 30 subgraphs. Netflix and Netflix2X are partitioned into 10 subgraphs.

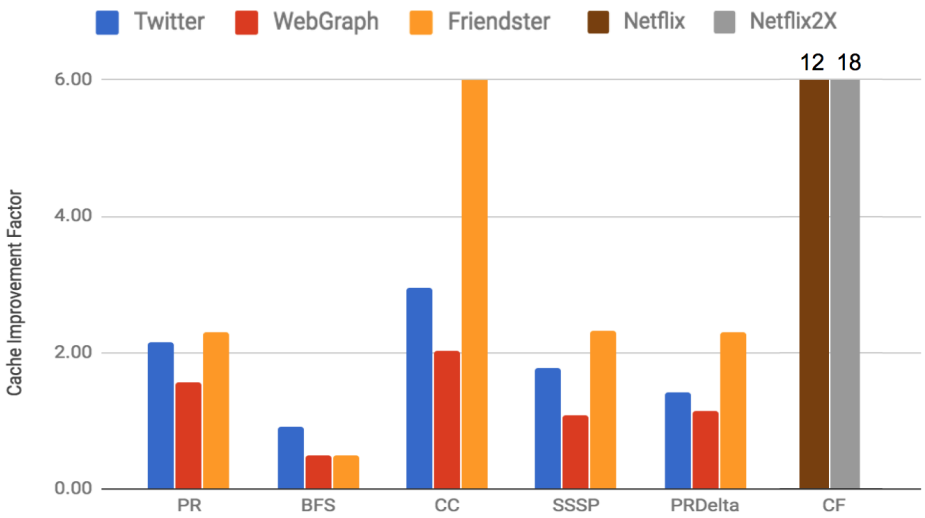

Figure 5-2: The factors by which cache misses are reduced after applying cache blocking.

collaborative filtering by up to 3X, but hurts the performance of breadth-first search. Cache blocking improves the cache hit rate and the overall performance of connected components by a larger factor than PageRank, because in addition to the random accesses to the application data, connected components also incurs random memory reads to the frontier, so cache blocking has a bigger impact.

	Twitter (16)	WebGraph (16)	Friendster (30)
$\|V\|$	61.58M	101.72M	124.84M
$\sum_{i=0}^{s} \|V_i\|$	188.16M	312.32M	602.27M
α	3.06	3.07	4.82

Table 5.2: The original vertex counts, the total numbers of home vertices across subgraphs, and the duplication factors of Twitter, WebGraph, and Friendster when partitioned into 16, 16, and 30 subgraphs.

5.2.2 Read and Write Trade-offs of Cache Blocking

Cache blocking in GraphIt makes trade-offs between memory reads and writes: the range of random reads are controlled to fit in cache, but the random write to a vertex's application data is performed once per subgraph in which this vertex is present. The total number of random writes increases as the duplication factor α increases. Since these random writes are bounded by $O(\alpha|V|)$ instead of $O(|E|)$, the performance is tolerant of these duplications to a certain degree. The α value of a graph also positively correlates to its density: a dense graph is more likely to duplicates its vertices when being partitioned. As a result, partitioning a denser graph causes more random memory writes, but a denser graph incurs more random reads, and benefits more from cache blocking. Figure 5.2 shows the duplication factor from partitioning Twitter, WebGraph, and Friendster. Friendster has a higher duplication factor and a higher density than Twitter and WebGraph, as shown in Table 5.2 and Table 5.1. Figure 5-2 shows that applying cache blocking on Friendster has a bigger improvement in the overall cache hits compared to Twitter and WebGraph, and leads to a bigger performance gain.

Breadth-first search is the only algorithm that we evaluate for which cache blocking causes a slowdown due to the increased random memory writes. Breadth-first search incurs fewer random memory reads during edge traversal for two reasons. First, in the pull direction, breadth-first search only reads the runtime state (i.e., the `frontier` array), and does not read any application data from a vertex's neighbors (e.g., ranks for PageRank). The `frontier` array can fit in cache when it is represented using a bit-vector. Second, unlike other algorithms, breadth-first search does not traverse all of the edges in the pull direction: the `frontier` array is read less than $\Theta(|E|)$ times.

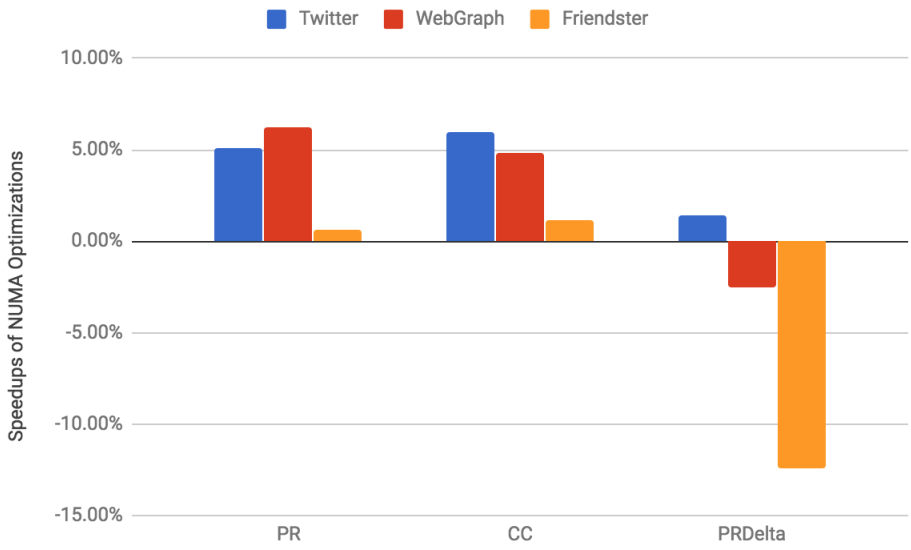

Figure 5-3: The speedups from applying NUMA optimizations compared to only using cache blocking. Twitter and WebGraph are partitioned into 16 subgraphs. Friendster is partitioned into 30 subgraphs. Negative speedups are equivalent to slowdowns.

As a result, cache blocking has little effect on random reads. On the other hand, graph partitioning increases the total number of random writes; hence cache blocking hurts the overall performance of breadth-first search.

5.3 NUMA Optimizations Performance

5.3.1 Overall Speedups

Figure 5-3 shows the speedups from NUMA optimizations over the existing best schedules with cache blocking for PageRank, connected components, and PageRank-Delta. Even though remote memory access is reduced from NUMA optimizations, the overall speedups are always less than 10% (and are sometimes within the measurement noise) because of work-efficiency and parallelism trade-offs. We have not evaluated the performance impact of NUMA optimizations on push-based traversals, which we consider as future work.

5.3.2 Locality, Work-Efficiency, and Parallelism Trade-offs

NUMA-Node Locality. NUMA optimizations improve NUMA-node locality. When vertex data fits in cache, remote memory access is mainly caused by reading the neighbors array (e.g., $N^-(v)$ on Line 2 of Figure 2-1). These memory reads are sequential and do not dominate the running time, so the effect of NUMA-node locality is minor. When we only apply NUMA optimizations but not cache blocking (i.e., partitioning the graph into two subgraphs and placing one subgraph on each socket), NUMA optimizations result in a larger speedup (e.g., 60% speedup on WebGraph running PageRank) since now the random reads of the vertex data are local to the socket.

Work-Efficiency. NUMA optimizations can negatively affect work-efficiency in two ways. First, an additional merge phase might be required to avoid atomic global operations in certain algorithms such as PageRank. Second, if updates are stored into socket-local buffers and are not immediately visible from threads running on a different socket, certain algorithms such as connected components might converge more slowly. Hence for connected components, instead of writing the new ID values to socket-local buffers, we directly update them in the global IDs array, sacrificing some locality to avoid redundant work. Breadth-first search is another example where NUMA optimizations can result in more work if updates are stored locally: a vertex present in both sockets needs to find an active parent on both sockets. Similarly, we can also choose to directly write to the global parent array. The running time of breadth-first search with NUMA optimizations is still worse than the original version due to reasons mentioned in Section 5.2.2.

Parallelism. NUMA optimizations hurt parallelism because when different sockets process different subgraphs simultaneously, work load imbalance can occur among the sockets. While dynamic cross-socket work-stealing is a first step towards better load balance, the granularity at which our work-stealing takes place (at the subgraph level) is too coarse-grained.

We do not apply NUMA optimizations for collaborative filtering because the Netflix graph is highly skewed. NUMA optimizations, as a result, severely reduce par-

allelism. The highly skewed distribution of the Netflix graph intuitively comes from the fact that while hundreds of thousands of users could have watched one popular movie, few, if any, users are likely to have watched even thousands of movies.

5.4 Comparisons with Other Frameworks

As a part of the larger effort of comparing GraphIt against other state-of-the-art frameworks including the ones that do not focus on cache or NUMA optimizations, we evaluate the performance of seven other in-memory graph processing systems: Ligra, GraphMat, Green-Marl, Galois, Gemini, Grazelle, and Polymer. Ligra has fast implementations of breadth-first search and single-source shortest paths [34]. Among prior work, GraphMat has the fastest shared-memory implementation of collaborative filtering [39]. Green-Marl is one of the fastest DSLs for the algorithms we evaluate [19]. Galois (v2.2.1) has an efficient asynchronous engine that works particularly well on road graphs [28]. Gemini and Grazelle are the focus of our comparisons as they both optimize for NUMA locality [18, 46]. Polymer is the first NUMA-aware graph

Application	PR					BFS				
Graph	LJ	TW	WB	RD	FT	LJ	TW	WB	RD	FT
GraphIt	**0.342**	**8.707**	**16.393**	0.909	**32.571**	0.028	**0.298**	**0.645**	**0.216**	**0.490**
Ligra	1.190	49.000	68.100	1.990	201.000	**0.027**	0.336	0.915	1.041	0.677
GraphMat	0.560	20.400	35.000	1.190		0.100	2.800	4.800	1.960	
GreenMarl	0.516	21.039	42.482	0.931		0.049	1.798	1.830	0.529	
Galois	2.788	30.751	46.270	9.607	117.468	0.038	1.339	1.183	0.220	3.440
Gemini	0.430	10.980	16.440	1.100	44.600	0.060	0.490	0.980	10.550	0.730
Grazelle	0.368	15.700	20.650	**0.740**	54.360	0.052	0.348	0.828	1.788	0.512
Polymer	1.420	47.500	64.800	1.400		0.654	6.080	13.200	115.000	
Application	CC					SSSP				
Graph	LJ	TW	WB	RD	FT	LJ	TW	WB	RD	FT
GraphIt	**0.055**	**0.890**	**1.960**	17.100	**2.630**	**0.044**	**1.349**	**1.680**	**0.285**	**4.302**
Ligra	0.061	2.780	5.810	25.900	13.000	0.051	1.554	1.895	1.301	11.933
GraphMat	0.365	9.8	17.9	84.5		0.095	2.200	5.000	43.000	
GreenMarl	0.187	5.142	11.676	107.933		0.093	1.922	4.265	93.495	
Galois	0.125	5.055	15.823	12.658	18.541	0.091	1.941	2.290	0.926	4.643
Gemini	0.150	3.850	9.660	85.000	13.772	0.080	1.360	2.800	7.420	6.147
Grazelle	0.084	1.730	3.208	**12.2**	5.880					
Polymer	0.158	3.710	5.280	38.400		0.161	1.680		0.989	
Application	PRDelta					CF				
Graph	LJ	TW	WB	RD	FT	NX	NX2			
GraphIt	**0.183**	**4.720**	**7.143**	**0.494**	**12.576**	**1.286**	**4.588**			
Ligra	0.239	9.190	19.300	0.691	40.800	5.350	25.500			

Table 5.3: Parallel running time (seconds) of GraphIt, Ligra, GraphMat, Green-Marl, Galois, Gemini, Grazelle, and Polymer. The fastest results are bolded. The missing numbers correspond to a framework not supporting an algorithm and/or not successfully running on an input graph.

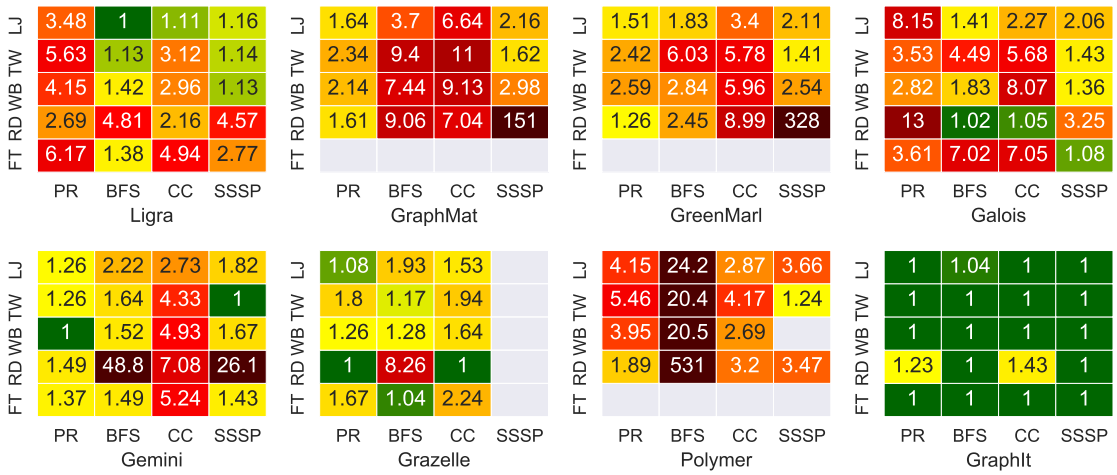

Figure 5-4: A heat map of slow downs of various frameworks compare to the fastest of all frameworks for PageRank (PR), breadth-first search (BFS), connected components (CC) using label propagation, and single-source shortest paths (SSSP) using Bellman-Ford, on five graphs with varying sizes and structures (LiveJournal (LJ), Twitter (TW), WebGraph (WB), USAroad (RD) and Friendster (FT). A lower number (green) is better with a value of 1 being the fastest for the specific algorithm running on the specific graph. Gray means either an algorithm or a graph is not supported by the framework. We use the same algorithms across different frameworks.

processing framework [43].

5.4.1 Overall Speedups

Table 5.3 shows the execution time of GraphIt and other systems. Figure 5-4 shows the slowdown of each of the algorithm-input pairs compared to the fastest framework. Table 5.4 shows the line counts of four graph algorithms for each framework. GraphIt often uses significantly fewer lines of code compared to the other frameworks, and outperforms the next fastest framework on 24 out of 27 experiments by up to 4.8 times. Yet GraphIt is never more than 43% slower than the fastest framework. On the graphs and algorithms where cache and NUMA optimizations are applied, GraphIt is strictly faster than existing frameworks.

5.4.2 Comparisons with Gemini

Table 5.5 shows that on the Twitter graph, GraphIt has the lowest LLC misses and cycles stalled compared to Gemini and Grazelle. GraphIt also has the lowest QPI

	GraphIt	Ligra	GraphMat	Green-Marl	Galois	Gemini	Grazelle	Polymer
PR	34	74	140	20	114	127	388	522
BFS	22	30	137	1	58	110	471	448
CC	22	44	90	25	94	109	659	376
SSSP	25	60	124	30	88	104		373

Table 5.4: Line counts of PR, BFS, CC, and SSSP for GraphIt, Ligra, GraphMat, Green-Marl, Galois, Gemini, Grazelle, and Polymer. Only Green-Marl has fewer lines of code than GraphIt. GraphIt has an order of magnitude fewer lines of code than Grazelle (the second fastest framework on the majority of the algorithms we measured). For Galois, we only included the code for the specific algorithm that we used. Green-Marl has a built in BFS.

Algorithm	PR				CC				PRDelta		CF	
Metrics	GraphIt	Ligra	Gemini	Grazelle	GraphIt	Ligra	Gemini	Grazelle	GraphIt	Ligra	GraphIt	Ligra
LLC miss rate (%)	**24.59**	60.97	45.09	56.68	**10.27**	48.92	43.46	56.24	**32.96**	71.16	**2.82**	37.86
QPI traffic (GB/s)	**7.26**	34.83	8.00	20.50	19.81	27.63	**6.20**	18.96	**8.50**	33.46	**5.68**	19.64
Cycle stalls (10^{12})	**2.40**	17.00	3.50	4.70	**0.20**	0.96	1.20	0.30	**1.25**	5.00	**0.09**	0.22
Running time (s)	**8.71**	49.00	10.98	15.70	**0.89**	2.78	3.85	1.73	**4.72**	9.19	**1.29**	5.35

Table 5.5: LLC miss rate, QPI traffic, cycles with pending memory loads and cache misses, and parallel running time (seconds) of PR, CC, and PRDelta running on Twitter, and CF running on NetFlix.

traffic on all algorithms except for connected components where NUMA optimizations affect convergence. GraphIt consistently outperforms Gemini for a few reasons. First, GraphIt has integrated more optimizations such as different representations of the frontier and cache blocking, whereas Gemini always uses a bit-vector for the frontiers and does not optimize for cache locality. Second, Section 5.3.2 pointed out that NUMA optimizations can have negative impacts on performance. Gemini always enables NUMA optimizations, which can hurt the performance of small graphs and applications such as breadth-first search and single-source shortest paths. The original Gemini paper also acknowledges these slowdowns [46]. Gemini's connected components performs poorly because NUMA optimizations negatively affect communications between threads on different sockets, as pointed out in Section 5.3.2.

5.4.3 Comparisons with Grazelle

GraphIt performs better than Grazelle on large graphs because GraphIt uses cache blocking but Grazelle does not. Grazelle uses the Vector-Sparse edge list to improve vectorization, which works well on graphs with low-degree vertices [18], outperforming GraphIt by 23% on PageRank and 43% on connected components running USAroad. GraphIt does not yet have this optimization, but we plan to integrate it in the future.

Chapter 6

Conclusion

6.1 Summary

In this thesis, we have integrated well-known techniques for improving memory locality—cache blocking and NUMA optimizations—into the GraphIt DSL. These optimizations improve the performance of GraphIt by up to a factor of 3. Combined with other optimizations, GraphIt can be up to 4.8 times faster than the fastest existing graph processing system. The blocking technique and NUMA primitives used in this work can also be applied to distributed graph processing. Overall, we show that cache and NUMA optimizations have a significant impact on the performance of graph processing, and building these optimizations in a DSL brings significant performance benefits while preserving the simplicity of the algorithm.

6.2 Future Work

GraphIt currently supports cache and NUMA optimizations for dense traversals. We also hope to improve locality for sparse traversals. In addition, we want to explore fine-grained inter-socket work-stealing to achieve better load balance. We think it is useful to support hierarchical NUMA optimizations that minimize communications between groups of NUMA nodes, but still allow communications within a group for better parallelism.

Bibliography

[1] Manu Awasthi, David Nellans, Kshitij Sudan, Rajeev Balasubramonian, and Al Davis. Handling the problems and opportunities posed by multiple on-chip memory controllers. In *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 319–330, Sept 2010.

[2] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 12:1–12:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[3] Scott Beamer, Krste Asanović, and David Patterson. Locality exists in graph processing: Workload characterization on an ivy bridge server. In *Workload Characterization (IISWC), 2015 IEEE International Symposium on*, pages 56–65, Oct 2015.

[4] Scott Beamer, Krste Asanović, and David Patterson. Reducing pagerank communication via propagation blocking. In *2017 IEEE International Parallel and Distributed Processing Symposium*, pages 820–831, May 2017.

[5] Richard Bellman. On a routing problem. *Quart. Appl. Math.*, 16:87–90, 1958.

[6] James Bennett, Stan Lanning, and Netflix Netflix. The netflix prize. In *KDD Cup and Workshop in conjunction with KDD*, 2007.

[7] Jari Björne and Tapio Salakoski. Generalizing an approximate subgraph matching-based system to extract events in molecular biology and cancer genetics. In *Proceedings of BioNLP Shared Task 2011 Workshop*, pages 183–191, Portland, Oregon, USA, 2011. Association for Computational Linguistics.

[8] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, August 1995.

[9] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1):107 – 117, 1998. Proceedings of the Seventh International World Wide Web Conference.

[10] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proc. VLDB Endow.*, 8(12):1804–1815, August 2015.

[11] Leonardo Dagum and Ramesh Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, Jan 1998.

[12] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 33–48, New York, NY, USA, 2013. ACM.

[13] Timothy A. Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011.

[14] Camil Demetrescu, Andrew Goldberg, and David Johnson. 9th dimacs implementation challenge - shortest paths. http://www.dis.uniroma1.it/challenge9/.

[15] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Num. Math.*, 1:269–271, 1959.

[16] Alessandro Epasto, Silvio Lattanzi, Vahab Mirrokni, Ismail Oner Sebe, Ahmed Taei, and Sunita Verma. Ego-net community mining applied to friend suggestion. *Proc. VLDB Endow.*, 9(4):324–335, December 2015.

[17] Paul Erdős. On the evolution of random graphs. *Bulletin of the Institute of International Statistics*, 38:343–347, 1961.

[18] Samuel Grossman, Heiner Litz, and Christos Kozyrakis. Making pull-based graph processing performant. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, pages 246–260, New York, NY, USA, 2018. ACM.

[19] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-Marl: A DSL for Easy and Efficient Graph Analysis. *SIGARCH Comput. Archit. News*, 40(1):349–362, March 2012.

[20] Fredrik Kjolstad, Shoaib Kamil, Jonathan Ragan-Kelley, David I. W. Levin, Shinjiro Sueda, Desai Chen, Etienne Vouga, Danny M. Kaufman, Gurtej Kanwar, Wojciech Matusik, and Saman Amarasinghe. Simit: A language for physical simulation. *ACM Trans. Graph.*, 35(2):20:1–20:21, March 2016.

[21] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 591–600, New York, NY, USA, 2010. ACM.

[22] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, and Christos Faloutsos. Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication. In *Proceedings of the 9th European Conference on Principles and Practice of Knowledge Discovery in Databases*, PKDD'05, pages 133–145, Berlin, Heidelberg, 2005. Springer-Verlag.

[23] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[24] David Levinthal. *Performance analysis guide for Intel core i7 processor and Intel Xeon 5500 processors.* 2010.

[25] Boduo Li, Sandeep Tata, and Yannis Sismanis. Sparkler: Supporting large-scale matrix factorization. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, pages 625–636, New York, NY, USA, 2013. ACM.

[26] Zoltan Majo and Thomas R. Gross. Memory System Performance in a NUMA Multicore Multiprocessor. In *Proceedings of the 4th Annual International Conference on Systems and Storage*, SYSTOR '11, pages 12:1–12:10, New York, NY, USA, 2011. ACM.

[27] Robert Meusel, Oliver Lehmberg, Christian Bizer, and Sebastiano Vigna. Web data commons - hyperlink graphs. http://webdatacommons.org/hyperlinkgraph, 2012.

[28] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471, New York, NY, USA, 2013. ACM.

[29] Rajesh Nishtala, Richard W. Vuduc, James W. Demmel, and Katherine A. Yelick. When cache blocking of sparse matrix vector multiply works and why. *Applicable Algebra in Engineering, Communication and Computing*, 18(3):297–311, May 2007.

[30] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[31] Francesco Ricci, Lior Rokach, and Bracha Shapira. *Introduction to Recommender Systems Handbook*, pages 1–35. Springer US, Boston, MA, 2011.

[32] Peter Sanders and Dominik Schultes. Highway hierarchies hasten exact shortest path queries. In *Algorithms – ESA 2005*, pages 568–579, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[33] Julian Shun. *Shared-Memory Parallelism Can Be Simple, Fast, and Scalable*. Association for Computing Machinery and Morgan & Claypool, New York, NY, USA, 2017.

[34] Julian Shun and Guy 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, pages 135–146, New York, NY, USA, 2013. ACM.

[35] Natcha Simsiri, Kanat Tangwongsan, Srikanta Tirthapura, and Kun-Lung Wu. Work-efficient parallel union-find. *Concurrency and Computation: Practice and Experience*, 30(4):e4333, 2017. e4333 cpe.4333.

[36] Stergios Stergiou, Dipen Rughwani, and Kostas Tsioutsiouliklis. Shortcutting label propagation for distributed connected components. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, WSDM '18, pages 540–546, New York, NY, USA, 2018. ACM.

[37] Josef Stoer, Roland Bulirsch, Richard H. Bartels, Walter Gautschi, and Christoph Witzgall. *Introduction to Numerical Analysis*. Springer, New York, NY, USA, 2002.

[38] Jiawen Sun, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. GraphGrind: Addressing Load Imbalance of Graph Partitioning. In *Proceedings of the International Conference on Supercomputing*, ICS '17, pages 16:1–16:10, 2017.

[39] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. GraphMat: High performance graph analytics made productive. *Proc. VLDB Endow.*, 8(11):1214–1225, July 2015.

[40] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. Speedup graph processing by graph ordering. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1813–1828, New York, NY, USA, 2016. ACM.

[41] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Computing*, 35(3):178 – 194, 2009. Revolutionary Technologies for Acceleration of Emerging Petascale Applications.

[42] Albert-Jan Nicholas Yzelman and Dirk Roose. High-level strategies for parallel shared-memory sparse matrix-vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 25(1):116–125, Jan 2014.

[43] Kaiyuan Zhang, Rong Chen, and Haibo Chen. NUMA-aware graph-structured analytics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 183–193, New York, NY, USA, 2015. ACM.

[44] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Saman Amarasinghe, and Matei Zaharia. Making caches work for graph analytics. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 293–302, Dec 2017.

[45] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. GraphIt - A High-Performance DSL for Graph Analytics. *ArXiv e-prints*, May 2018.

[46] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 301–316, GA, 2016. USENIX Association.