Scale-Free Graph Processing on a NUMA Machine

by

Tanuj Kr Aasawat

B. Engineering, Jadavpur University, India, 2015

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Applied Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Electrical and Computer Engineering)

The University of British Columbia
(Vancouver)

October 2018

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

Scale-Free Graph Processing on a NUMA Machine

submitted by **Tanuj Kr Aasawat** in partial fulfillment of the requirements for the degree of **Master of Applied Science** in **Electrical and Computer Engineering**.

Examining Committee:

Matei Ripeanu, Electrical and Computer Engineering
Supervisor

Sathish Gopalakrishnan, Electrical and Computer Engineering
Examining Committee Member

Karthik Pattabiraman, Electrical and Computer Engineering
Examining Committee Member

Abstract

The importance of high-performance graph processing to solve big data problems targeting high-impact applications is greater than ever before. Graphs incur highly irregular memory accesses which leads to poor data locality, load imbalance, and data-dependent parallelism. Distributed graph processing frameworks, such as Google's Pregel, that employs memory-parallel, shared-nothing systems have experienced tremendous success in terms of scale and performance. Modern shared-memory systems embrace the so called Non-Uniform Memory Access (NUMA) architecture which has proven to be more scalable (in terms of numbers of cores and memory modules) than the Symmetric Multiprocessing (SMP) architecture. In many ways, a NUMA system resembles a shared-nothing distributed system: physically distinct processing cores and memory regions (although, cache-coherent in NUMA). Memory accesses to remote NUMA domains are more expensive than local accesses. This poses the opportunity to transfer the know-how and design of distributed graph processing to develop shared-memory graph processing solutions optimized for NUMA systems (which is surprisingly little-explored).

In this dissertation, we explore if a distributed-memory like middleware that makes graph partitioning and communication between partitions explicit, can improve the performance on a NUMA system. We design and implement a NUMA aware graph processing framework that treats the NUMA platform as a distributed system, and embraces its design principles; in particular explicit partitioning and inter-partition communication. We further explore design trade-offs to reduce communication overhead and propose a solution that embraces design philosophies of distributed graph processing system and at the same time exploits optimization opportunities specific to single-node systems. We demonstrate up to 13.9$\times$ speedup

over a state-of-the-art NUMA-aware framework, Polymer and up to 3.7× scalability on a four-socket machine using graphs with tens of billions of edges.

Lay Summary

Large-scale graphs processing introduces various performance and efficiency challenges due to the scale and inherent irregular topology of graphs. Distributed graph processing frameworks, like Google's Pregel, that employs multi-node platforms, have experienced tremendous success in terms of scale and performance. Modern single-node systems embrace Non-Uniform Memory Access (NUMA) architecture which is more scalable than other architectures. In many ways, a NUMA system resembles a distributed system: physically distinct CPUs and memory. This poses the opportunity to transfer the wisdom of distributed graph processing to NUMA systems.

In this dissertation, we design and implement a NUMA-aware graph processing framework that explores if a distributed-memory like middleware that makes graph partitioning and inter-partition communication explicit, can improve the performance on a NUMA system. We demonstrate up to 13.9$\times$ speedup over a state-of-the-art NUMA-optimized framework and up to 3.7$\times$ scalability on a four-socket machine using graphs with tens of billions of edges.

Preface

This thesis is based on the research project done by me under the supervision and guidance of Professor Matei Ripeanu. I was responsible for the design, implementation, modeling, validation, evaluation and analysis of the results, along with taking the lead in publication writing effort. The research presented in this thesis have been either published or accepted for publication.

The work that this thesis extends and evaluates against, was selected based on the following preliminary study; Professor Ripeanu and Tahsin helped me in the analysis of the results and editing the publication.

Tanuj Kr Aasawat, Tahsin Reza, Matei Ripeanu, *How well do CPU, GPU and Hybrid Graph Processing Frameworks Perform?*, Pages 458-466, 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), May 2018.

The research presented herein has been accepted for publication. Professor Ripeanu and Tahsin helped me in the analysis of my design and the results, and editing the publication.

Tanuj Kr Aasawat, Tahsin Reza, Matei Ripeanu, *Scale-Free Graph Processing on a NUMA Machine*, IEEE Workshop on Irregular Applications: Architectures and Algorithms (IA3) in conjunction with SC18, The International Conference for High Performance Computing, Networking, Storage, and Analysis, Dallas, TX, USA, November 2018.

Table of Contents

List of Tables

List of Figures

Acknowledgments

I would like to sincerely thank my advisor, Professor **Matei Ripeanu** for giving me the opportunity to work on this high-quality research project and for his invaluable guidance, insightful feedback and support throughout this journey. I am grateful to him for encouraging and allowing me to do internships at IBM Almaden Research Center and Amazon Web Services.

I would like to thank my lab colleagues for providing their feedback during meetings and research presentations, and sharing their knowledge and experience with me.

Last but not the least, this journey would not have been possible without the consistent motivation from my brother **Manish**, and immense support and inspiration from my parents **Jawahar** and **Mohini**. My deepest gratitude to them.

To my parents and brother

Chapter 1

Introduction

Graph processing is at the core of a wide range of big data problems, such as online social networks analysis [11, 18], bioinformatics [19, 29], transport network analysis [37], financial and business analytics [20], to name a few. Additionally, graph processing has found new applications in machine learning and data mining.

Graph algorithms incur highly irregular data-dependent memory access patterns, which leads to poor data locality. Further, most of the graph algorithms have a low compute-to-memory access ratio, i.e., they are memory-bound. Many real-world graphs are massive: some have hundreds of billions of edges - hence have huge memory footprint. For example, the Facebook graph [8] and Web Data Commons, a hyperlink graph [4], have more than 100 billion edges, which requires over two terabytes of memory.

To process such huge graphs, traditionally frameworks like Google's Pregel [26] and GraphLab [17] running on large shared-nothing clusters have been used, as these platforms provide large aggregated memory. Most of these frameworks use the Bulk Synchronous Parallel (BSP) Processing Model [39]. Here, the graph is partitioned explicitly among the processing units and as these clusters are not cache-coherent, the communication between different processing units is explicit. This is in contrast with graph processing frameworks [30, 35] that target single-node shared-memory systems, and treat shared-memory system as if it is based on Symmetric Multi-Processor (SMP) architecture. In SMP architecture the access time to any location in memory is uniform, therefore, there is no need for data

partitioning.

Non-Uniform Memory Access (NUMA, a.k.a. distributed shared-memory) architecture machines introduce a dilemma: on the one side, they provide shared memory - thus graph processing frameworks that treat shared-memory system as SMP architecture, can be directly used. On the other side, the cost of memory accesses is non-uniform (i.e, a socket has faster access to the local memory associated with it, than to remote/non-local memory associated with other sockets), thus explicit data placement is needed to obtain maximum performance and a graph framework developed in the style of frameworks that target distributed systems may prove to offer advantages.

1.1 Hypothesis

Since NUMA architecture resembles distributed systems, our intuition is, a graph processing framework, targeting NUMA-architecture, developed in the style of frameworks that target distributed systems (explicit partitioning and communication), provides following three potential avenues for performance improvement: (i) control over data placement with explicit partitioning, which allows design and experimentation with different partitioning strategies to improve load balancing and overall performance, (ii) better locality, and (iii) explicit partitioning helps in exploring different communication trade-offs since NUMA is a shared-memory system. Based on these intuitions, we postulate the following hypothesis: *A distributed-memory like middleware that makes graph partitioning and communication between partitions explicit, can improve the performance on a NUMA system.*

To test this hypothesis, we design and implement a NUMA-aware graph processing framework that treats the NUMA platform as a distributed system, hence embraces its design principles; in particular explicit partitioning and communication, and evaluate it against the state-of-the-art NUMA-oblivious [15] and NUMA-aware [42] graph processing frameworks. We further describe optimization techniques to reduce communication overhead. And finally, provide a set of practical guidelines for choosing the appropriate partitioning and communication strategies.

1.2 Contributions

The contributions of this dissertation are:

1) Design Exploration: Given their resemblance, there exist opportunities to transfer the know-how and design philosophies of distributed graph processing to develop shared-memory graph processing solutions optimized for NUMA systems. To this end, we explore a reference distributed design (Section 3.3). In particular, we evaluate the performance of a fully distributed (referred to as NUMA 2-Box design, §3.3.2) and one shared-memory (referred to as NUMA 1-Box design, §3.3.3) inter-partition communication strategies (where each partition belongs to a NUMA domain) and how they compare against a NUMA-oblivious implementation. We found that, on a NUMA platform, a graph processing solution based on the design philosophies that targets shared-nothing distributed system, consistently outperforms the state-of-the-art NUMA-oblivious shared-memory solution (Section 5.2). Additionally, we explore two distributed graph partitioning techniques for NUMA, and introduce a new partitioning technique (Section 3.2) that leads to load balance of up to 95% and overall performance improvement of up to 5.3×.

2) A New NUMA-aware Design: Based on our design explorations, we propose a design (referred to as NUMA 0-Box design, §3.3.4), that takes into account *distributed shared-memory* nature of NUMA, and consists of explicit graph partitioning and implicit communication. It improves data locality through NUMA-aware partitioning and at the same time minimizes the overhead of remote accesses by overlapping remote memory operations with computation. (Section 3.3)

Evaluation shows, this new design offers, for BFS up to 2.37×, SSSP up to 2.27× and PageRank up to 1.89× improvement in time-to-solution over the respective NUMA-oblivious implementations. This design, however, did not improve performance of PageRank over the NUMA 2-Box design (explained in Section 5.2).

3) Analytical Model for Performance Prediction: We present an analytical model for predicting algorithm performance for the three aforementioned NUMA designs (Section 3.4). We demonstrate the effectiveness of our prediction model for PageRank by comparing with empirical results. (Section 5.3)

4) Evaluation: We evaluate the aforementioned three NUMA-aware designs

for the following applications: PageRank, BFS and SSSP, using both real-world and synthetic graphs (with up to 128 billion undirected edges), on a Intel NUMA platform with four sockets and 1.5TB memory. Summary of our findings are the following:

(i) We compare the three graph partitioning strategies and find that our proposed approach offers up to 5.3× speedup and 95% load balanced partitions. (Section 5.1)

(ii) We demonstrate scalability on up to four sockets on a NUMA platform: maximum speedup (over one socket) achieved by PageRank is 3.7×, BFS is 2.9× and SSSP is 2.8×. (Section 5.2)

(iii) We show RMAT scaling using up to Scale 32 graph. Our BFS implementation achieves a maximum of 39 giga traversed edges per second (GTEPS). (Section 5.2)

(iv) We compare our work with a recent NUMA-aware graph processing framework, Polymer and demonstrate that our solution consistently outperforms Polymer, e.g. up to 13.9× faster for BFS. Additionally, our solution is ∼4.4× more memory efficient. (Section 5.4)

(v) Finally, we present the performance numbers we achieved in Graph500 competition, where we secured World Rank 2 (June, 2018 list) for SSSP kernel, and among top 3 single-node submissions for BFS kernel. (Section 5.2)m

1.3 Dissertation Structure

The rest of this dissertation is organized as follows. Chapter 2 presents background and related work. Chapter 3 describes the design of our NUMA-aware graph processing framework. Chapter 4 presents the methodology used to implement and evaluate the designs introduced in Chapter 3. Chapter 5 evaluates the performance of our designs. And, Chapter 6 concludes the dissertation.

Chapter 2

Background

This section provides the necessary background information required to understand the contributions of this dissertation. First, this chapter presents a brief overview of graph (§2.1) and graph processing (§2.2). Then it describes three common CPU based hardware platforms (§2.3) used for graph processing. Next, this chapter describes Bulk-Synchronous Parallel (BSP) model (§2.4), a popular processing model among distributed systems, and explains thoroughly how it is leveraged in the context of graph processing (§2.5). Finally, it provides an overview of the graph algorithms that we have used (§2.6), followed by related work (§2.7).

2.1 Graph

A graph $G = (V, E)$, as shown in Figure 2.1, consists of a set of vertices V and a set of edges E. If the edges of a graph are unidirectional, the graph is called a directed graph. While, if all the edges of the graph are bidirectional, then it is called an undirected graph. Edges of a directed graph are represented by arrows (pointing towards the destination vertex), as shown in Figure 2.1, while the edges in an undirected graph are typically drawn as lines.

Graph Storage. Graphs are stored, usually, using either linked-lists or arrays. For high-performance and to efficiently store large graphs in memory, array based formats like Compressed Sparse Row (CSR), Coordinate (COO), Compressed Sparse Column (CSC), or Doubly Compressed Sparse Column (DCSC) formats are used.

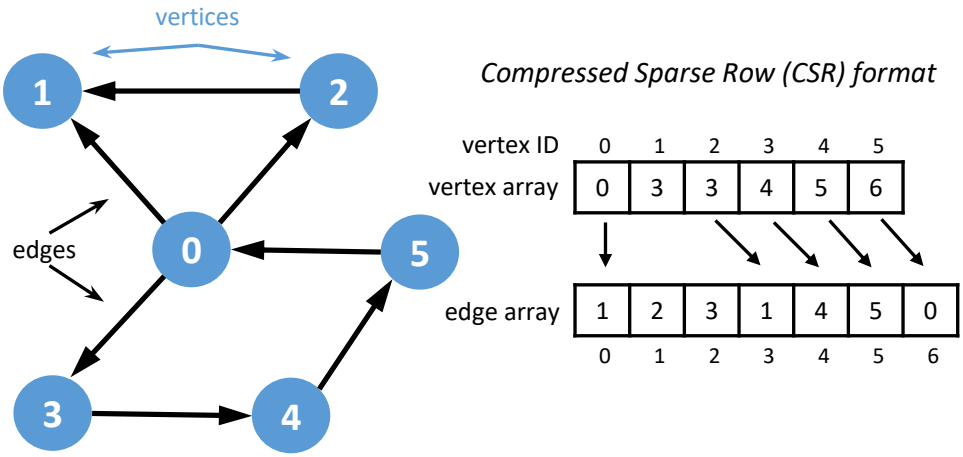


Figure 2.1: On left, a directed graph with 6 vertices and 7 edges. On right is the Compressed Sparse Row (CSR) representation of the graph. CSR format has two arrays, 'vertex array' (or offset array) and 'edge array'. 'vertex array' contains starting index of the outgoing edges originating from each vertex (represented by the indices of vertex array). 'edge array' contains only the destination vertices of the edges.

Figure 2.1, presents the CSR format, a popular format (that we have also used) used to achieve better performance and memory efficiency. CSR format targets directed graphs. To store undirected graphs, each edge of the undirected graph is represented by two direct edges (one in each direction). As shown in Figure 2.1, 'vertex array' and 'edge array' are the two CSR data structures. Size of the 'vertex array' is same as the number of vertices in the graph, and it contains the starting index of the neighbors list of each vertex. Indices of the 'vertex array' represents the vertex ID. 'edge array' contains the destination vertices of the edges originating from source vertex in 'vertex array', and its size equals the number of edges in the graph. For example, in Figure 2.1, vertex 2 has one outgoing edge, to vertex 1. The value, 3, at index 2 in 'vertex array', is the index value of the start of the neighbors list of vertex 2 in edge array. Size of the neighbors list of a vertex is determined by subtracting the current value at the index location in vertex array from the next value or from the edge count for the last vertex. Therefore, neighbors list size of vertex 2 is (value at index location $2+1$) - (value at index location 2) i.e. $4-3=1$.

2.2 Graph Processing

A wide set of big data problems, like analyzing online social networks, bioinformatics, financial and business analytics, transport network analysis, to name a few, can be modeled as graphs. For example, in online social networks, people are represented by vertices and an edge between the two represents their friendship. Further, domains like machine learning and data mining are also exploring graph processing at their core. In all these high impact applications, in order to get meaningful insights from the huge data, these massively large graphs need to be processed fast yet efficiently (w.r.t cost).

Graph algorithms and workloads pose following key characteristics that make them challenging to process efficiently.

1. *Iterative.* A typical graph algorithm processes a graph in rounds, where in each round only a set of vertices is active and can be processed in parallel. For example, in BFS, processing starts from a source vertex and it activates its neighbor vertices only, which then iterate over their respective neighbor vertices in the next round, and so on.

2. *Highly irregular, data-dependent memory access patterns.* Graph processing suffers from highly irregular data-dependent memory access pattern as the neighbors are scattered in memory. It leads to poor data locality and high random memory accesses.

3. *Low compute-to-memory-access ratio.* Most of the graph algorithms, like BFS and SSSP, have low compute-to-memory-access ratio, i.e. they do very less computation per memory access, thereby being memory bound. For example, in BFS, very little processing is done on the data associated with a vertex, and most of the time is spent in accessing the neighbors.

4. *Hard to obtain balanced partitions.* Many real-world graphs have heavily skewed, 'power-law' [14] vertex degree distribution: most of the vertices have low edge degree, while a few vertices have high edge degree (e.g., celebrities in online social networks) - that connect to a large part of the graph. These type of graphs are also called as ***scale-free graphs***.

If we process these graphs in a distributed system, their heavily skewed degree distribution makes it hard to obtain balanced partitions to achieve better load balance and over all performance. Further the partitioning algorithms developed specifically to obtain good partitioning are computationally expensive.

5. *Large memory footprint.* Many real-world graphs are massive: some have hundreds of billions of edges - leading to a huge memory footprint. For example, current Facebook graph (a social network graph with $\sim$137 Billion edges) and Web Data Commons - Hyperlink Graph (a web graph with $\sim$128 Billion edges) [3, 4], require more than 2TB of memory. To process such large graphs efficiently, the whole graph needs to be in the memory.

2.3 Hardware Platforms

This section aims at describing the three popular CPU-based hardware platforms.

2.3.1 Shared-nothing cluster

A distributed system or a shared-nothing cluster consists of hundreds to thousands of processing units (also called as nodes), where each processing unit has access to only its own memory, that are connected with each other through fast interconnects, like OmniPath and InfiniBand.

Given the huge memory footprint of real-world graphs, traditionally these large shared-nothing clusters have been used, as they have large aggregated memory. These memory-parallel, shared-nothing clusters have experienced tremendous success in terms of scale and performance, as could be seen in Graph500 Competition [2] (which ranks supercomputers for data intensive applications), as well as been used by many distributed graph processing frameworks including Google's Pregel [26] and GraphLab [17].

These shared-nothing clusters are not cache-coherent. Here, the graph is partitioned explicitly (one partition on each of the nodes) and the communication between different nodes is explicit. Graph partitioning leads to having boundary edges that cross-over between nodes. The nodes run the graph algorithm kernel in-

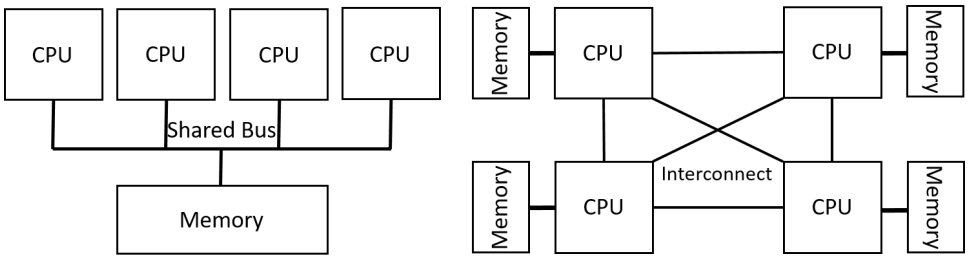


Figure 2.2: An illustration of SMP/UMA (left) and NUMA (right) architectures.

parallel on their respective partition, and communicate with other processing units to share the remotely updated vertex state.

2.3.2 Symmetric Multi-Processor (SMP) Architecture

In Symmetric Multi-Processor (SMP) architecture, memory is shared between processing units, and it provides uniform access time to any location in memory. It is, therefore, also termed as Uniform Memory Access (UMA) architecture. As shown in Figure 2.2 (left), the CPU cores share the same memory bus to access the memory. This leads to uniform access time from any core to any location in memory. In recent SMP architecture machines, the memory available could be up to few hundreds of gigabytes. This helps in processing mid-size graphs, that fit into memory, at a lower development cost compared to distributed system as there is no need of explicit partitioning and inter-partition communication.

The drawback of this architecture is, as the memory bus is shared among all the cores, this design leads to contention on the shared bus with increase in the number of CPU cores. Therefore, the design does not scales with number of CPU cores and memory.

2.3.3 Non-Uniform Memory Access (NUMA) Architecture

NUMA is a shared-memory architecture consisting of a set of processors (often called sockets), each with their own local memory. Each socket is connected with other sockets through an interconnect (Quick-Path Interconnect in Intel systems). Accessing socket-local memory takes distinctively less time than accessing remote

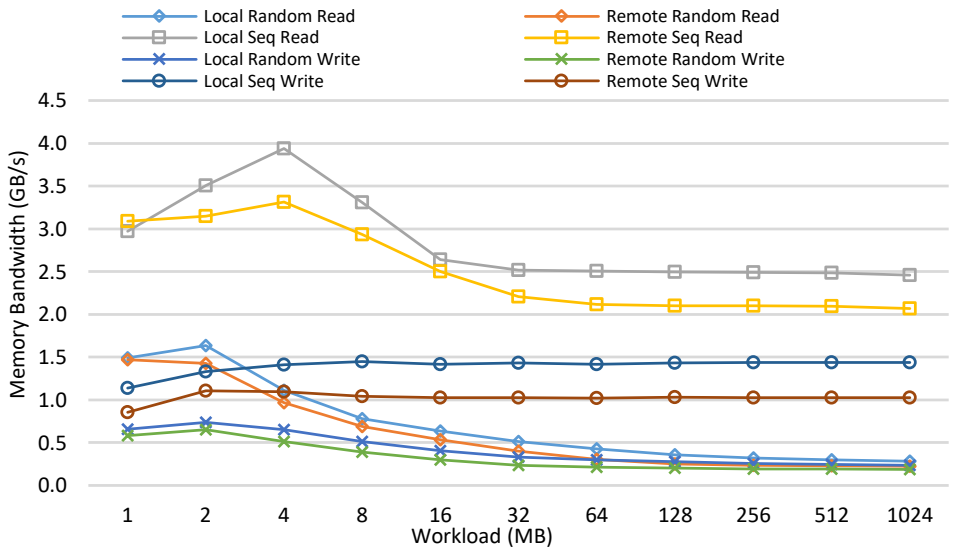


Figure 2.3: Local and remote memory bandwidth for read and write operations on 4-Socket Intel Xeon (E7-4870 v2, Ivy Bridge) machine, for different workloads (size in MB).

memory over the interconnect. NUMA addresses the scalability issues of SMP architecture, and provides higher overall memory bandwidth.

NUMA distributes memory to each processors: (Fig. 2.2 - right) each processor has fast access to its local memory, while to access local memory of another processor, it has to traverse over the slow interconnect. This reduces contention over local memory bus as well as provides the opportunity to scale the system. Note that scalability comes at the cost of remote memory access. Obtaining maximum performance requires careful placement of data to avoid/minimize remote memory accesses with lower latency and higher bandwidth.

We benchmark our testbed, a four socket Intel Xeon system (more description in Section 4.2), by extending Stream [27] benchmark to measure local and remote, read and write memory bandwidth for both sequential and random accesses. We measure these access patterns, because they frequently occur in graph processing. We run the experiments for array size from 1 MB and double the array size until 1 GB (at which point it saturates the memory bandwidth). Table 2.1 presents the memory bandwidth achieved in different access modes for arrays of size 1 GB. We

	Access	Local	Remote
Read Throughput (MB/s)	Sequential	2464	2069
	Random	286	226
Write Throughput (MB/s)	Sequential	1438	1024
	Random	238	188
Latency (ns)		119	178

Table 2.1: Memory bandwidth characteristics of our Testbed (Four socket Intel Xeon, E7-4870 v2; more description in Section 4.2). Memory bandwidth is measured with custom benchmark with arrays of size 1 GB. Memory latency is measured using Intel Memory Latency Checker.

observe that non-local access throughput is up to 26% and 40% slower than local access throughput for read and write operations, respectively. Interestingly, remote sequential access throughput is as much as 9× and 6× more than local random access throughput for read and write operations, respectively. This observation is important for graph processing, which incurs highly irregular memory access patterns. On the other side, remote memory latency, as measured using Intel Memory Latency Checker, is 49% more than local memory latency.

2.4 Bulk-Synchronous Parallel (BSP) Processing Model

Bulk-Synchronous Parallel (BSP) model is a popular processing model targeting distributed systems. Therefore, BSP model implies that the data is partitioned and partitions are allocated on the processing elements. In the BSP model (Fig. 2.4), the processing consists of a sequence of rounds or *supersteps* (in BSP terminology). Each superstep consists of three phases (executed in order): computation, communication, and synchronization.

- In the *computation phase*, each processing unit processes their respective partition independently.
- In the *communication phase*, processing units exchange messages with respective remote partitions, as well as apply the remote updates to their local buffers.

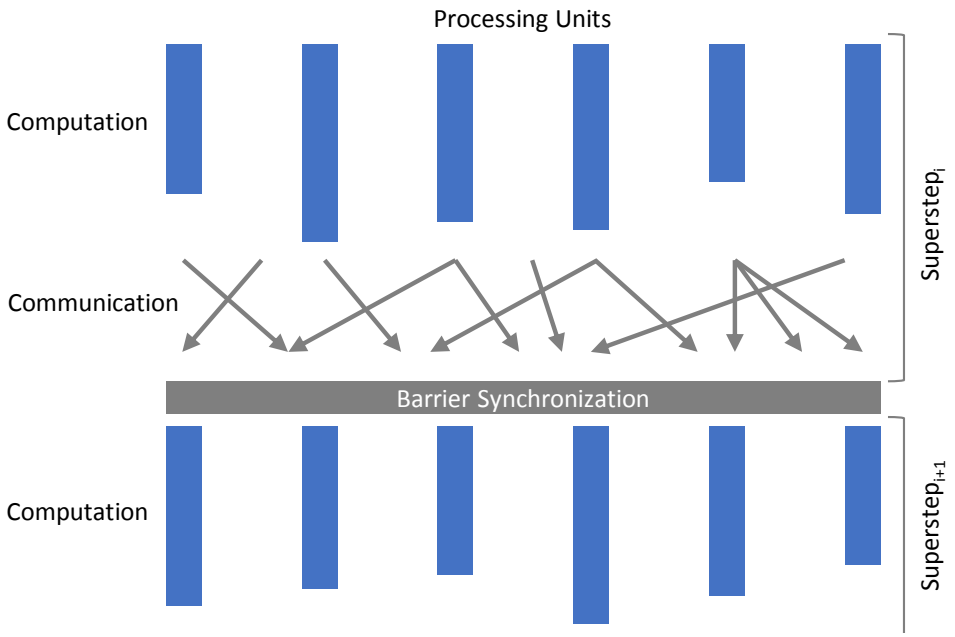


Figure 2.4: A high level illustration of the Bulk Synchronous Parallel Model.

- The *synchronization phase* guarantees that the next cycle restarts only after all messages have been delivered.

This sequence of supersteps continues until convergence or termination conditions has been satisfied. Finally, if required, the results are aggregated from all the partitions.

2.5 BSP-Style Graph Processing

Graph computations can be modeled as Gather-Apply-Scatter (GAS) [17], where the graph processing follows sequences of **g**ather, **a**pply and **s**catter operations. In gather phase, vertices *gather* information from their neighbors to update their local state in *apply* phase, and then in *scatter* phase, they communicate their updated value to their neighbors. For example, in PageRank, a vertex computes its rank by gathering rank of its in-degree neighbors, and scatters its new rank to its out-degree neighbors. The BSP processing model inherently matches with this iterative nature of graph algorithms, where sequence of gather, apply and scatter operations

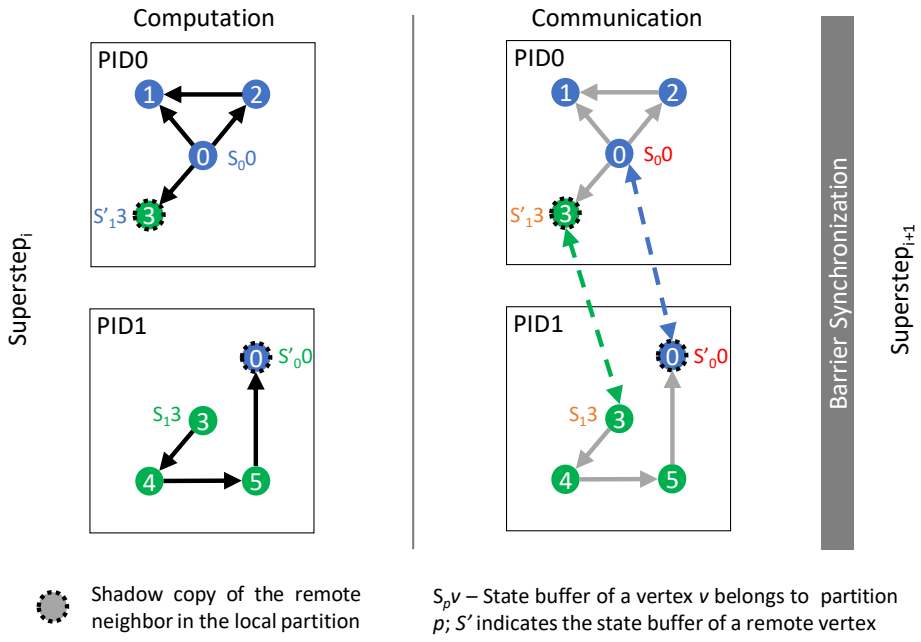


Figure 2.5: BSP graph processing depicting computation and communication phases in a superstep. In the computation phase, the state of a vertex and its shadow copy (on a remote partition) are updated independently. In the communication phase, the vertex and its shadow copy communicate to determine the correct state. For example, in SSSP, after communication, both the vertex and its shadow copy commit to the minimum distance at that point in traversal.

resembles the three phases of a superstep in BSP model.

Since BSP model implies that the data is partitioned and allocated on the processing elements, initial step is to partition the graph (partitions in Figure 2.5 are of the graph shown in Figure 2.1). Each partition has a set of local vertices and edges. Since an edge is associated with two vertices, which could be on different partitions, a map is maintained for the remote vertices (vertex 3 in PID0 and vertex 0 in PID1, in Fig. 2.5) in each partition. Further each partition maintains algorithm specific state buffer(s) (such as *rank* array in PageRank) for local vertices (buffer S_0 in PID0 and S_1 in PID1, in Fig. 2.5) as well as remote vertices (buffer $S_1^{'}$ for remote vertices in PID0 and $S_0^{'}$ for remote vertices in PID1, in Fig. 2.5).

The three phases of a superstep of BSP model are performed as follows in the context of graph processing:

In *computation phase*, processing units work in parallel, and execute the graph algorithm specific kernel on the set of vertices belonging to their partition, and update their local state buffer (buffer S_0 and S_1 in Fig. 2.5). The local state of active remote vertices is also updated and aggregated locally in the respective buffer (buffer $S_1^{'}$ and $S_0^{'}$ in Fig. 2.5).

In *communication phase*, each partition exchange the messages for the boundary edges, and applies the remote updates received to their local state buffers. In Fig. 2.4, both the partitions transfer the state of remote vertices to make sure local and remote states of the vertices are same, i.e. S_0 and $S_0^{'}$ are the same, and $S_1$and $S_1^{'}$ are the same.

Finally, *synchronization phase* ensures that all the partitions are updated with the latest state of the remote vertices, before the superstep cycle restarts.

Similar to the generic BSP model, the sequence terminates once every processing unit has finished processing their respective partitions. After termination, final result is aggregated from all the processing units through a global reduction.

2.6 Graph Algorithms

We consider PageRank, Breadth-First Search - Top Down (BFS-TD), Breadth-First Search - Direction Optimized (BFS-DO), and Single-Source Shortest Path (SSSP) algorithms. ***We use these algorithms to evaluate our work because*** (i) these algorithms have been widely studied in the context of high-performance graph processing systems and have been used in the past studies [5, 15, 17, 30, 35, 36, 42], (ii) BFS and SSSP are also used as benchmarks for the Graph500 competition [2], to rank supercomputers for data intensive applications, (iii) they are the building blocks of more complex graph algorithms (for example, BFS is used as a subroutine in complex graph algorithms like connected components, max flow, betweeness centrality and clustering), and (iv) these algorithms are good representation for studying the performance of any hardware platform for irregular memory access pattern.

A short description of the algorithms is described below.

2.6.1 PageRank

PageRank [31] is a well-known algorithm used by search engines for ranking web pages. In PageRank, a vertex computes its rank based on the rank of its neighbors. The algorithm continues until the convergence of the rank of all the vertices, or a predefined number of iterations have been completed. *PageRank has a high compute-to-memory-access ratio, and the workload is stable in every iteration, since in each iteration it computes the rank of all its vertices*. It could be implemented as a pull-based or push-based algorithm [35]. In the pull-based approach, each vertex 'pulls' the rank of its neighbors, over the incoming edges, to compute its new rank. In the push-based approach, each vertex 'pushes' its rank to its neighbors, over the outgoing edges. Note that the push-based approach is less efficient, since, its parallel implementation requires atomic operations [30]. We implement pull-based approach and the algorithm kernel executes for predefined number of iterations [16].

2.6.2 Breadth-First Search

BFS is a graph traversal algorithm which determines the level of each vertex, starting from a source vertex. It is a fundamental graph algorithm which is also used as a sub-routine in complex graph algorithms, like connected components, betweeness centrality, max flow, and clustering. *BFS has a low compute-to-memory-access ratio, and since it is a traversal based algorithm, workload is not stable in every iteration (or superstep)*. Like other graph traversal algorithms, BFS presents the concept of a *frontier*, which consists of a set of active vertices that are processed in the current iteration, to build the *next frontier*. The *next frontier* can be manipulated in different ways. We explore three implementations of BFS algorithm.

BFS - Top-Down (BFS-TD)

It is the classic level-synchronous approach of doing BFS. In each iteration it processes all the edges of the vertices in the current frontier, to build the next frontier with the unvisited vertices that can be reached. For power-law graphs, it has been observed that this approach leads to *(1) drastic increase in the frontier in initial few supersteps followed by steep decrease in frontier size in tailing supersteps, and (2)*

high write traffic, in the initial supersteps, since many edges in the current frontier tries to add the same vertex in the next frontier [10].

BFS - Direction Optimized (BFS-DO)

Direction Optimized BFS [10] addresses the above mentioned drawbacks of Top-Down version of BFS, by manipulating the next frontier in bottom-up way when the current frontier is large. In Bottom-Up step, it iterates over unvisited vertices and selects those for the next frontier which have a neighbor in the current frontier. This helps in *drastically reducing the number of edges explored especially when the frontier is large*, since once an unvisited vertex, that has a neighbor in current frontier, is explored, there is no need to explore its other edges. This, especially, reduces work for high-degree vertices. Further, this approach *does not require any atomic operation* as the write operation is done only to update the state of the unvisited vertex, to include it in *next frontier*, while rest of the accesses are read - to check if any of its neighbors are in the *current frontier*, thereby reduces the contention [10, 34]. Direction-Optimized BFS kernel starts with Top-Down step, and once the frontier size is large enough, it switches to Bottom-Up step. For the final supersteps, when the frontier size is again small, it switches back to Top-Down step. ***Note that switching between the steps (from Top-Down to Bottom-Up and from Bottom-Up to Top-Down) is heuristics based, and one needs to hand-tune them to attain maximum performance on a particular graph.***

BFS-Graph500

Graph500 competition [2] has different requirements for measuring the performance. First, it counts an undirected edge as only one edge, while we represent an undirected edge as two directed edges, one in each direction. Therefore we have to half the number of edges traversed, while computing the performance. Second, it requires including algorithm initialization time as well in the algorithm execution time, not a standard practice in the literature. And finally, it requires the BFS tree as output rather than the level of each vertex in the BFS tree. We have implemented our BFS-DO as the kernel inside Graph500 skeleton, and modified its data structures with Graph500 requirements.

2.6.3 Single-Source Shortest Path

SSSP is a traversal based graph algorithm and finds the shortest path from a source vertex to every vertex in the connected component. It has wide applications including IP routing, transportation networks, and social network analysis. In SSSP, each edge is associated with a predefined weight which, typically, is a measure of 'cost' to make a transition from one vertex to one of its neighbors. Weights increase the memory footprint of the graph by almost $2\times$ (depending on the data type used). Following are the two implementations of SSSP algorithm that we use:

SSSP

We adapt Bellman-Ford algorithm [1] to implement SSSP, as it provides better scope of parallelism and the opportunity to allow the active vertices to perform relax operation on its edges within the same iteration [16]. *This reduces the number of supersteps, since more number of vertices become active per superstep.* The algorithm gives the distance of each vertex from the given source vertex.

Graph500-SSSP

Along with the distance buffer, containing distance of each vertex from the source vertex, for SSSP, Graph500 also requires sssp-tree containing parent of each vertex. This is expected to increase the communication overhead in every superstep by almost $2\times$ because of communicating the parents for boundary edges among the partitions, along with the respective distance value. ***We optimize this by not requiring to communicate the tree at all during supersteps, and aggregate the tree in only the aggregation phase.*** In the computation phase, if the edge is a boundary-edge, we store the partition ID of the remote vertex and the local ID of the parent vertex in the tree buffer. Distance buffer is updated same as the above implementation of SSSP. During communication, if for the remote vertex, the distance in the remote distance buffer is less, then it stores the remote partition ID for the respective vertex in the local tree buffer. This way it knows that parent is in the respective remote partition. In aggregation phase, it iterates over the local tree buffers of all the partitions to aggregate the results. If the value in the local tree buffer corresponds to a remote partition, it determines the parent by looking at the

respective tree buffer in that remote partition, and gets the global id of the parent from the global map.

2.7 Related Work

Since their inception, NUMA architecture has been the source of performance issues, because of their distributed shared-memory, in performance critical applications targeting shared-memory systems.

This section describes the work done on addressing the performance issues on NUMA architecture, in general. Then it discusses the graph processing frameworks which targets single-node shared-memory systems, and are NUMA-oblivious. And, finally it discusses the graph processing framework that targets shared-memory NUMA architecture, followed by NUMA-aware graph kernels.

NUMA-aware work. As shared-memory NUMA architectures are becoming ubiquitous in today's commodity servers, many work have shown the performance issues on running the applications that were implemented for shared-memory (assuming SMP architecture), and have presented optimizations which improved their performance on NUMA-architecture based shared-memory systems. For Databases, works like [22, 25] have shown maximum performance gain in the range of $3\times$ - $6\times$ on accelerating different data management primitives and in-memory storage operations, with NUMA optimizations. Further, NUMA effects are also severe in Machine Learning [28] and Deep Learning [33], where workload is regular. NUMA-Caffe [33] have shown that the convolution layer (the most significant and time consuming layer [21]) in Convolution Neural Network (a type of Deep Neural Network) leads to maximum remote memory accesses, and with NUMA optimizations they achieved performance gain of $2\times$ to $14\times$.

Shared-memory Graph Processing Frameworks. Since shared-memory systems have up to few hundreds of gigabytes of memory available, which is enough to process mid-size graphs, frameworks like Galois [30], Ligra [35] and Totem [15] have been developed. These frameworks treats shared-memory system as if it is based on SMP architecture. And since NUMA architecture is also shared-memory based, these frameworks run on NUMA architecture as well. But, they suffer from the distributed nature of the shared-memory in NUMA, and hence do not perform

and scale well [5].

NUMA-aware Graph Processing Framework. To the best of our knowledge, Polymer [42] is the only NUMA-aware graph processing framework. It embraces the design philosophy of distributed systems, and extends Ligra to improve performance on NUMA-architecture based shared-memory systems. (discussed in detail in Section 5.4) ***NUMA-aware graph kernels.*** There are work, like [40, 41], that have optimized specific graph kernels, primarily BFS for Graph500, for NUMA architecture. The NUMA-aware optimizations that they have done are specific to graph kernels. On the other hand, *all the NUMA optimizations in our work are graph algorithm agnostic.*

Chapter 3

A BSP-style NUMA-aware Graph Processing Framework

3.1 Intuition

The NUMA architecture resembles distributed shared-nothing platforms. As described in previous section, the BSP processing model naturally matches graph computation. Therefore, for distributed graph processing, BSP graph processing model is commonly used. On a NUMA machine, the expected benefits of using BSP model are: (i) *Explicit data placement,* which means data is processed at the node-local level - thus having the potential to reduce processing time through *better locality* as during processing no remote accesses are made, (ii) *Explicit partitioning* allows experimentation with different load balancing techniques, and (iii) as NUMA is a shared-memory system, we can explore different inter-partition communication trade-offs, to reduce communication overhead. The advantages obtained through explicit data placement and partitioning need to be greater than the overheads present in having BSP model on a shared-memory NUMA system. The expected overheads are: (i) inter-partition communication overhead, (ii) memory overhead - since we have to store the state of remote vertices on each partition, (iii) thread management, (iv) development overhead - designing and implementing partitioning, inter-partition communication and result aggregation, and (v) preprocessing overhead of partitioning.

The goal of this study is to evaluate if having a distributed-memory like middleware on a shared-memory NUMA machine provides performance advantages in spite of aforementioned overheads.

This chapter describes the design of our BSP-style, NUMA-aware graph processing framework, starting with graph partitioning strategies (§3.2), followed by describing the design opportunities for BSP-style graph processing on shared-memory NUMA system (§3.3). Then it provides an analytical model to predict their performance (§3.4). Finally, it describes how the GAS model maps to our design (§3.5).

3.2 Graph Partitioning

Distributed graph processing begins with partitioning the graph and allocates the partitions on the processing units. The goals of partitioning are: (i) to process large graphs - to leverage the large aggregated memory, (ii) to improve load balance, and (iii) to process the partitions in parallel.

Graph partitioning is an NP-complete problem [7], and having balanced partitioning on real-world power-law [14] graphs is challenging [6, 23, 24]. Popular distributed graph processing frameworks like Pregel [26] and GraphLab [17], do random partitioning, where vertices are distributed randomly among the processing units, as it leads to uniform vertex degree distribution. Heterogeneous distributed system like Totem [15] uses Sorted/Degree-aware partitioning and have shown that this strategy performs better than Random partitioning on a single-node hybrid system. *The **success criteria** for a good partitioning strategy are: (i) better load balance, and (ii) most importantly, better overall performance.*

In a NUMA system, explicitly partitioning the graph does not help in processing larger graphs, as the memory available is fixed, but it provides better locality (by serving all the accesses from local memory of the NUMA node where the partition is assigned to), and enables implementing and experimenting with different partitioning strategies, designed for distributed systems, to improve load balance and overall performance. We have implemented two graph partitioning strategies, random and Sorted/Degree-aware. We also introduce a new partitioning strategy that leads to better load balance and higher performance, than the above two parti-

tioning strategies.

Random Partitioning. In this partitioning strategy, vertices are assigned randomly to the processing units. Random partitioning is a popular strategy among the distributed graph processing systems, like Pregel and GraphLab, targeting graphs having power-law vertex degree distribution [14]. It increases the probability of each partition having equal variability in terms of vertex degree.

Sorted or Degree-aware Partitioning. In this approach the vertices are first sorted by degree, and then they are assigned to the processing units as a contiguous chunk of vertices with even share of edges. This strategy leads to better locality since the likelihood of having most of the neighbors in the same partition increases. It has been shown to perform better than random partitioning in Totem [15], a heterogeneous distributed system. But, load imbalance increases significantly, as shown in Fig. 5.1, since few partitions get dense subgraph (chunk with high-degree vertices will have few vertices) while others get sparse subgraph (tailing chunk consist of low-degree vertices, thereby the subgraph will have most of the vertices).

New Strategy - Hybrid Partitioning. We observed that Random partitioning leads to better load balance but suffers from poor data locality. Sorted or Degree-aware partitioning on the other hand achieves better data locality, but leads to severe load imbalance. With these observations, we designed and implemented a hybrid partitioning technique that alleviates this problem. In the first step, we randomly assign the vertices to the processing units, same as random partitioning. And then, we sort the vertex list of individual subgraphs by degree. Randomly assigning the vertices to the processing units increases the probability that each partition has equal variability in terms of vertex degree (thereby increasing the chance that the generated load is well balanced). Sorting individual vertex lists improves data locality [15]. Later we discuss its performance compared to other two strategies in Fig. 5.1 and Fig. 5.2.

3.3 Design Opportunities for NUMA-aware Graph Processing

Since we partition the graph and place one partition on each NUMA node, it allows us to do computation in parallel with all the accesses served from the local mem-

ory, during the computation phase of a superstep. Since NUMA system is a shared-memory system, we have the opportunity to explore shared-memory specific optimizations to reduce communication overhead. We explore three communication alternatives that address the motivation of this dissertation: *To what degree, designing for NUMA as for a distributed memory system can enable performance (by explicitly presenting locality), in spite of inherent overheads (message exchange), in an application agnostic way.*

In this section, we describe the data structures we have used in our framework, and the three design options to optimize communication overhead.

3.3.1 Data structures

To store the graph in-memory, we use Compressed Sparse Row (CSR) format, as described earlier in Section 2.1 as well as in Figure 3.1 (arrays V and E). As presented in [16], the arrays V and E represent the CSR data structure, where V_i contains the start index of the neighbors of the vertex i in the edge array E. In each partition p, the vertex IDs range from zero to ($|V_p|$ - 1), where V_p is the set of local vertices belonging to a partition. Edge array E stores the destination vertex of an edge, which has partition ID encoded in high-order bits (shown in Fig. 3.1 as subscripts). *For boundary (or remote) edges, value stored in E depends on the communication design we select*. For NUMA 2-Box (§3.3.2) and NUMA 1-Box (§3.3.3) designs, value stored is *the index to its entry in the outbox buffer* (discussed later), not the remote neighbor ID. But, for NUMA 0-Box design (§3.3.4), *value stored is the remote neighbor ID*.

The array S, of length $|V_p|$, represents the algorithm-specific local state of each local vertex in the partition. The message (outbox and inbox) buffers allocation varies by the design options (discussed in details in every design options). The outbox buffer is for the messages for the remote neighbors, and has an entry for each remote neighbor. The inbox buffer is for the messages for the local vertices which are remote to other partitions, therefore has an entry for each local vertex that is remote to another partition. Both the message buffers have two arrays: one to store the remote vertex ID, and the other stores the corresponding message. More

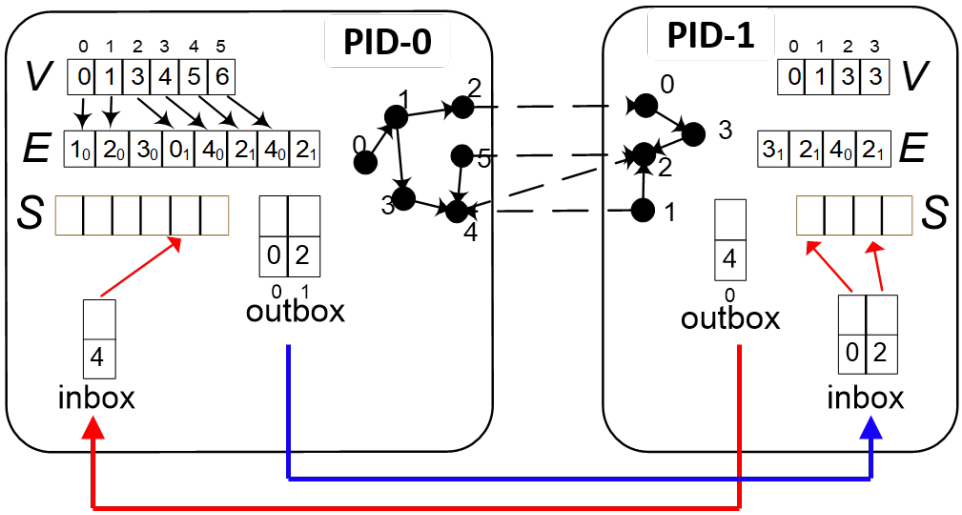


Figure 3.1: High-level illustration of inter-partition communication of NUMA 2-Box design. *V* and *E* are the buffers to represent the graph in CSR format (as mentioned in Section 2.1 and Figure 2.1). *S* is the state buffer for local vertices. Bottom blue and red solid lines depicts communication paths for NUMA 2-Box, where explicit memory copy through in - and out - boxes are required. For push-based algorithms, during computation phase, each partition manipulates its local state buffer *S* for local vertices and updates for remote vertices are aggregated locally in the outbox buffer. During communication phase outbox is copied into the inbox of the respective remote partition, which are then applied to the respective local state buffers.

details are provided according to the design options described below.

3.3.2 NUMA 2-Box Design

In this design, we fully embrace the design philosophy of a distributed system, thereby assuming NUMA as a shared-nothing distributed system - where nodes are independent and are connected through the interconnect. In this design, as shown in Fig. 3.1, for communication, it has two message buffers (outbox at source and inbox at destination partition). Further, as mentioned before, for remote edges, value stored in *E* is the index to its entry in the outbox buffer. So, the value in *E* for the entries 0_1 and 2_1 (in left partition - PID-0), and 4_0 (in right partition - PID-

1) is replaced by the index to its entry in respective outbox buffer, i.e. by 0_1 and 1_1 (where subscript 1 stands for the outbox for remote partition 1), and 0_0 (where subscript 0 stands for the outbox for remote partition 0), respectively.

Following is the BSP-style graph processing in this design for both push-based and pull-based algorithms (how we implement GAS model in our design is explained in §3.5).

For *push-based* algorithms like BFS and SSSP, in computation phase, each partition manipulates its local state buffer *S* for updates for its local vertices. *All the updates for remote vertices are aggregated and stored in respective outbox buffers.* In communication phase, as shown in Figure 3.1, the partitions transfer (blue and red arrows) the respective outbox buffer to the corresponding remote inbox buffer, and apply the remote updates, received from remote partitions in their corresponding inbox, to their local state buffers (red arrows from inbox to buffer *S*) if necessary conditions are met (for example, in SSSP, remote distance value is committed if it is lesser than the current value).

For *pull-based* algorithms, like PageRank, during compute phase, each local vertex updates its state by reading the state of its incoming-neighbors. The state of remote incoming-neighbors are accessed from respective outbox buffer. During the communication phase, the local vertices (which are remote in other partitions) update their new state in the respective inbox buffer, which is then copied to the outbox buffer of the remote partition. In the next superstep, this updated state is utilized to calculate the new state of the local vertices.

Advantages

(i) ***Zero remote memory accesses.*** This design leads to zero remote memory accesses, since all the accesses are local in both computation and communication phases, and the message buffer (out/in box) is explicitly copied to the remote partition's message buffer (in/out box).
(ii) ***Message aggregation.*** A vertex can be associated with many edges (as average degree of the vertices is 32 for synthetic workloads and $\sim$75 for real-world graphs 4.3). *Aggregating the remote updates for the remote vertices locally leads to sending only one message per remote vertex during the communication phase.*

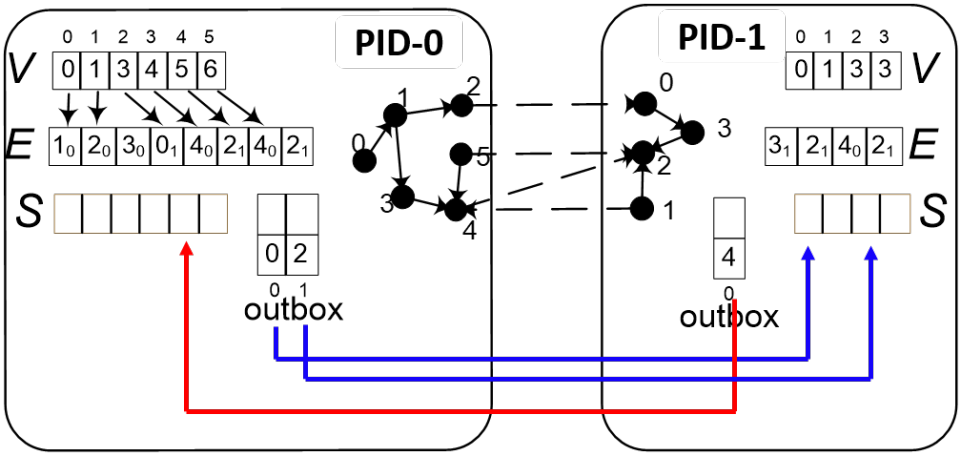


Figure 3.2: High-level illustration of NUMA 1-Box design. *V* and *E* are the buffers to represent the graph in CSR format (as mentioned in Section 2.1 and Figure 2.1). *S* is the state buffer for local vertices. For push-based algorithms, during computation phase, each partition manipulates its local state buffer *S* for local vertices and updates for remote vertices are aggregated locally in the outbox buffer. During communication phase, updates in the remote outbox are sequentially accessed and applied to the respective local state buffers.

This drastically decreases the inter-partition traffic and the communication time.

Drawbacks

(i) ***Communication overhead.*** Since remote vertices are marked and counted during partitioning step, the size of the message buffers remain unaltered during the algorithm execution. Though *message aggregation* leads to reducing the number of messages send, there is still communication overhead when the message buffer is mostly empty, which is often the case while processing for algorithms, like BFS and SSSP, where communication happens via selective edges only in every superstep.

3.3.3 NUMA 1-Box Design

Since NUMA is a shared memory system, instead of having two explicit message boxes, one at source and another at destination, only one buffer can be physically allocated on the partition, and the pointer to the box could be swapped during communication phase. In this design we allocate only one message buffer, at the source, and assign it to outbox, because of the fact that outbox in source partition is inbox in the destination partition. Similar to NUMA 2-Box design, the value stored for remote-edges in E is the index to its entry in the respective outbox buffer.

Following is the BSP-style graph processing in this design for both push-based and pull-based algorithms.

The computation phase for both *push-based* and *pull-based* algorithms are same as NUMA 2-Box design, as in both the cases outbox is allocated on source partition, and only the communication phase differs.

For *push-based* algorithms, in communication phase, the partitions swap the pointer to the respective outbox message buffer with the corresponding remote inbox message buffer's pointer. It does remote sequential access to read the remote updates, and applies them to their local state buffer, as shown in Figure 3.2.

For *pull-based* algorithms, during the communication phase, it writes the new state of its local vertices (which are remote in other partitions), stored in local state buffer S, to the respective inbox buffer (which is a pointer in this design, and points to the outbox buffer in the remote partition - inbox in source partition is outbox in destination partition), by doing remote sequential writes.

Advantages

(i) ***No explicit message transfer.*** This design leads to zero remote memory accesses during computation phase, same as the previous design. In communication phase, it passes only the pointer to the address of physically allocated box, rather than the entire message buffer.
(ii) ***Message aggregation.*** Message aggregation advantage is same as in the NUMA 2-Box design.

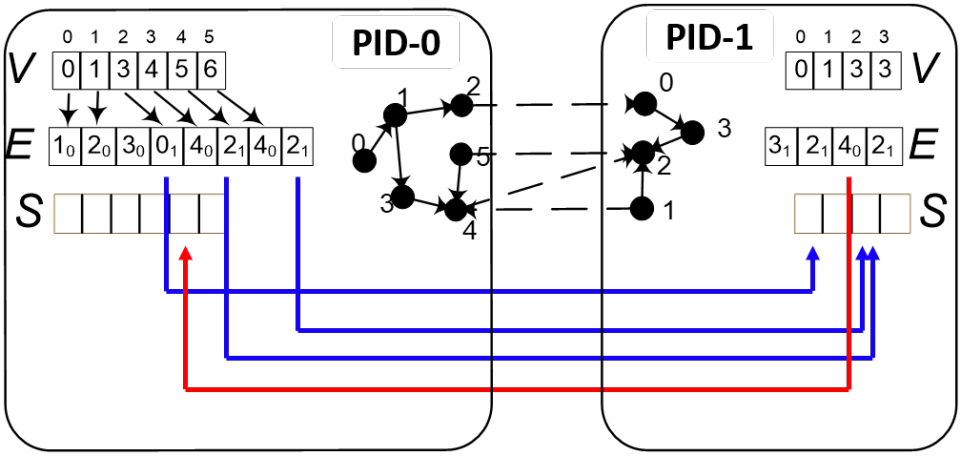


Figure 3.3: High-level illustration of NUMA 0-Box design, which overlaps computation with communication. *S* is the state buffer for local vertices. Note that in this design we get rid of communication infrastructure. During computation phase, each partition manipulates its local state buffer *S* for local vertices and remote updates are directly written to the respective local state buffer of the remote partition. Atomic writes are used to ensure correctness.

Drawbacks

(i) ***Communication overhead.*** Though this design leads to not transferring the message buffer explicitly in the communication phase, all the accesses to the message buffer are remote sequential, which in turn depends on the costly random access to update the local state buffer. Further, similar to NUMA 2-Box design, it suffers from the communication overhead when the message buffer is mostly empty.

3.3.4 NUMA 0-Box Design

In this design we consider the fact that NUMA is a *distributed* shared-memory system. We do explicit partitioning as if NUMA is a distributed system, but we access the state buffers as if we are in a shared-memory system. As shown in Fig. 3.3, we do not use communication infrastructure.

During computation phase, for *push-based* algorithms, if a remote vertex is

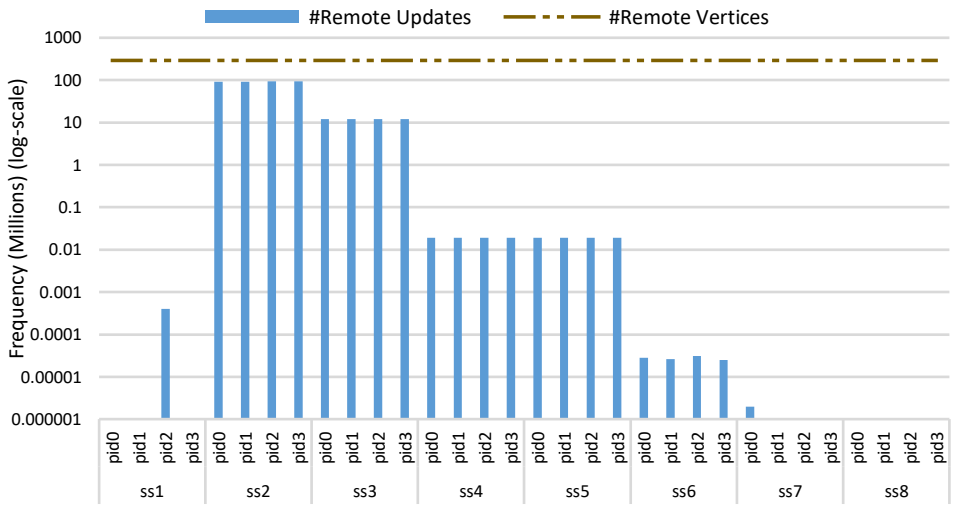


Figure 3.4: Number of Remote vertices vs number of remote updates in each partition in every superstep of Direction-Optimized BFS (BFS-DO) for RMAT31 in NUMA 2-Box design. The Y-axis represents frequency (in millions), **in log-scale**, of the number of remote updates (per superstep) and the total number of remote vertices in each partition. The ss-'x' on X-axis represents the sequence of supersteps. We observe that remote updates are $\sim$**22**$\times$ less than the number of remote vertices.

visited, the state is updated directly in the local state buffer of the respective remote partition, thereby ***it overlaps computation with communication***. Atomic operation is used for updating shared states, to ensure consistency.

For *pull-based* algorithms, like PageRank, during compute phase, each local vertex updates its state (e.g. rank for PageRank) by gathering the state of its incoming-neighbors. For remote incoming-neighbors, it reads the state of the remote vertex from the local state buffer of the respective remote partition.

Note that, in this design, *the state of* ***all the boundary edges*** *is accessed remotely.*

Advantages

(i) ***Overlapping computation with communication.*** This design overlaps computation with communication. It performs better for algorithms like BFS and SSSP,

where communication happens only via a selective set of edges in every superstep. For example, from our experiment, as shown in Fig. 3.4, we observe the number of remote updates in the execution of BFS is $\sim$22x less than the total number of remote vertices.

Drawbacks

(i) ***Communication overhead.*** Since in this design the state of all the boundary edges is accessed remotely, it performs poor for algorithms like PageRank where there is a message via every boundary edge, compared to NUMA 2-Box design, where number of messages equals the number of remote vertices (not edges). ***No message aggregation*** increases the number of remote memory accesses severely.

3.4 Analytical Model for Estimating Performance

To determine the right communication design for an algorithm, we evaluate the three designs analytically for PageRank, as a use case. Table 3.1 presents the memory access pattern of all the designs in computation and communication phases, for the pull-based PageRank algorithm.

In NUMA 2-Box design, during computation phase, all the accesses being local, it sequentially updates the rank of all the local vertices, by doing random accesses to read the value of its neighbors scattered in local memory, including remote neighbors (that are stored in corresponding outbox buffers). In the communication phase, it sequentially updates the recent state of the local vertices, which are remote in other partitions, to the respective inbox buffers. Lastly, it transfers the inbox buffers to the outbox buffers, allocated on respective remote partitions.

In NUMA 1-Box design, access pattern during computation phase is same as NUMA 2-Box design. But in communication phase, since the inbox in source partition points to the outbox in respective remote partition, it performs remote sequential writes to update the new rank of its local vertices (accessed in local-random fashion) to the outbox of remote partitions (where the local vertex is remote).

Further note that, for NUMA 1-Box design, *message buffer could be allocated at destination partition*. But it leads to much expensive *(E' * Random Remote*

NUMA 2-Box Design		
Computation	Communication	
Local Accesses	Local Accesses	Memcopy
V * $\text{Write}_{Seq}^{Local}$ + (E+E') * $\text{Read}_{Rand}^{Local}$	(N-1) * V' * ($\text{Read}_{Rand}^{Local}$ + $\text{Write}_{Seq}^{Local}$)	(N-1) * memcopy()

NUMA 1-Box Design - Box on Source partition		
Computation	Communication	
Local Accesses	Pointer Copy	Local + Remote Accesses
V * $\text{Write}_{Seq}^{Local}$ + (E+E') * $\text{Read}_{Rand}^{Local}$		(N-1) * V' * ($\text{Read}_{Rand}^{Local}$ + $\text{Write}_{Seq}^{Remote}$)

NUMA 0-Box Design	
Overlapped Computation and Communication	
Local Accesses	Remote Accesses
V * $\text{Write}_{Seq}^{Local} + E * Read_{Rand}^{Local}$	E' * $\text{Read}_{Rand}^{Remote}$

Table 3.1: Memory Access Pattern for different NUMA designs, for PageRank algorithm. V/V' and E/E' represents number of local/remote vertices and edges in the partition. N is the number of partitions. Memory accesses are represented by X_Y^Z, where X is: read/write operation, Y is: sequential/random access, and Z is: local/remote memory access.

Read) accesses in computation phase. Therefore, we discard this variation.

Finally, in NUMA 0-Box design, computation and communication phases are overlapped. In computation phase, rank of each local vertex is computed by reading the rank of its incoming neighbors. To access the rank of its remote neighbors, it does Random Remote Read accesses.

Having an analytical model helps in selecting the appropriate communication mode depending on the access pattern of an algorithm. Similar analytical model could be designed for other algorithms by observing the access pattern during both computation and communication phase.

3.5 Mapping GAS model to NUMA design

In this section, we briefly describe how our design matches with the graph computation that follows GAS (Gather-Apply-Scatter) model by considering the NUMA 2-Box design option as an example.

In our design, the graph computation is implemented either as Gather-Apply-Scatter, for *pull-based* algorithms like PageRank, or as Scatter-Apply-Gather, for *push-based* algorithms like BFS and SSSP.

In *pull-based* algorithms, like PageRank, during compute phase, each local vertex *updates* its state (e.g. rank for PageRank) in local state buffer *S*, by *gathering* the state of its neighbors. If the neighbor is a remote vertex, the value in edge array E stores the index to its outbox entry, which contains its state (such as rank in PageRank). For example, as shown in Figure 3.5, in PID-0 (left), vertex 5 *gathers* the rank of its neighbors 4_0 (local vertex) and 2_1 (remote vertex). Similarly, in PID-1, vertex 2 *gathers* the rank of its neighbor 1_1 (local vertex). In *apply* phase, Figure 3.5, the state of the vertex is updated with the new computed rank. Now, since the new state of the vertex needs to be in sync with its shadow copy in remote partition as well, in communication phase it (i) first *scatters*/updates its value in the respective inbox buffer, and then (ii) the inbox buffer is transferred to the outbox buffer on remote partition. In this way, all the shadow copies of a vertex have the same state, before the next superstep starts.

In *push-based* algorithms, like BFS, during computation phase, each active vertex *scatters* its state to its neighbors. For local vertices, the state is *applied*/synced implicitly (to its entry in local state buffer S), but for remote vertices, the state is updated only in their outbox entry. In communication phase, the partitions transfer the respective outbox buffer to the corresponding remote inbox buffer, and then the local vertices *gather* the updated value and commit the change.

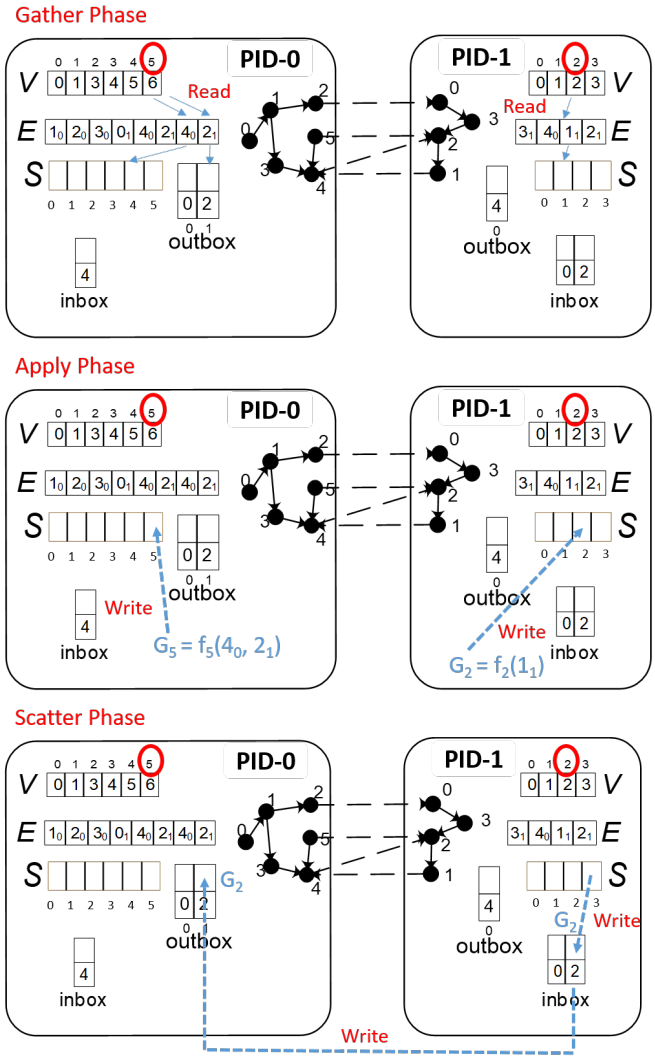


Figure 3.5: Gather, Apply and Scatter phases in NUMA 2-Box design, for a pull based algorithm. During computation phase, each local vertex *gathers* the state of its neighbors and *applies* the computed value to its state. In communication phase, it *scatters* its new state to the respective inbox buffer, which is then copied to the outbox buffer on remote partition. In this way, all the shadow copies of a vertex have the same state, before the start of next superstep.

Chapter 4

Experiment Design

This section describes the implementation of our graph processing framework, our NUMA testbed, the synthetic and real-world workloads (graphs) used, and the experimental methodology we follow.

4.1 System Implementation

To implement our NUMA-aware designs, we extend a state-of-the-art NUMA-oblivious graph processing framework, Totem [15], that presumes SMP based CPUs. It does hybrid graph processing on CPU and GPUs, where GPUs have discrete memory, thereby follows distributed systems design. Similar to distributed systems, it follows Bulk Synchronous Parallel processing model, and does communication between CPU and GPU with message buffers. We use this NUMA-oblivious framework because: **First**, in our previous study [5] we observed that it outperforms state-of-the-art graph processing frameworks including Intel's GraphMat [36] and Galois [30], by up to an order of magnitude. **Second**, and most importantly, its processing model, BSP, matches with our needs.

Fig. 4.1 presents the high level design of our framework. As input, user provides the graph (workload), graph_kernel (e.g. BFS, PageRank), partitioning and communication strategy, along with optimization options. We allocate all the data structures belonging to a partition on its respective NUMA node by using *libnuma* library. To launch the partitions in parallel and do the computation independently,

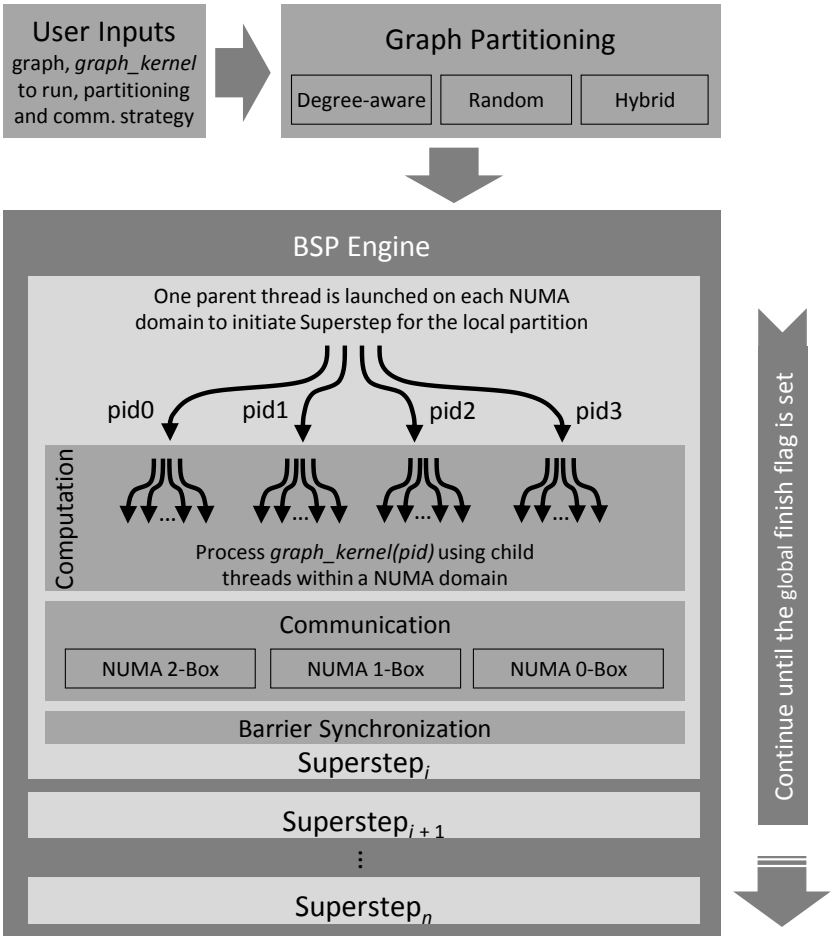


Figure 4.1: High-level design of our BSP-style NUMA-aware framework.

we leverage nested parallelism offered in OpenMP. In the first level of parallelism, we create as many threads as the number of NUMA nodes available, called parent threads. From each of these parent threads, child threads, equal to the numbers of cores available on each NUMA node, are spawned on the respective NUMA nodes. Threads are assigned to the respective numbered cores (i.e. Thread 0 is assigned to Core 0, and so on). We set the OMP_PROC_BIND=spread,close to ensure initial threads are spawned on different NUMA nodes and child threads are close to their respective parent thread. Similarly, during the communication phase, especially

in NUMA 2-Box design, the parent thread on each NUMA node are responsible for transferring the content of outbox to inbox of remote partition, and then child threads applies the updates from inbox to the respective local state buffers. This process continues until the global finish flag is set.

In our experiments, we evaluate how our NUMA-aware framework performs against the state-of-the-art NUMA-oblivious framework Totem. Further, we run the NUMA-oblivious framework with *numactl interleave* command, which allocates memory on all the NUMA nodes in a round robin fashion, instead of Linux's *first-touch* policy, that allocates data on the memory node touched first by the thread. We compare against this as well, as it allocates memory pages *uniformly* on all the NUMA nodes in a *non-deterministic* way.

4.2 TestBed

To explore benefits of our designs we use a four socket Intel Xeon machine (E7-4870 v2, Ivy Bridge), having 60 cores, 1536 GB of Memory and L3 Cache of 120 MB. In Section 2.3.3 we have discussed the key memory characteristics of our testbed using Figure 2.3 and Table 2.1.

4.3 Workload

Graph	#Vertices (|V|)	#Edges (2|E|)	Edge-list size (in GB)
RMAT28	256M	8B	64
RMAT29	512M	16B	128
RMAT30	1B	32B	256
RMAT31	2B	64B	512
RMAT32	4B	128B	1024
Twitter50	51M	3.9B	15
clueWeb12	978M	74B	286

Table 4.1: Workload used for evaluation.

We consider both real-world and large Recursive MATrix (RMAT) scale-free graphs from scale 28 to 32 for evaluating our designs. Twitter [9] is an online

social network graph, while clueWeb12 [3] is a hyperlink web graph. Synthetic graphs are generated using the RMAT generator [12] with the following parameters: (A,B,C) = (0.57, 0.19, 0.19) and an average vertex degree of 16. All the graphs were made undirected, following the Graph500 standard. We use RMAT graphs to evaluate our design because: **First**, it is adopted by today's widely accepted Graph500 benchmark [2]. **Second**, RMAT graphs have similar characteristics to real-world graphs: they have a low diameter and a 'power-law' [14] (highly heterogeneous) vertex degree distribution.

We use 64-bit vertex and edge id to store the RMAT graphs in-memory, as we store partition id in the highest ordered bits. The largest graph we run, RMAT32, has the edge-list of size 1TB (1024 GB). For evaluation, we run the experiments 20 times for each workload and report the average. For BFS and SSSP, we use different randomly generated source vertex. We use 32-bit weights, for SSSP, in the range of (0, 1M] so as to have highly diverse weight distribution. For PageRank, we run each experiment for five PageRank iterations and normalize the execution time to one iteration.

4.4 Experimental Methodology

Partitioning

Explicit partitioning enables implementation and experimentation with different partitioning strategies to achieve better load balancing and data locality. We experiment with the three partitioning strategies that we described in Section 3.2. We define ***load imbalance*** as ratio between computation time of the slowest partition to that of the fastest partition.

Performance Evaluation of Designs

We evaluate the performance of the NUMA designs we introduced in Section 3.3. We compare the performance of our designs with that of NUMA-oblivious framework Totem and running Totem with non-deterministic numactl, and report the algorithm execution time. Consistent with the standard practice in the domain, 'execution time' does not include time spent in pre- or post-processing steps such

as graph loading, graph partitioning and result aggregation. We further evaluate our designs for strong scaling w.r.t resources. For scalability, we consider largest graph that could fit in the memory of one socket (384 GB). For PageRank and BFS we consider RMAT30, with edge-list size 256 GB, and for SSSP we consider weighted RMAT29, with weighted edge-list size 192 GB.

Note: We are unable to provide hardware counters measurement because hardware counters for remote memory accesses are not supported by the testbed.

Accuracy of the Analytical Model

To select appropriate communication design, we have presented an analytical model in Table 3.1 for PageRank. We verify its accuracy.

Comparison with Existing Work - Polymer

Finally, we compare against Polymer [42], the only NUMA-aware single-node graph processing framework (to the best of our knowledge). It has shown to perform better than state-of-the-art single-node graph processing frameworks Galois [30], Ligra [35], and X-Stream [32].

Chapter 5

Experimental Results

This chapter presents and discusses the experimental results of our designs. We first discuss the impact of graph partitioning on load balancing and overall performance improvement. Next, in Section 5.2 we explore the performance of the NUMA designs (described in Chapter 3) for different graph applications (§2.6) on both synthetic and real-world graphs(§4.3). In Section 5.3, we evaluate the accuracy of our analytical model. Finally, we compare our designs with a state-of-the-art NUMA-aware graph processing framework, Polymer (Section 5.4).

5.1 Impact of Graph Partitioning

Explicit partitioning enables designing and experimentation with different partitioning strategies to achieve better load balancing and overall performance improvement. In this section, we evaluate the impact of the three partitioning strategies (discussed in §3.2) on load balancing and overall performance improvement for PageRank (*which has fixed workload in every superstep*) and BFS - Direction Optimized, BFS-DO, (*which has dynamic workload per superstep*) algorithms.

Figure 5.1 shows the impact of our hybrid strategy compared to Random and Sorted/Degree-aware (§3.2) strategies, on load balancing for PageRank using the RMAT31 graph. For PageRank, where workload in every superstep is fixed, Random strategy leads to a load imbalance of only 1.03× (i.e. the slowest partition is only 3% slower than the fastest partition). Sorted/Degree-aware strategy suffers

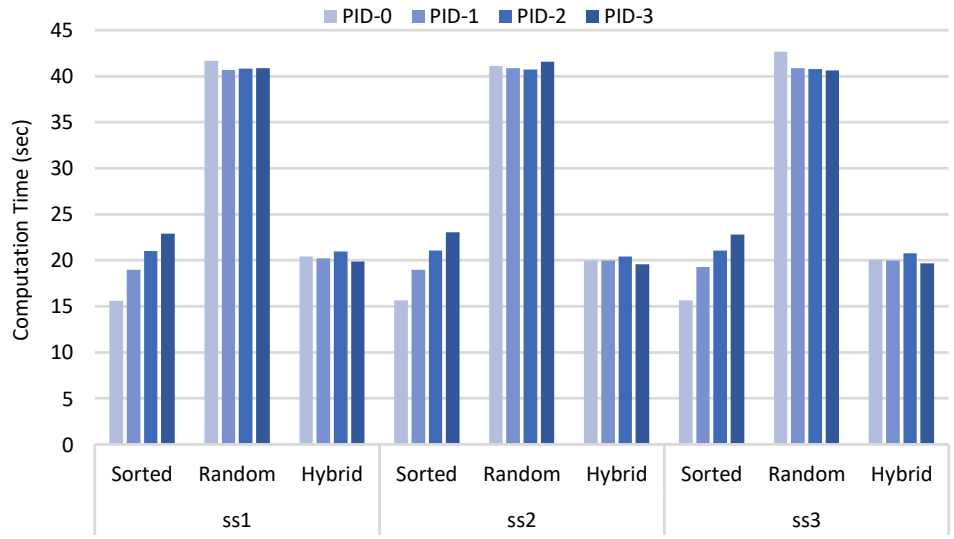


Figure 5.1: Load imbalance of traditional (Sorted and Random) and new Hybrid partitioning strategy for PageRank algorithm for RMAT31 graph using NUMA 2-Box design. The x-axis is for supersteps (denoted by 'ss') required for execution. The y-axis is for computation time (lower the better) of the four partitions for the three partitioning strategies.

a load imbalance of 1.46×, but performs 1.69× better than Random partitioning. This is because, random partitioning strategy increases the probability that each partition has equal variability in terms of vertex degree. Therefore, we observe better load balance. On the other hand, it also increases the probability that the neighbors of a vertex are scattered in memory, leading to poor data locality [16]. While with Sorted strategy, first partition gets the most dense graph (containing few high degree vertices, for e.g., PID-0 in Figure 5.1) and the last partition gets the most sparse graph (containing most of the vertices with low degree). Sorted strategy leads to better data locality and over all performance for PageRank, but since dense partition gets processed faster than the sparse partition, it leads to higher load imbalance compared to Random strategy.

Hybrid strategy achieves load imbalance of only 1.05×. This leads to an overall performance improvement of 1.18× and 2× against Sorted and Random strategies, respectively.

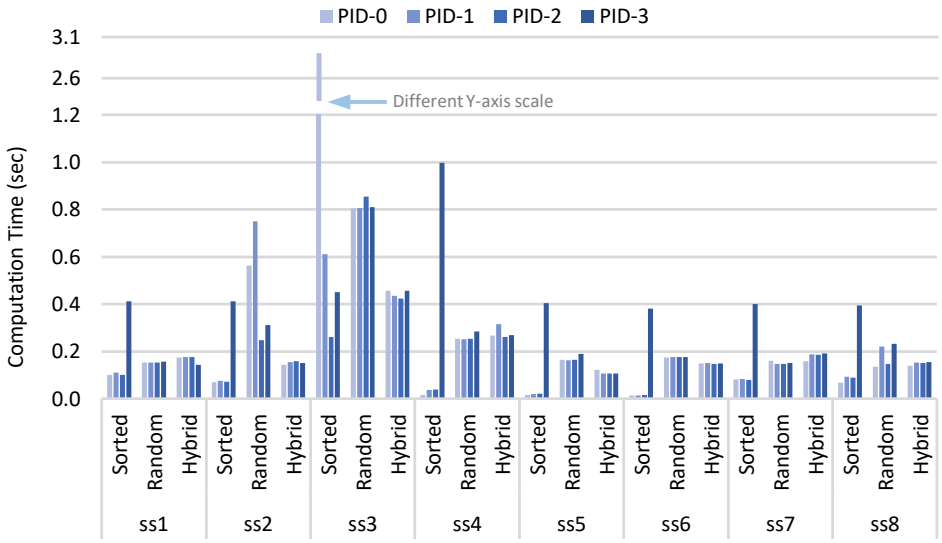


Figure 5.2: Load imbalance of traditional (Sorted and Random) and new Hybrid partitioning strategy for BFS-DO algorithm for RMAT31 graph using NUMA 2-Box design. The x-axis is for supersteps (denoted by 'ss') required for execution. The y-axis is for computation time (lower the better) of the four partitions for Traditional and New partitioning strategies.

For BFS-DO, where workload changes drastically in every superstep, as shown in Figure 5.2, we observe significantly higher load imbalance, of 10.1×, with Sorted strategy. With Sorted strategy, initial three supersteps are executed with Top-down kernel, followed by three supersteps with Bottom-up kernel, and the remaining again with Top-down kernel. In Top-down stage, frontier builds up quickly for dense partition (since it has high degree vertices), hence we observe that in superstep 3, the dense partition (PID-0) takes significant amount of time because of processing the huge frontier. Random strategy achieves load imbalance of 1.35×. *Even for algorithms like BFS-DO, where workload during every superstep is highly dynamic, Hybrid strategy achieves load imbalance of only 1.13×.* Since, BFS-DO is a memory bound algorithm and cache sensitive, better load balance and locality leads to better performance. Random strategy performs 3.4× better than Sorted strategy, while *Hybrid strategy performs 5.3× and 1.55× better than Sorted and*

Random strategies, respectively.

Since our hybrid strategy achieves better overall performance, in all the following experiments, we use our hybrid partitioning strategy to partition the graph for NUMA-aware graph processing.

5.1.1 Partitioning - Key Insights

1. Better load balance ***does not*** mean better performance (as observed in Random vs Sorted strategy). Note that it is important to achieve better load balance, to optimize resource usage (i.e. to avoid overload of few resources while the remaining resource are idle). But, the end goal of high performance graph processing is to process the graph as fast as possible.

2. The hybrid partitioning strategy strikes the right balance between load balance and locality, hence offers improved performance.

3. Graph partitioning is an NP-Complete problem. There exists sophisticated partitioning strategies that offer improved load balancing across partitions, however, are costly. For them, graph partitioning takes much longer than the simpler partitioning techniques we explored in this dissertation.

4. Real-world graphs are heavily skewed, therefore are very difficult to partition. The partitioning strategies, leveraged by distributed graph processing frameworks like Google's Pregel and GraphLab, and the ones we have experimented with are simple and low-cost. These techniques make the hypothesis that there are no natural communities in these real-world graphs. But we are skeptical that there are natural clusters in many of these graphs.

5. Further, our infrastructure is flexible enough that users can plugin and experiment with different partitioning strategies.

5.2 Performance Evaluation of Designs

In this section we first evaluate the performance of our three communication designs on both synthetic and real-world workloads for different graph algorithms.

Then we provide the strong scaling, w.r.t resources, experiment results of our designs. Finally we mention the performance numbers of our Graph500 submissions.

5.2.1 Performance of NUMA 2-Box design.

As observed in Fig. 5.3, for the RMAT31 graph, NUMA 2-Box is 2.07× and 1.63× faster than NUMA-oblivious Totem for PageRank and BFS-DO algorithms, respectively.

Since PageRank has a high compute-to-memory-access ratio, most of the time is spend in computation phase. Further, because of having explicit 2-Box communication, *all the remote updates are send in a batch and all the accesses are local (in both computation and communication phase).* This leads to spending only 3% of execution time in communication phase.

In BFS-DO, which has a low compute-to-memory-access ratio, as shown previously in Fig. 3.4, relatively few remote vertices have messages in each superstep. This leads to higher communication cost of 26.9% of execution time. *numactl* does not provide enough performance because the pages are distributed among NUMA nodes in a non-deterministic round-robin fashion, thereby the data distribution is not graph topology aware.

Further, Fig. 5.4 and Fig. 5.5, present that (i) for **synthetic graphs**, NUMA 2-Box performs better than both Totem and numactl for all the algorithms (up to **2.08×, 1.88×, and 1.91×** against Totem, and **20%, 26%, and 33%** better than numactl, for PageRank, BFS-DO and BFS-TD, respectively) except for SSSP. For SSSP, it performs up to 33% better than Totem, but is up to 91% slower than numactl (discussed later). (ii) For **Twitter graph**, NUMA 2-Box is up to 63%, 29%, and 11% faster for PageRank, BFS-TD, and SSSP algorithms, respectively, against Totem. Real-world graphs like **clueWeb** are heavily skewed and require huge number of supersteps to converge (**#supersteps:** 94 and 135 for BFS-DO, and 76 and 129 for SSSP, by Totem and NUMA 2-Box, respectively, for clueWeb). For clueWeb graph, NUMA 2-Box could achieve good performance, of 69%, only for PageRank. For traversal based algorithms, especially for BFS-DO and SSSP, it does not perform well. **BFS-DO** *requires hand-tuning the parameters to switch between Top-down and Bottom-up stages. That's why we do not observe major*

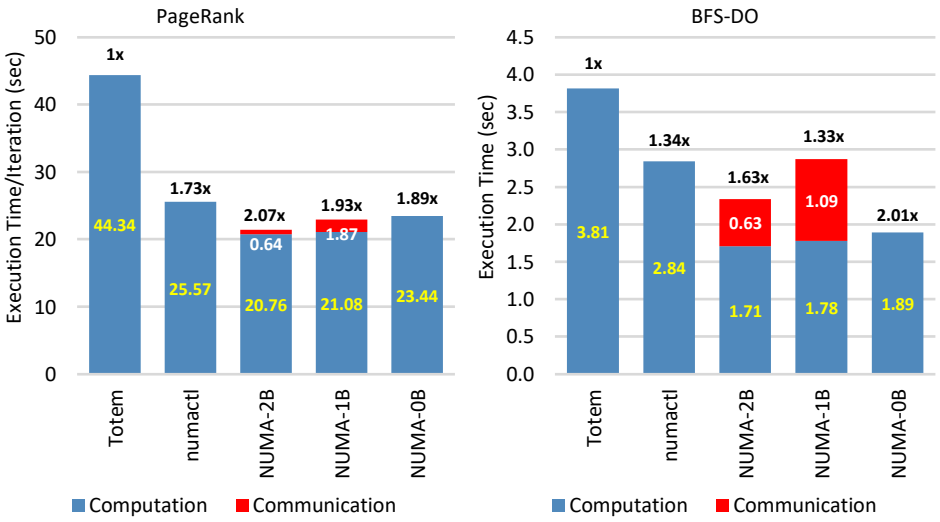


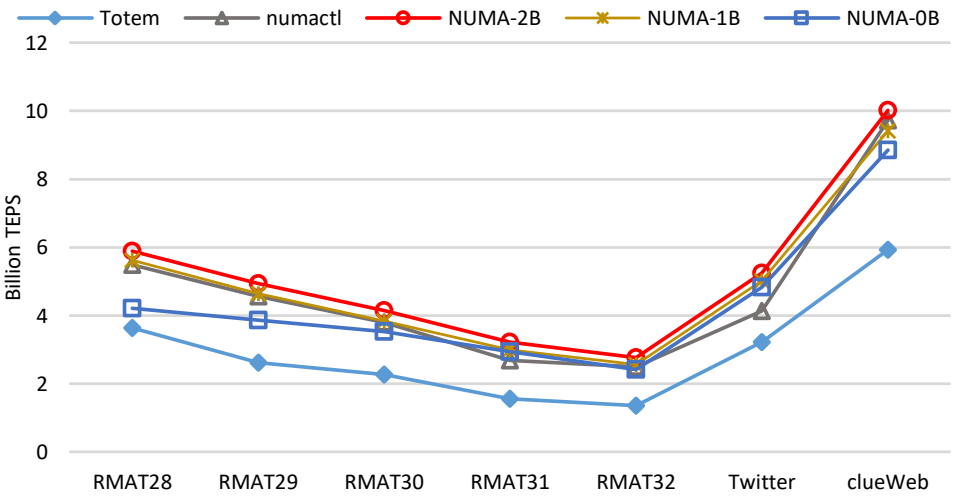
Figure 5.3: NUMA designs performance against Totem and numactl for PageRank (left) and BFS-DO (right) algorithms on RMAT31 graph. The Y-axis is for execution time (lower the better). For NUMA-2B and NUMA-1B, where we do explicit communication, we show the break-down of execution time with computation and communication time.

improvement. Since all the scales of synthetic graphs have similar characteristics, the switching parameters are easy to tune.

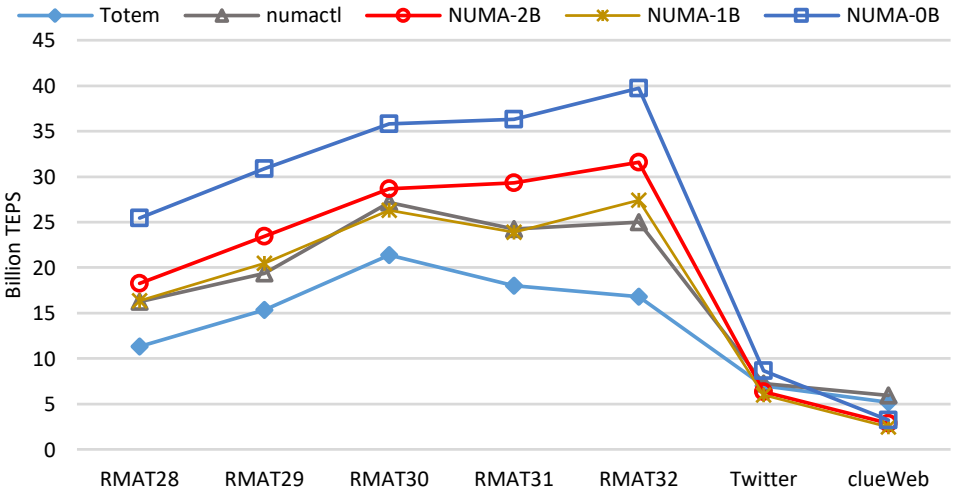
For both type of workloads (synthetic and real-world), numactl performs better than NUMA 2-Box design for **SSSP**. As mentioned earlier, for SSSP, the optimizations, to activate the neighbors in the same iterations, were done to reduce the number of supersteps assuming SMP based architecture [16]. *This leads to the partition with source vertex spend more time in the computation phase than others, in the initial few supersteps*. **Note that our NUMA-aware design is application agnostic and we do not modify the applications.**

5.2.2 Performance of NUMA 1-Box design.

NUMA 1-Box design performs better than Totem and is competitive with numactl. Though, it does not perform well compared to NUMA 2-Box because it consumes more time during communication phase, as it does remote sequential accesses to

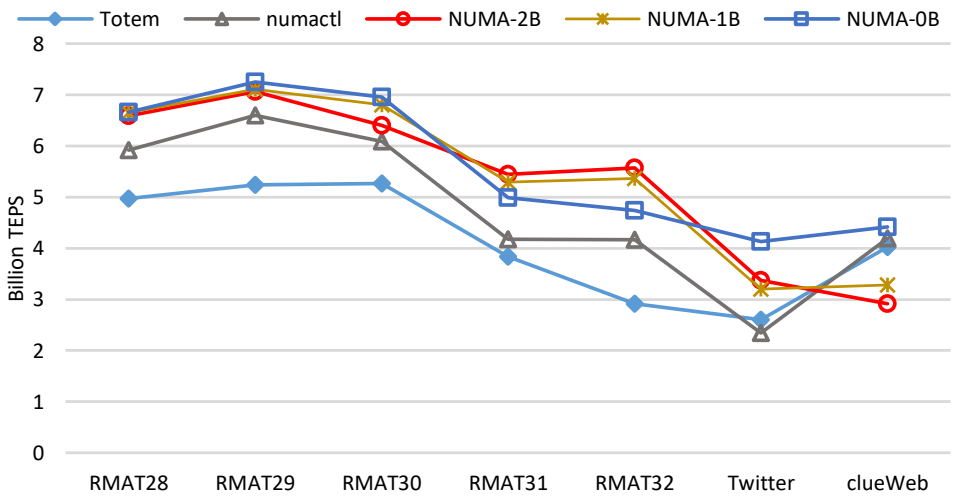


(a) PageRank

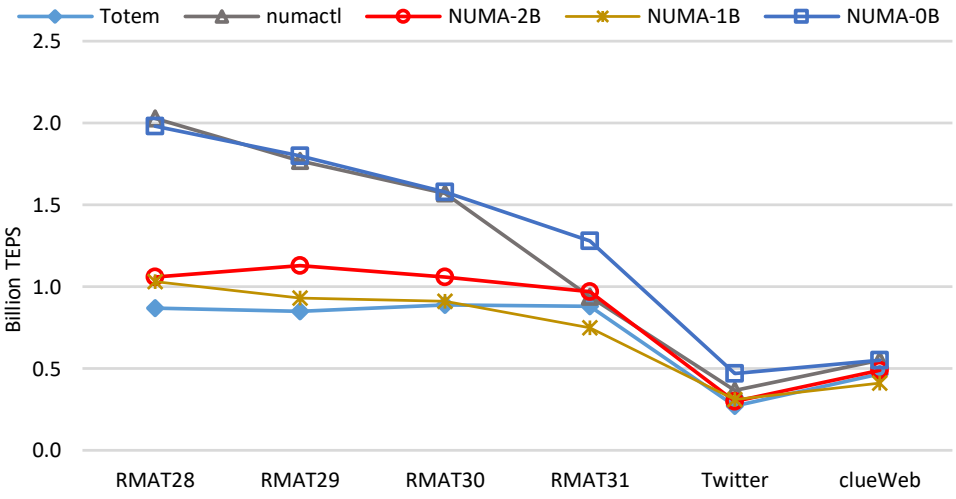


(b) BFS-DO

Figure 5.4: Billion Traversed Edges Per Second (TEPS) achieved by Totem, numactl and the NUMA designs, for (a) PageRank and (b) BFS-DO algorithms on RMAT[28-32] (synthetic), and Twitter and clueWeb (real-world) workloads. Note, the Y-axis is for Traversed Edges Per Second (TEPS) (higher the better).



(a) BFS-TD



(b) SSSP

Figure 5.5: Billion Traversed Edges Per Second (TEPS) achieved by Totem, numactl and the NUMA designs, for (a) BFS-TD and (b) SSSP algorithms on RMAT[28-32] (up to RMAT31 for SSSP, since it requires weighted edge-list) (synthetic), and Twitter and clueWeb (real-world) workloads. Note, the Y-axis is for Traversed Edges Per Second (TEPS) (higher the better).

the remote message buffer, which is bounded by slow local random updates to the state buffer, as shown in Figure 5.3. In NUMA 1-Box design too, all the remote updates are aggregated in message buffers, in similar way as in NUMA 2-Box design. Hence, it does not perform well for BFS-DO and SSSP algorithms, but performs well for PageRank and BFS-TD compared to both Totem and numactl, as shown in Figure 5.4 and Figure 5.5.

5.2.3 Performance of NUMA 0-Box design.

NUMA 0-Box design performs better for algorithms like BFS and SSSP, where in each superstep there are messages from selective boundary edges only, not from all. As shown in Fig. 5.3, for BFS-DO, *overlapping computation and communication (by directly updating the remote vertices) in every superstep leads to better performance.* Note that the gain, almost equivalent to the communication time in NUMA 2-Box, is achieved because of implicit communication, since the number of remote updates in every superstep is much less than the number of remote vertices ($\sim$22$\times$ for RMAT31 graph), as shown previously in Fig. 3.4.

From Fig. 5.3, Fig. 5.4 and Fig. 5.5, we observe that NUMA 0-Box design performs better than Totem, numactl as well as other NUMA designs for **PageRank (up to 1.89$\times$), BFS-TD (up to 1.62$\times$), BFS-DO (up to 2.37$\times$) as well as SSSP (up to 2.27$\times$)** algorithms. Further, even though real-world graphs are heavily skewed, it provides better performance improvement, for traversal based algorithms, than other NUMA designs as well as Totem and numactl (except for **BFS-DO on clueWeb** graph, which requires hand-tuning the switching parameters).

For **BFS-TD**, though it performs better, performance gain is less compared to other NUMA designs. This is because in BFS-TD frontier size increases drastically, which increases the remote memory accesses. But, for **SSSP**, *it activates the remote vertices as well, in the same iteration, which increases the likelihood of every partition having active vertices in the initial supersteps, thereby achieving better load balance and overall performance.*

Further, for algorithms like **PageRank**, where each vertex calculates its rank by pulling ranks of all its neighbors, *all the remote messages, sent over every boundary edge, are read/accessed in remote random fashion, which leads to degradation in*

performance.

5.2.4 Communication Designs - Key Insights

1. Although explicit communication can be perceived as extravagant for a cache-coherent shared memory system, its performance benefits on a NUMA system are indisputable.

2. Performance gain in NUMA 2-Box, compared to NUMA 1-Box, comes from doing local accesses during computation, and copying data in bulk from source partition to destination partition, followed by local random/sequential accesses for local read/write operations in communication phase. *But in NUMA 1-Box design, even though the remote updates are read sequentially, it is bounded by slow local random writes to the local state buffers.*

3. NUMA 2-Box design leads to zero remote memory accesses and reduces the number of messages send during the communication phase to only the number of remote vertices, regardless of the number of edges associated with them.

4. NUMA 2-Box performs better for algorithms, like PageRank, where the communication volume is high (i.e. where most of the neighbors are updated). While, for algorithms which have low communication volume (i.e. where only a small subset of neighbors are updated), like traversal based algorithms such as BFS and SSSP, NUMA 0-Box provides better performance. *This explains why some algorithms are finding better solution with one design and some with other design.*

5. For BFS and SSSP, the partition with source vertex ends up spending more time in initial supersteps in NUMA 2-Box and 1-Box designs, as the active vertices are confined to the partition having the source vertex. This degrades the performance of NUMA 2-Box and 1-Box designs for these algorithms. On the other hand, for PageRank, NUMA 0-Box ends up doing remote random access for each remote edge, hence no message reduction like NUMA 2-Box design, which degrades its performance.

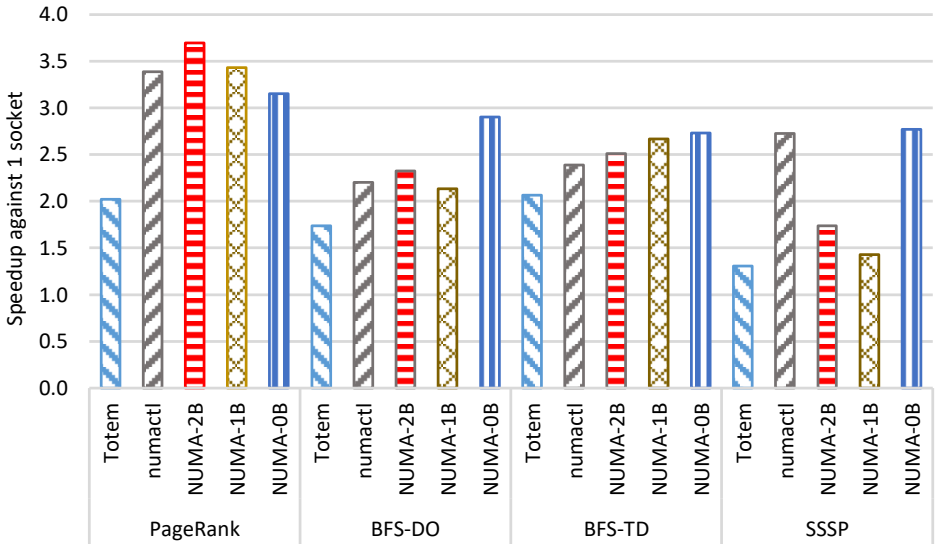


Figure 5.6: Strong Scaling of Totem, numactl and NUMA designs on our 4-socket machine compared to 1-socket (memory: 384GB). Weighted RMAT29 (weighted edgelist size: 192GB) is used for SSSP and unweighted RMAT30 (edgelist size: 256GB) was used for all other algorithms.

5.2.5 Strong Scaling Experiments

Here we evaluate how our designs scale on our four socket NUMA testbed compared to one socket. We use the largest graph that could fit into the memory of one socket (384 GB). For SSSP, which requires weighted graph, we use RMAT29 with weighted edge list size of 192 GB. For all other algorithms we use unweighted RMAT30 with edge list size 256 GB.

Fig. 5.6 presents that our NUMA design fills the performance gap left by Totem by scaling to as much as 3.7×, 2.9×, 2.7× and 2.8× compared to 2.0×, 1.7×, 2.1×, and 1.3× achieved by Totem for PageRank, BFS-DO, BFS-TD and SSSP algorithms, respectively.

5.2.6 Graph500 submissions.

Graph500 competition ranks supercomputers worldwide for data intensive applications, BFS and SSSP.

In **SSSP**, a new kernel addition in Graph500 since November 2017, we secured ***World Rank 2*** (in June 2018 list, published during ISC conference).

For **BFS**, a mature kernel in Graph500, we rank among top three single-node submissions. For RMAT31, our NUMA 0-Box design achieved 10.73 Billion TEPS, a performance gain of 28% against our previous submission by running NUMA-oblivious Totem with numactl on the same NUMA machine, 8.37 Billion TEPS. Note that, this performance improvement is less compared to 50% performance improvement of NUMA 0-Box against numactl for plain BFS-DO for the same RMAT31 workload, as shown in Figure 5.3. This is because, as expected, our NUMA design consumes more time in initialization (which is timed in Graph500) than NUMA-oblivious, as it has to initialize the state of all the partitions. *Further, the largest graph submission we made has 128 Billion undirected edges (RMAT32), with edge list size 1TB. NUMA-oblivious Totem could not run RMAT32 as its memory requirement are $\sim 2\times$ the edge-list size, as observed in our previous study [5].*

Note that our framework is a generic graph processing framework that enables users to develop multiple applications, including BFS and SSSP, and applies optimizations in an application-agnostic way. While, the codes we compete against in Graph500 are developed for these specific applications (as published in the corresponding publications [13, 38, 40, 41]).

5.3 Accuracy of the Analytical Model

From our experiments, we have observed that different designs perform differently depending on the *memory access pattern they provide*, along with the *communication volume in different graph algorithms*. For example, NUMA 2-Box design, which offers all the accesses to be local and does message aggregation for remote vertices, performs best for algorithms having messages for *most of the remote vertices* in each superstep, like PageRank. While, NUMA 0-Box, which overlaps computation with communication, gives best performance for algorithms like BFS and SSSP, where there are messages over selective edges only.

We calculate the expected theoretical time to solution for each of the three designs for PageRank algorithm using the machine characteristics available in Table 2.1 for different workloads. We observed that for all the RMAT graphs and for

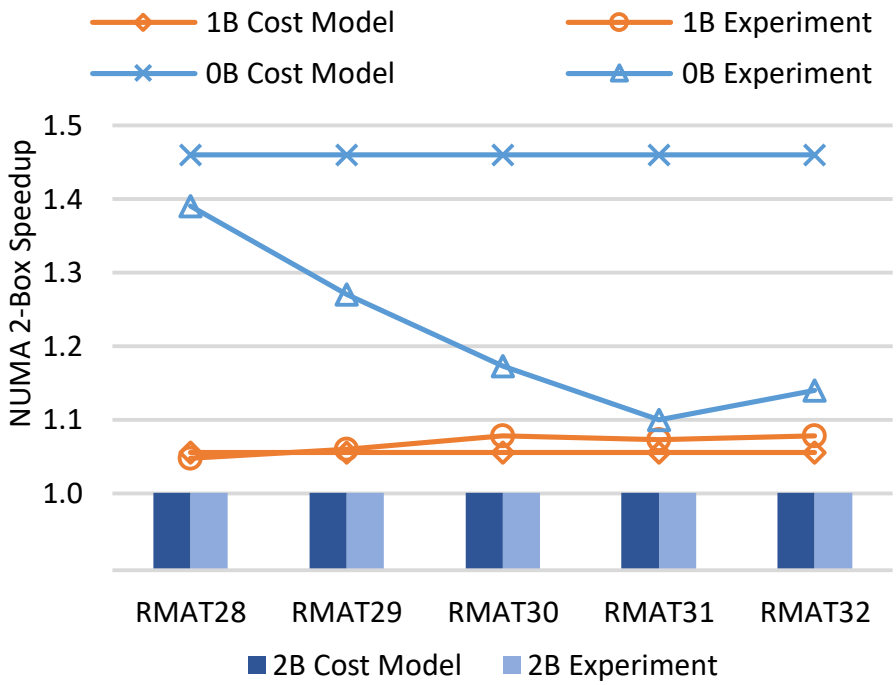


Figure 5.7: Analytical Model prediction for PageRank algorithm. The Y-axis shows the speedup of NUMA 2-Box against NUMA 1-Box and NUMA 0-Box as predicted by cost model and calculated empirically through experiments.

all the partitions, ∼25% of the edges were local and ∼75% were remote (since we have four partitions, with random distribution, probability of a vertex to be local is 0.25, and 0.75 for being remote), and local vertices and remote/ghost vertices constitute ∼64% and ∼36% of the total vertices (V+V') in each of the partitions. As seen in Fig. 5.7 and in Table 5.1, our model ***correctly predicts the relative sequence of the designs according to the performance***. *Note that this is a high level prediction, since we have not taken into account caching, effect of prefetching and other minute level details; and especially for NUMA 0-Box design, we did not take into account impact of overlapping computation and communication in the analytical model.*

Design	Cost Model	Empirical
NUMA 2-Box	1×	1×
NUMA 1-Box on source partition	1.055×	1.07×
NUMA 0-Box	1.46×	1.21×

Table 5.1: Cost Model evaluation for PageRank algorithm. The numbers represent average speedup of NUMA 2-Box against other designs, for all workloads, as predicted by cost model and observed empirically from experiments.

Application developers can easily extend this model for other applications to determine the optimal communication strategy, since in all the graph applications, we only need to look at the memory access pattern as specified in Table 3.1.

5.4 Comparison with Existing Work

Finally, we compare our framework against Polymer, the only NUMA-optimized graph processing framework we are aware of. It also embraces the design philosophy of distributed systems, and is developed on top of Ligra, a shared-memory graph processing framework. The key difference between Polymer and our work is, Polymer does vertex-cut partitioning [17], while we do edge-cut partitioning. In vertex-cut partitioning, a single vertex is partitioned among multiple nodes. This requires replicating the vertices on multiple nodes. But, it enables distributing the computation on a single vertex over multiple NUMA nodes. This is especially beneficial for highly skewed real-world power-law graphs where few vertices have millions of edges associated with them. This technique further provides the advantage of placing only those edges on a partition whose destination vertex is also local to the partition, thereby both the source and the destination vertices are local. This reduces the number of remote memory accesses. Polymer implements push-based PageRank and adapts Bellman-Ford algorithm for SSSP. For BFS, it implements Top-Down BFS, and does not supports Direction-Optimized BFS.

Table 5.2 summarizes the performance of the best performing NUMA design against Polymer. For BFS, we present the performance of our design for BFS-Top Down only, since Polymer does not have Direction-Optimized BFS implementation. Missing data points means Polymer failed (Error: *Segmentation fault (core*

Algorithm	Workload	Polymer		NUMA-xB	
		Time (s)	Memory (GB)	Time (s)	Memory (GB)
PageRank	RMAT28	5.1	365	1.46	80
	RMAT29	11.6	735	3.48	162
	RMAT30	26.2	1401	8.29	330
	RMAT31	-	-	21.4	674
	RMAT32	-	-	49.4	1366
	Twitter	1.53	144	0.75	21.8
BFS	RMAT28	11.37	366	1.29	81
	RMAT29	17.17	740	2.36	162
	RMAT30	34.63	1302	4.94	322
	RMAT31	-	-	12.63	653
	RMAT32	-	-	24.65	1319
	Twitter	13.1	93	0.94	19.2
SSSP	RMAT28	11.5	437	4.34	115
	RMAT29	24.95	886	9.56	232
	RMAT30	-	-	21.79	452
	RMAT31	-	-	53.59	867
	Twitter	5.3	115	8.4	41

Table 5.2: Execution time (in second) and peak Memory consumption (in GB) of Polymer and our best performing NUMA design (NUMA-xB). We show peak memory consumption among all the NUMA designs. Missing data points means Polymer was out-of-memory.

dumped)) to execute for the respective graph workloads (including clueWeb graph), as it was out of memory. Our design outperforms Polymer by up to **3.5×, 13.9× and 2.6× for PageRank, BFS and SSSP algorithms** respectively. Polymer does *vertex-cut* partitioning, and consumes ∼**5.7**× more memory than the size of the respective edge-list of the graph, and ∼**4.4**× **more memory than our NUMA designs**. Polymer is faster than our design only on Twitter for SSSP algorithm. It performs 1.58× better but at the cost of consuming 2.8× more memory.

Chapter 6

Conclusion

In this dissertation, we postulated our hypothesis that a distributed-memory like middleware, that makes graph partitioning and inter-partition communication explicit, improves the performance on a NUMA system. To test our hypothesis, we designed a NUMA-aware graph processing framework that embraces the design philosophies of distributed graph processing framework, especially the explicit partitioning and communication. Based on the lessons from the above design, we proposed (i) a **new hybrid partitioning strategy**, which leads to **near optimal load balance of 95%** and improves the **overall performance by up to 5.3**×, and (ii) a **new NUMA-aware hybrid design** that considers the fact that **NUMA is a *distributed* shared-memory architecture**. It leverages the benefits of distributed system (by explicitly partitioning the graph among NUMA nodes) and shared-memory system (by performing implicit communication to access the state buffers), and **overlaps computation with communication**. This design provides **performance gain of 1.89**×**, 2.37**×**, and 2.27**× for PageRank, BFS and SSSP algorithms, respectively, against state-of-the-art NUMA-oblivious framework, as well as secured good ranking for us in Graph500 competition.

To summarize, we presented a **scalable (up to 3.7**×**), high performant (up to 13.9**×**), memory efficient (**∼**4.4**×**)**, **generic** NUMA-aware graph processing framework that outperforms the state-of-the-art **NUMA-oblivious (up to 2.37**×**)** as well as **NUMA-aware (up to 13.9**×**)** graph processing frameworks. We observed that considering NUMA as a distributed system not only improves performance but

also provides the opportunity to explore different partitioning and communication strategies in NUMA machine. *Finally, since now-a-days each node in a high-end large-scale distributed system embraces NUMA architecture with at least two sockets, our design has the potential to improve the performance of the entire cluster, by improving the performance of each node.*

Bibliography

[1] *Ford, L.A. 1956*. URL NetworkFlowTheory.ReportP-923. → page 17

[2] *Graph500*. URL https://graph500.org/. → pages 8, 14, 16, 37

[3] *Laboratory for Web Algorithmics*. URL http://law.di.unimi.it/datasets.php. → pages 8, 37

[4] *Web Data Commons - Hyperlink Graph*. URL http://webdatacommons.org/hyperlinkgraph/. → pages 1, 8

[5] T. K. Aasawat, T. Reza, and M. Ripeanu. How well do cpu, gpu and hybrid graph processing frameworks perform? In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 458–466, May 2018. doi:10.1109/IPDPSW.2018.00082. → pages 14, 19, 34, 50

[6] A. Abou-Rjeili and G. Karypis. Multilevel algorithms for partitioning power-law graphs. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, pages 124–124, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 1-4244-0054-6. URL http://dl.acm.org/citation.cfm?id=1898953.1899055. → page 21

[7] K. Andreev and H. Räcke. Balanced graph partitioning. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, pages 120–124, New York, NY, USA, 2004. ACM. ISBN 1-58113-840-7. doi:10.1145/1007912.1007931. URL http://doi.acm.org/10.1145/1007912.1007931. → page 21

[8] L. Backstrom, P. Boldi, M. Rosa, J. Ugander, and S. Vigna. Four degrees of separation. In *Proceedings of the 4th Annual ACM Web Science Conference*, WebSci '12, pages 33–42, 2012. ISBN 978-1-4503-1228-8. → page 1

[9] A. S. Badashian and E. Stroulia. Measuring user influence in github: The million follower fallacy. In *Proceedings of the 3rd International Workshop on CrowdSourcing in Software Engineering*, CSI-SE '16, pages 15–21, 2016. ISBN 978-1-4503-4158-5. → page 36

[10] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 12:1–12:10, 2012. ISBN 978-1-4673-0804-5. → page 16

[11] M. Cha, H. Haddadi, F. Benevenuto, and K. P. Gummadi. Measuring user influence in twitter: The million follower fallacy. In *in ICWSM 10: Proceedings of international AAAI Conference on Weblogs and Social*, 2010. → page 1

[12] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *Proceedings of the Fourth SIAM Int. Conf. on Data Mining*, page p. 442. Society for Industrial Mathematics, 2004. → page 37

[13] Z. Cui, L. Chen, M. Chen, Y. Bao, Y. Huang, and H. Lv. Evaluation and optimization of breadth-first search on numa cluster. In *2012 IEEE International Conference on Cluster Computing*, pages 438–448, Sept 2012. doi:10.1109/CLUSTER.2012.29. → page 50

[14] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '99, pages 251–262, 1999. ISBN 1-58113-135-6. → pages 7, 21, 22, 37

[15] A. Gharaibeh, L. Beltrão Costa, E. Santos-Neto, and M. Ripeanu. A yoke of oxen and a thousand chickens for heavy lifting graph processing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 345–354, 2012. ISBN 978-1-4503-1182-3. → pages 2, 14, 18, 21, 22, 34

[16] A. Gharaibeh, E. Santos-Neto, L. B. Costa, and M. Ripeanu. Efficient large-scale graph processing on hybrid CPU and GPU systems. *CoRR*, abs/1312.3018, 2013. → pages 15, 17, 23, 40, 44

[17] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and*

Implementation, OSDI'12, pages 17–30, 2012. ISBN 978-1-931971-96-6. → pages 1, 8, 12, 14, 21, 52

[18] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh. Wtf: The who to follow service at twitter. In *Proceedings of the 22Nd International Conference on World Wide Web*, WWW '13, pages 505–514, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2035-1. doi:10.1145/2488388.2488433. URL http://doi.acm.org/10.1145/2488388.2488433. → page 1

[19] T. Ideker, O. Ozier, B. Schwikowski, and A. F Siegel. Ideker t, ozier o, schwikowski b, siegel afdiscovering regulatory and signalling circuits in molecular interaction networks. bioinformatics 18(suppl 1):s233-s240. 18 Suppl 1:S233–40, 02 2002. → page 1

[20] G. Iori, G. D. Masi, O. V. Precup, G. Gabbi, and G. Caldarelli. A network analysis of the italian overnight money market. *Journal of Economic Dynamics and Control*, 32(1):259 – 278, 2008. ISSN 0165-1889. doi:https://doi.org/10.1016/j.jedc.2007.01.032. URL http://www.sciencedirect.com/science/article/pii/S0165188907000474. Applications of statistical physics in economics and finance. → page 1

[21] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. B. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *CoRR*, abs/1408.5093, 2014. URL http://arxiv.org/abs/1408.5093. → page 18

[22] T. Kissinger, T. Kiefer, B. Schlegel, D. Habich, D. Molka, and W. Lehner. Eris: A numa-aware in-memory storage engine for analytical workload. In *ADMS@VLDB*, 2014. → page 18

[23] K. Lang. Finding good nearly balanced cuts in power law graphs. *Tech. Rep. Yahoo! Research Labs*, Nov. 2004. → page 21

[24] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *CoRR*, abs/0810.1355, 2008. URL http://arxiv.org/abs/0810.1355. → page 21

[25] Y. Li, I. Pandis, R. Müller, V. Raman, and G. M. Lohman. Numa-aware algorithms: the case of data shuffling. In *CIDR*, 2013. → page 18

[26] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, 2010. ISBN 978-1-4503-0032-2. → pages 1, 8, 21

[27] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, Dec. 1995. → page 10

[28] D. Mhembere, D. Zheng, C. E. Priebe, J. T. Vogelstein, and R. Burns. Knor: A numa-optimized in-memory, distributed and semi-external-memory k-means library. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '17, pages 67–78, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4699-3. doi:10.1145/3078597.3078607. URL http://doi.acm.org/10.1145/3078597.3078607. → page 18

[29] N. Nagarajan and M. Pop. Sequence assembly demystified. *Nature Reviews Genetics*, 14:157–167, 2013. → page 1

[30] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471, 2013. ISBN 978-1-4503-2388-8. → pages 1, 14, 15, 18, 34, 38

[31] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web, 1999. → page 15

[32] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 472–488, 2013. ISBN 978-1-4503-2388-8. → page 38

[33] P. Roy, S. L. Song, S. Krishnamoorthy, A. Vishnu, D. Sengupta, and X. Liu. Numa-caffe: Numa-aware deep learning neural networks. *ACM Trans. Archit. Code Optim.*, 15(2):24:1–24:26, June 2018. ISSN 1544-3566. doi:10.1145/3199605. URL http://doi.acm.org.ezproxy.library.ubc.ca/10.1145/3199605. → page 18

[34] S. Sallinen, A. Gharaibeh, and M. Ripeanu. Accelerating direction-optimized breadth first search on hybrid architectures. *CoRR*, abs/1503.04359, 2015. URL http://arxiv.org/abs/1503.04359. → page 16

[35] J. Shun and G. E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 135–146, 2013. ISBN 978-1-4503-1922-5. → pages 1, 14, 15, 18, 38

[36] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey. Graphmat: High performance graph analytics made productive. *Proc. VLDB Endow.*, 8(11):1214–1225, July 2015. ISSN 2150-8097. → pages 14, 34

[37] A. Tizghadam and A. Leon-Garcia. A graph theoretical approach to traffic engineering and network control problem. In *2009 21st International Teletraffic Congress*, pages 1–8, Sept 2009. → page 1

[38] K. Ueno, T. Suzumura, N. Maruyama, K. Fujisawa, and S. Matsuoka. Extreme scale breadth-first search on supercomputers. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 1040–1047, Dec 2016. doi:10.1109/BigData.2016.7840705. → page 50

[39] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, Aug. 1990. ISSN 0001-0782. → page 1

[40] Y. Yasui, K. Fujisawa, and K. Goto. Numa-optimized parallel breadth-first search on multicore single-node system. In *Proceedings of the 2013 IEEE International Conference on Big Data, 6-9 October 2013, Santa Clara, CA, USA*, pages 394–402, 2013. doi:10.1109/BigData.2013.6691600. URL https://doi.org/10.1109/BigData.2013.6691600. → pages 19, 50

[41] Y. Yasui, K. Fujisawa, and Y. Sato. Fast and energy-efficient breadth-first search on a single numa system. In *Proceedings of the 29th International Conference on Supercomputing - Volume 8488*, ISC 2014, pages 365–381, New York, NY, USA, 2014. Springer-Verlag New York, Inc. ISBN 978-3-319-07517-4. doi:10.1007/978-3-319-07518-1_23. URL http://dx.doi.org/10.1007/978-3-319-07518-1_23. → pages 19, 50

[42] K. Zhang, R. Chen, and H. 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, 2015. ISBN 978-1-4503-3205-7. → pages 2, 14, 19, 38