SAPIENZA
UNIVERSITÀ DI ROMA

Graph Analytics on Modern Massively Parallel Systems

Flavio Vella

Submitted to the Department of Computer Science
in partial fulfillment of the requirements for the Degree of
Doctor of Philosophy in **Computer Science**
at the **Sapienza University of Rome**

February 1, 2017

GRAPH ANALYTICS ON MODERN MASSIVELY PARALLEL SYSTEMS

Approved by:

Dr. Massimo Bernaschi, Advisor
Institute for Applied Computing
National Research Council of Italy

Prof. Irene Finocchi
Department of Computer Science
Sapienza, University of Rome

Prof. Paolo Bottoni
Department of Computer Science
Sapienza, University of Rome

Prof. Giuseppe F. Italiano
Department of Civil and Computer Engineering
"Tor Vergata", University of Rome

Prof. John Owens
Department of Electrical and Computer Engineering
University of California, Davis

Date Approved: January 1, 2017

To my daughter Sofia

Table of Contents

Dissertation Overview **ix**

1 Introduction to Parallel Graph Analytics **1**

Introduction **1**

1.1 Massively Parallel Systems for graph analytics 4

1.2 Challenges in parallel graphs analytics 7

1.2.1 Mapping Parallel Algorithms to Architectures 9

2 Preliminaries **11**

2.1 Modeling real-world networks 11

2.1.1 Graphs 11

2.1.2 Real-World and Synthetic Graphs 13

2.2 Graph partitioning on Distributed Systems 15

3 Graph Traversal and Reachability **19**

3.1 Background and Related Work 20

3.1.1 Parallel and Distributed Breadth-First Search 20

3.2 ST-CON on Multi-GPU Systems 24

3.3 Performance Metric for st-connectivity 27

3.4 Experimental Results 29

3.5 Summary and Discussion 35

4 Betweenness Centrality **37**

4.1 Background and notation 38

4.2 Betweenness Centrality on Unweighted Graphs 41

4.2.1 Related Work 41

4.2.2 Betweenness Centrality on Multi-GPU systems 43

4.2.3 Active-Edge Parallelism 44
4.2.4 Algorithm Description 45
4.2.5 Sub-clustering 50
4.3 Heuristics for Exact Betweenness Centrality 52
4.4 Experimental Result on Unweighted Graphs 61
4.4.1 Evaluation Platforms and Data Sets 62
4.4.2 Single-GPU 62
4.4.3 Multi-GPU and Sub-Clustering 64
4.4.4 Heuristics 69
4.5 Betweenness Centrality on Weighted Graphs 75
4.5.1 Notation 77
4.5.2 Related Work 78
4.5.3 Maximal Frontier Algorithm 78
4.5.4 Communication Complexity 86
4.5.5 Parallel Sparse Matrix Multiplication 88
4.5.6 Implementation 95
4.5.7 Experimental Result 99
4.6 Summary and Discussion 105

5 Clustering Coefficient 107
5.1 Background 108
5.1.1 Local Coefficient Cluster 108
5.1.2 Caching and MPI-3 One-Sided in a nutshell 109
5.1.3 Caching RMA 112
5.2 Related Work 114
5.3 CLaMPI: a caching layer for MPI One-sided 115
5.3.1 Caching-Enabled Windows 116
5.3.2 Processing Gets 118
5.3.3 Data Structures 120
5.3.4 Eviction Procedure 124
5.3.5 Parameter Tuning 126
5.4 Scalable LCC Algorithm 128
5.5 Experimental Results 129
5.5.1 Micro-Benchmarks 130
5.5.2 LCC 136
5.6 Summary and Discussion 142

6 Conclusions 145
6.1 My Publications 147

Summary 149

List of Tables 151

List of Figures 155

References 182

Dissertation overview

Graphs provide a very flexible abstraction for understanding and modeling complex systems in many fields such as physics, biology, neuroscience, engineering, and social science. Only in the last two decades, with the advent of Big Data era, supercomputers equipped by accelerators –i.e., Graphics Processing Unit (GPUs)–, advanced networking, and highly parallel file systems have been used to analyze graph properties such as reachability, diameter, connected components, centrality, and clustering coefficient. Today graphs of interest may be composed by millions, sometimes billions, of nodes and edges and exhibit a highly irregular structure. As a consequence, the design of efficient and scalable graph algorithms is an extraordinary challenge due to irregular communication and memory access patterns, high synchronization costs, and lack of data locality. In the present dissertation, we start off with a brief and gentle introduction for the reader to graph analytics and massively parallel systems. In particular, we present the intersection between graph analytics and parallel architectures in the current state-of-the-art and discuss about the challenges encountered when solving such problems on large-scale graphs on these architectures (Chapter 1). In Chapter 2, some preliminary definitions and graph-theoretical notions are provided together with a description of the synthetic graphs used in the literature to model real-world networks. In Chapters 3-5, we present and tackle three different relevant problems in graph analysis: reachability (Chapter 3), Betweenness Centrality (Chapter 4), and clustering coefficient (Chapter 5). In detail, Chapter 3 tackles reachability problems by providing two scalable algorithms and im-

plementations which efficiently solve st-connectivity problems on very large-scale graphs. Chapter 4 considers the problem of identifying most relevant nodes in a network which plays a crucial role in several applications, including transportation and communication networks, social network analysis, and biological networks. In particular, we focus on a well-known centrality metrics, namely Betweenness Centrality (BC), and present two different distributed algorithms for the BC computation on unweighted and weighted graphs. For unweighted graphs, we present a new communication-efficient algorithm based on the combination of bi-dimensional (2D) decomposition and multi-level parallelism. Furthermore, new algorithms which exploit the underlying graph topology to reduce the time and space usage of betweenness centrality computations are described as well. Concerning weighted graphs, we provide a scalable algorithm based on an algebraic formulation of the problem. Finally, thorough comprehensive experimental results on synthetic and real-world large-scale graphs, we show that the proposed techniques are effective in practice and achieve significant speedups against state-of-the-art solutions. Chapter 5 considers clustering coefficients problem. Similarly to Betweenness Centrality, it is a fundamental tool in network analysis, as it specifically measures how nodes tend to cluster together in a network. In the chapter, we first extend caching techniques to Remote Memory Access (RMA) operations on distributed-memory system. The caching layer is mainly designed to avoid inter-node communications in order to achieve similar benefits for irregular applications as communication-avoiding algorithms. We also show how cached RMA is able to improve the performance of a new distributed asynchronous algorithm for the computation of local clustering coefficients. Finally, Chapter 6 contains a brief summary of the key contributions described in the dissertation and presents potential future directions of the work.

Chapter 1

Introduction to Parallel Graph Analytics

In recent years, there has been an ever-increasing interest and research activity in the study of real-world networks [New10, BE05] involving several disciplines including biology [CWZ09], computer science, economy, electrical engineering and sociology [Was94], just to cite few. Networks easily models complex systems generated directly or indirectly by human activity and interaction. For example, in social networks vertices may represent people, or sometimes communities, and edges some form of social interaction between them, such as friendship. Another example is the structure of the Internet at level of autonomous systems; vertices in this case represent autonomous systems and edges show the routes taken by data moving between them (Figure 1.1). An introduction to networks and a complete list of its own applications are reported by Newman [New10] and Brandes and Erlebach [BE05].

In terms of mathematical structures, networks can be modeled as *graphs*. Indeed, a strong relationship between graph theory and network analysis exists. Historically, the first notable example in network analysis is represented by "The Seven Bridges of Koenigsberg (1736):

> *Koenigsberg is divided into four parts by the river Pregel, and connected by seven bridges. Is it possible to tour Koenigsberg*

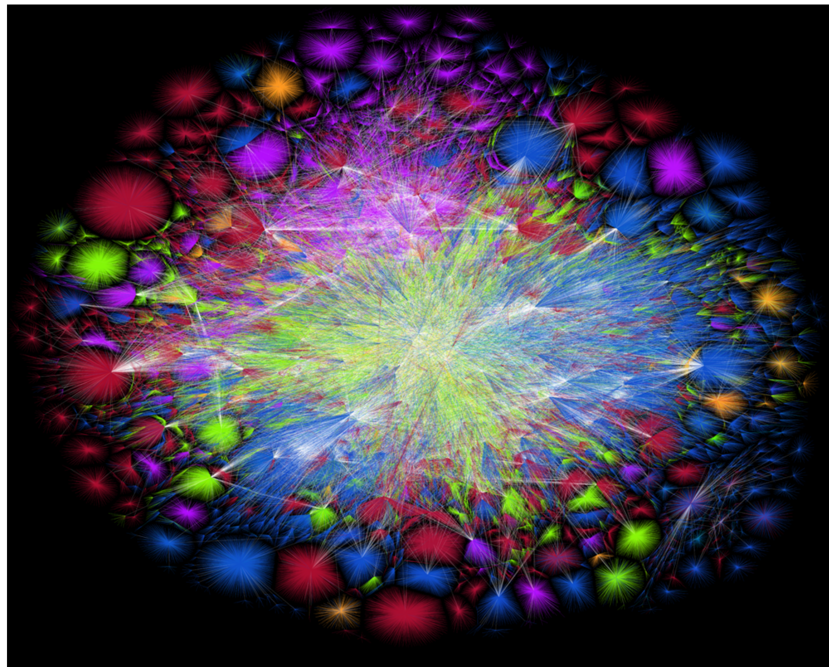


Figure 1.1: Visualization of the routing paths of the Internet. Figure made by the Opte Project (www.opte.org). CC Attribution-Non-commercial 4.0 International License [opt].

> *along a path that crosses every bridge once, and at most once? You can start and finish wherever you want, not necessarily in the same place.*

Euler proved that the problem has no solution, laying the foundations of graph theory [New03].

Today network analysis, involves searches and path analysis, connectivity analysis, community analysis and centrality analysis of graphs composed of billions of vertices and edges (*large-scale* graph):

Searches an structure analysis. This analysis requires the visit and the exploration of the graph.

Path analysis. This type of analysis can be used to determine the shortest distance between two or more nodes in a graph, for example. A well-known use case is route optimization that is applicable to logistics, supply and distribution chains and traffic optimization for smart cities.

Connectivity analysis. It asks for the minimum number of nodes (or edges) that needs to be removed to disconnect the remaining nodes from each other. As a use case, connectivity analysis allows determining weaknesses in networks such as a utility power grid or weak spots in communication networks.

Community analysis. Distance and density-based analysis are used to find groups of interacting people in a social network, for example, identify whether they are transient, and predict if the network will grow.

Centrality analysis. This analysis type enables to identify relevancy to find the most influential people in a social network, for example.

Some problems of network analysis and related algorithms used to solve them are reported in Table 1.1.

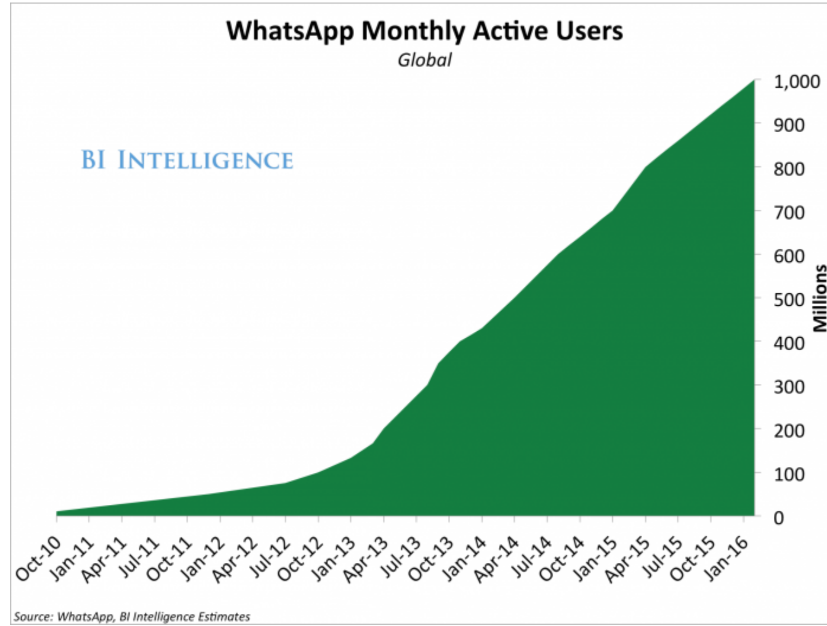


Figure 1.2: Whasapp monthly active users [biw].

Type of analysis	Problem	Graph algorithm	Time Complexity	Space Complexity
Search	Graph Traversal	Breadth First Search (BFS) [CLR90]	$\mathcal{O}(n+m)$	$\Theta(n+m)$
Path	Single-Source Shortest Path (SSSP)	Bellman-Ford [CLR90]	$\Theta(nm)$	$\Theta(n)$
Path	Single-Target connectivity (ST-CON)	s-t connectivity (ST-CON) [BR91, Rei05]	$\mathcal{O}(n+m)$	$\Theta(log_2(n))$
Community	Clustering Coefficient (CC)	Local Clustering Coefficient (LCC) [GMB14]	$\mathcal{O}(n\hat{\rho}^2)$	$\Theta(n)$
Centrality	Betweenness Centrality	Brandes'algorithm [Bra01]	$\mathcal{O}(nm)$	$\mathcal{O}(n+m)$

Table 1.1: Examples of algorithms used in graph analytics.

1.1 Massively Parallel Systems for graph analytics

The computational requirements of large-scale graph processing demand powerful and efficient high-performance computing systems. A variety of parallel systems with different capabilities are commercially available. Traditional systems are built out of microprocessors, with several layers of hierarchical memory. Recently, in addition to traditional multi-core processors, accel-

erators including Graph Processing Unit (GPU), Field Programmable Gate Array (FPGA), or co-processors (i.e., Xeon Phy [xeo]) provide extra computing resources to modern supercomputers and single workstations [Cha10,top]. While arithmetic intensity applications[a] represent the classic supercomputing workload, problems in discrete mathematics like graph analysis require fast memory access. These applications tend to have high latency and low utilization of floating point units on traditional microprocessors [MK07]. Massively multi-threaded architectures naturally fit well for graph algorithms for a variety of reasons. First of all, they provide a global address space reducing the complexity and the latency of data access unlike distributed memory systems. As a result, it is easy to implement both coarse and fine-grained parallelism. Furthermore, the data-dependent degree of parallelism exhibited by graph algorithms can be dynamically tuned, even if it is difficult to predict. However, the principal additional issue in parallel graph algorithms is that the number of threads can be much larger than the number of processors, therefore memory contention issues are more significant. Architectures such as Cray XMT have demonstrated impressive performance on executing graph algorithms [BM06]. Unfortunately, such architectures are prohibitively costly. Graphics Processing Units (GPU) have become popular as general computing device due to their massively parallel architectures in accelerating many regular applications [NBGS08]. Recently GPUs have been also used in accelerating low-arithmetic intensity applications such as graph algorithms [BNP12, HN07] mainly due to their significant theoretical memory bandwidth [HKOO11]. They leverage a single-instruction, multiple-thread (SIMT) programming model where consecutive threads execute the same instruction on different elements of data. However, thread workload imbalance, branch divergence, uncoalesced memory accesses may limit performances of GPU-accelerated graph algorithms. In some case, also a careful mapping of the algorithm into GPU architecture does not allow to achieve

[a] Arithmetic intensity is the ratio of floating point and/or integer operations to memory accesses.

good performances.

The major limitation of shared-memory systems is represented by memory capacity. Many large-scale graph instances may not fit into the memory that single machine provides; in this cases one solution is to partition the graph among several machines as we will argue in Section 2.2. Briefly speaking, distributed-memory systems consist of a set of processors and memory interconnected by high-speed network like Infiband [Pfi01]. Such systems are most commonly programmed by explicit message passing where the user is responsible for dividing the data among the memories of the different processors and for determining which processor performs which task. Typically, data are exchanged between processors by means of the MPI communication library [GLS99]. The application workflow, in particular the computation and communication, is almost completely user-defined. A typical MPI-based application follows Bulk Synchronous Parallel (BSP) model, in which processors alternate between working independently on local data, and participating in collective communication operations [Val90, KGGK94, JáJ92]. Lenharth *et al.* [LNP16] provided an example of a graph algorithm execution under BSP-style semantics. With a careful workflow analysis of the application the overall latency cost of collective operations can be substantially reduced, for example by accumulating data exchanges into larger buffers. Although in principle message passing programs need not be bulk synchronous, most implementations of MPI are optimized for *two-sided* communication in which sends and receives are coupled. However asynchronous messages can be interleaved with computation in an arbitrary way also thanks to the recent emerging support for *one-sided* communication primitives which allow to design new parallel and asynchronous graph algorithms [GHTL14]. The new generation of distributed systems additionally integrates accelerators increasing memory hierarchies, data access/movement complexity and, as a result, programming effort as well. For example, in Multi-GPUs systems traditionally the communication also involves bus interconnection as well as the host memory. Recently, new cutting-edge technologies try to reduce commu-

nication overhead among accelerators by exploiting Remote Direct Memory Access capability (RDMA) [PHV+13]. However, this technology does not seem yet to be mature enough for high performance graph analytics.

1.2 Challenges in parallel graphs analytics

One of the great challenges in parallel graphs analytics is to process very large graphs efficiently. Indeed, graphs may exhibit a vast series of different issues:

Problem size. As mentioned before, in the Big Data era graphs are composed by billions of nodes and edges; for example, Web graph has more than 50 billion nodes, representing web pages, and one trillion edges, representing hyper links between pages [KRR+00]. The Facebook graph has approximately one billion nodes, representing Facebook users, and 200 billion edges, representing friendships between users [UKBM11]. Most of graph operations, also those with linear or near linear time complexity, are extremely hard to solve in practice.

Different structures. Graphs coming from different domains can have different structural properties e.g., in road networks the number of edges connected to a vertex (degree) is fairly uniform across all vertices. Furthermore such graphs also have good separators [Pot97] which allow a better partitioning of the graph in terms of balancing among computing nodes. On the contrary, studies have shown that social network graphs like the Web or Facebook have a very different structure; although the average degree of a vertex is "relatively" small, the average diameter grows only as the logarithm of the number of vertex. These graphs do not expose good separators [FR07, Lu01]. Therefore, finding general strategies which work efficiently over all types of graph is challenging.

Irregularity. The data in graph problems are typically unstructured and highly irregular. The irregular structure of graph data makes difficult to

extract parallelism and exhibits poor irregular access memory patterns. As for scalability, this issue causes unbalancing among computing nodes resulting from poorly partitioned data.

Data-driven. Graph computations are often data-driven [LNP16, NBP13]. The computations performed by a graph algorithm are guided by vertices and edges of the graph on which it is working. For example, in a traversal-based algorithm a subset of vertices is involved at each step of the computation. The degree of parallelism based on workload distribution of computation changes during the execution. As a consequence, parallelism can be difficult to express because the structure of computations is not known a priori [MAB+10]. A parallel top-down BFS based on vertex parallelism is an example.

Poor locality. Strictly related to irregular and unstructured properties, computations and data access patterns tend not to have very much locality. Performance in modern parallel processors is predicated upon exploiting locality. Thus, significant performance can be hard to obtain for graph algorithms even on serial processors [LGHB07]. Distribution over many computing nodes exacerbates the locality issue and, as a consequence, increases the communication.

Low computation to communication ratio. Graph algorithms are often based on exploring the structure of a graph rather than performing large numbers of arithmetic operations. As a result, there is a higher ratio of data access to computation than to scientific computing applications, so many graph algorithms are memory-bounded. As mentioned before, these accesses tend to have a low amount of exploitable locality therefore the runtime can be dominated by the wait for memory loads.

1.2.1 Mapping Parallel Algorithms to Architectures

Modern Multi-core processors promise a dramatic increase in performance. However they also bring an unprecedented level of complexity in algorithmic design and software development. There are several aspects (in addition to those so far described) that we need to take into account when designing parallel graph algorithms. The emerging issues require careful solutions based on the specific system (and capabilities provided) which we consider. We have identified four main aspects to be considered in parallel algorithms design:

Granularity of parallelism. One of the fundamental point in designing graph algorithm is how to parallelize them. Some algorithms exhibit coarse-grained parallelism. For example, some centrality computations such as betweenness centrality [Bra01] require solving many shortest path problems. Each shortest path problem from distinct sources could be a separate and independent task. However, most of graph algorithms, particularly those with linear or near liner time complexity, exhibit a more fine-grained level of parallelism.

Memory contention. In shared-memory systems or, more in general, systems which support a unified address space, multiple threads may try to simultaneously access the same memory address. This issue, also called memory contention, can significantly reduce performance. The importance of this problem grows with increasing degrees of parallelism in particular on massively multi-threaded systems such as Graphics Processing Unit (GPU). Several solutions allow to overcome this issue: the simplest consists of the usage of atomic operations. Notice that the performance of atomic operations are strictly related to hardware capability as we will argue in Chapter 3.

Load balancing. The vertices being visited in a graph algorithm may have some spatial locality in global memory. This means that some processors/threads will have more work to do than others. Methods for

reassigning this work may improve performance. Although this issue is less pronounced on shared-memory machines because work can be migrated without explicitly moving data, finding balanced data-thread mapping is challenging since it also depends on the specific architecture. Temporal aspects of load balance can also be important. A single graph algorithm might have a time-varying amount of work to do. For example, in a synchronous BFS, early steps may have few vertices to visit, while significantly more in the next steps.

Framework and software design. Solutions to many of the challenges underlined should be encapsulated within software frameworks. In addition, extensibility and portability must be taken into account in order to design efficient and general frameworks. Within this context several frameworks for graph analytics have been developed both for distributed and shared memory systems employing different programming models. In the present thesis, we do not provide a comparison against general frameworks for graph analytics. Indeed, in the following chapters we describe solutions that are hardwired and customized for a specific cluster. Several surveys reported exhaustive comparisons between such frameworks [DV14, SSP+14].

Chapter 2

Preliminaries

2.1 Modeling real-world networks

Graphs provide a very flexible abstraction to model interactions between discrete entities in a variety of networks including social networks (friendship circles, organizational networks), the Internet (router topologies, the web-graph, peer-to-peer networks), transportation networks, electrical circuits, genealogical research and biology networks (protein-interaction networks, brain networks). In this section we provide the list of concepts and notations from graph theory and related algorithms necessary to understand the rest of the dissertation.

2.1.1 Graphs

Definition 2.1.1 *A graph G is a pair $G = (V, E)$ consisting of a set of entities (vertices) V and a set of binary relations (edges) E.*

Vertices and edges are also referred to as nodes and arcs, respectively.

Definition 2.1.2 *Two vertices $u, v \in V$ of a graph $G = (V, E)$ are called adjacent if there is an edge between u and v that is $(u, v) \in E$. For a graph $G = (V, E)$, the set of the vertices adjacent to v is called neighbors of v denoted by $\Gamma_G(v) = \{u : (u, v) \in E\}$.*

Definition 2.1.3 *A graph $G = (V, E)$ is undirected if for each unordered pairs (u, v), (u, v) and (v, u) represent the same relation with $u, v \in V$ and $u \neq v$ otherwise G (digraph) is called directed.*

Definition 2.1.4 *For a graph $G = (V, E)$, the degree of a vertex v, $\rho(v)$ is the number of the neighbors of v, that is $\rho(v) = |\Gamma_G(v)|$.*

For digraphs, we may distinguish between in-degree ($\rho^-(v)$) and out-degree ($\rho^+(v)$).

Definition 2.1.5 *A graph (digraph) $G = (V, E, w)$ is weighted where w is a weight function such that $w : E \to \mathbb{R}$*

Weighted graphs capture relationships of different cost, length, and strength. In what follows, we only consider edge-weighted graphs (as opposed to node-weighted graphs). If each edge weights is one, the graph is defined as unweighted.

Definition 2.1.6 *For a graph (digraph) $G = (V, E)$, a path P in G from a vertex v_0 to a vertex v_h in G is a sequence $\langle v_0, v_1, \dots, v_h \rangle$ of vertices such that $((v_0, v_1), (v_1, v_2), \dots, (v_{h-1}, v_h)) \in E$. The length l of a path in G is the sum of its edges weights $l = \sum_{i=0}^{i=h-1} w(v_i, v_{i+1})$ Two vertices s, t are reachable if there is a path between s and t in G. G is connected if every vertex is reachable from all other vertices.*

The connected components of a graph are the equivalence classes of vertices according to the relation "is reachable from".

Definition 2.1.7 *Let $\mathcal{P}_G(u, v)$ denote the set of paths from u to v in G. The distance d_G is the minimum length of the path from u to v:*

$$d_G(u, v) = \min_{\widetilde{P} \in \mathcal{P}_G(u,v)} l(\widetilde{P}) \tag{2.1.1}$$

Graph structure		
	G	A given graph, $G = (V, E)$; V and E are sets of vertices and edges. If G is weighted, then $G = (V, E, w)$ where $w : E \to \mathbb{R}$.
	n, m	Numbers of vertices and edges in G; $n = \|V\|, m = \|E\|$.
	$\rho(v)$	Degree of v; $\overline{\rho}$ and $\hat{\rho}$ are the average and maximum degree in G.
	$d(G)$	The diameter of a given graph G.
	A	The adjacency matrix of G.

Table 2.1: Summary of the symbols used in the dissertation.

$\widetilde{P}$ is called shortest path between u and v. Roughly speaking, the shortest path between two vertices u, v is a sequence of vertices such that the sum of the weights of the edges which form the path is minimized.

Definition 2.1.8 *The diameter $d(G)$ of a graph $G = (V, E)$ is the maximum distance between two vertices:*

$$d(G) = \max_{u,v \in V} d_G(u, v) \tag{2.1.2}$$

We denote the adjacency matrix of G as A; $A(i, j) = w(i, j)$ if $(i, j) \in E$, otherwise $A(i, j) = \infty$. In Table 2.1, the symbols used in the rest of the manuscript are reported.

2.1.2 Real-World and Synthetic Graphs

In general, networks appear in a large variety of contexts and often exhibit a surprisingly similar structure. As a matter of the fact, experimental studies [BA99,FFF99,New01,New03] have shown that they share common properties, two of the most important ones are heavy-tailed degree distributions and small diameters. We refer to the studies reported by Leskovec *et al.* [LCK+10] and Chakrabarti and Faloutsos [CF12] for detailed studies.

- The degree distribution of real-world networks follows a power law distribution. This means that the number of nodes N_ρ with degree ρ is

given by $N_\rho \propto \rho^{-\gamma}$ with $\gamma > 0$, where γ denotes the power law exponent.

- Most real-world graphs exhibit relatively small diameter (the small-world property [WS98]), or "six degrees of separation" [Mil67]).

Several models [CZF04, NWS02, WS98, LCK+10] have been proposed to generate synthetic graph instances with these characteristics. Here, we briefly describe the models of synthetic graphs generators used in the experimental sections in the following chapters.

Random graph models is the simplest and classical model for generating a random graph according to the Erdos-Renyi (ER) model [ER59]. Starting from a set of n vertices, we add an edge for every pair of vertices with probability p. ER generates graphs with a low diameter and a single large component depending on p. However, random graphs generated in this way have a Poisson degree distribution, whereas real-world networks often exhibit a power-law distribution. Furthermore, concerning community structure, ER graphs do not have cluster coefficients comparable to real-world networks.

R-MAT. The recursive matrix model (R-MAT) [CZF04], unlike ER model, generates graphs with $n = 2k$ vertices and m edges with the characteristics such as degree distribution and community structure similar to those that real-world networks exhibit. The procedure of graph generation can be summarized as follows: First an empty adjacency matrix is divided into four equal-sized partitions. The algorithm chooses one of the four partitions with probabilities a, b, c, d, respectively. The chosen partition is again sub-divided into four smaller partitions, and the procedure is repeated until a cell in the adjacency matrix is reached. At this point, an edge is created in the graph corresponding to this cell. This process is repeated until all m cells (edges) are assigned. In order to introduce perturbation in the degree distributions, we can

add noise to the a, b, c and d values at each stage of the recursion. This schema overcomes the main issue of ER model for communities generation. Intuitively the partitions a and d represent separate groups of vertices, with b and c being cross-links between these two groups. The recursive nature of the partitions ensures that the algorithm automatically generates sub-communities within existing communities. The R-MAT model can easily be extended to generate different graphs including undirected and bipartite graphs as well. From a practical point of view, implementations of R-MAT model generates synthetic graph by specifying the distribution of probability p (it also called *initiator parameters*), the *scale* (S) and the *edge-factor* (EF) of the graphs. The last two parameters allow to generate a graph with 2^S vertices and $2^S \times EF$ edges. As a matter of the fact the edge-factor represents the average degree of the vertices. As for distribution probability p we set-up a,b,c and d parameters equal to 0.57, 0.19, 0.19, 0.05, respectively. Such configuration follows Graph500 benchmark [graa]. Notice that R-MAT generator creates a small number of redundant edges between two vertices as well as self-loops. Multiple edges, self-loops, and isolated vertices can be removed in the preprocessing according to the specific problem.

2.2 Graph partitioning on Distributed Systems

Scientific computing applications extensively use graph partitioning tool to ensure load balance and minimize communication. With the advent of ever larger instances in applications such as scientific simulation or networks analytics, graph partitioning has become more and more a relevant problem. Therefore cutting a graph into smaller pieces is one of the fundamental component for designing efficient and scalable algorithms as well. As a matter of the facts, a suitable decomposition of a graph allows achieving better

results and satisfactory scalability on distributed systems. Examples of distributed algorithms in graph processing are BFS [YCH+05a], Triangle Enumeration [CC11], PageRank and Connected Components [SW13] and, more recently, Betweenness Centrality [BCV15]. Formally, we define the graph partitioning problem as follow:

Definition 2.2.1 *Given a positive integer* k*, let* $G = (V, E)$ *be an undirected graph. Let* $f : E \rightarrow \mathbb{R}_+$ *be an objective function which represents a metric* i.e., the communication volume *or merely the edge weights of the graph. The graph partitioning problem asks to find disjoint sets of vertices* $V_1, \cdots, V_k$ *such that* $\forall i \in 1, \cdots k$

- *the cost of all the edge weights in each* V_i *is distributed evenly (load balancing condition)*
- *the cost of all edge weights among the partitions* V_i *is minimized.*

In practice, we aim at finding a partition that minimizes (or maximizes) an objective function. The decision problem of partitioning a graph into k disjoint sets of roughly equal size such that the cut metric is minimized is NP-complete [HR73, GJS76]. There are several works in literature on methods and algorithms that solve graph partitioning problems such as Kerninghan-Lin (KL) algorithm [KL70], Fiduccia-Mattheyses (FM) algorithm [FM82], Spectral Bisection [BS94] and k-way partitioning method [KK98] and several tools (i.e., Metis [KK98]). An exhaustive overview can be found in [BMS+13].

In the present dissertation, we re-call two well-known strategies based on the static distribution of the vertices (1-D partitioning) or edges (2-D partitioning) among the processes. These solutions do not require specific information about the graph (i.e, *vertex or edge-separator* [PSL90]). The process owner a given vertex (or edges) can be easily derived by exploiting the index of the vertex in the adjacency metrix of the graph and the processor index in the mesh. As a consequence, we do not store additional information concerning the vertex-process mapping. More in detail, the 1-D partitioning is the simplest way of distributing the vertices of a graph [BCM+15] in a

distributed system. Assume there are $|P|$ processes $P = (p_1, \ldots, p_{|P|})$ in a distributed system. Let b be the the number of processes that may run on the same computing node cn_j with $1 \leq j \leq \left\lfloor \frac{|P|}{b} \right\rfloor$. P_j denoting the partition of P assigned to the computing node cn_j. V is divided into P subsets V_i and every $V_i \subseteq V$ is assigned to a different process p_i.

For example, a simple approach assigns each vertex u_k of the graph (together with its neighbors) to the process p_i according to the rule $i = k\%p$, where % indicates the remainder of the integer division k/p. However, for graph traversal based algorithms, 1-D partitioning suffers from poor scalability since it requires communication among all the $|P|$ computing nodes [YCH+05b, BM11, BCM+15]. In order to reduce the communication cost several works presented 2-D partitioning schema [YCH+05b, BM11]. 2-D partitioning assumes that the processors are arranged as a bi-dimensional mesh having R rows and C columns. The mesh is mapped onto the adjacency matrix $A_{N\times N}$ once horizontally and C times vertically thus dividing the columns in C blocks and the rows in RC blocks. Processor p_{ij} handles all the edges in the blocks $(mR + i,\ j)$, with $m = 0, ..., C - 1$. Vertices are divided into RC blocks and processor p_{ij} handles the block $jR + i$. 2-D partitioning properties can be summarized as follows: *i)* the edge lists of the vertices handled by each processor are partitioned among the processors in the same grid column; *ii)* for each edge, the processor in charge of the destination vertex is in the same grid row. The traversal steps can be grouped in two major phases: 1) build the current frontier of vertices on each processor belonging to the same column of the mesh (*expansion*); 2) exchange new discovered vertices involving the processor on the same row (*folding*). The 1-D partitioning requires $\mathcal{O}(|P|)$ data transfers at each step, whereas the 2-D partitioning requires only $\mathcal{O}(\sqrt{|P|})$ communications since only the processors in one mesh-dimension are involved in the communication at the same time.

Chapter 3

Graph Traversal and Reachability

Determining if two nodes are reachable from each other is one of the most fundamental problem in network analysis. This query requires efficient algorithms and strategies for a quick response on large-scale graph. *St*-connectivity (ST-CON) is the decision problem to determine whether or not two vertices s and t are connected by a path in G. ST-CON is also a basic building block for more complex connectivity and path problems [BN16, FJF15].

Breadth-First Search (BFS) [CLR90] is one of the basic for the design of efficient graph algorithms, and is also representative of a broader class of memory-intensive combinatorial problems on unweighted graph such as minimum-cut problem [CLR90], betweenness centrality [Bra01] and *st*-connectivity [BM06, BCMV15] as well. We can also find BFS as a subroutine for the GP problem [ASA16], described in Section 2.2 or commercial graph databases [FDB+14]. More recently, BFS is used as benchmark since it is considered a representative kernel for evaluating the performance of novel architectures on irregular applications [graa, gre, ABC+06]. BFS naturally solves ST-CON in linear time complexity [BR91] and log space complexity [Rei05]. The laziest approach would be to start a BFS from s, and stop when t is visited if some path form s to t exists. A more sophisticated solution consists of starting

two concurrent search from s and t as we will argue later. Such approach well fits on parallel architecture, nevertheless it poses new issues that need to be addressed.

The key contributions in present chapter are as follows:

- two distributed implementation of the bi-directional search algorithm on Multi-GPU systems;
- an efficient solution for the st-connectivity problem on large-scale unweighted graphs;
- a performance metric for the evaluation of the solution proposed;
- our solution achieves up to 25 of speed-up over the best existing approach;

3.1 Background and Related Work

3.1.1 Parallel and Distributed Breadth-First Search

Formally, given the source vertex s, BFS algorithm systematically expands the edges of G to discover every vertex that is reachable from s. During the traversal step, all vertices at a level k are first visited, before discovering any vertices at level $k+1$. Therefore a typical BFS output consists of a BFS-tree rooted in s, where the vertices at level k represent the predecessors of the vertices at level $k+1$. Optionally, BFS is also used to compute the distance from s to each reachable vertex. The BFS frontier is defined as the set of vertices in the current level. A first-in first-out (FIFO) queue-based sequential algorithm for BFS takes $\mathcal{O}(n+m)$ time. The fastest known algorithm for parallel BFS represents the graph as an incidence matrix, and involves repeatedly squaring this matrix, where the element-wise operations are in the min-plus semiring [GM88]. This solution computes the BFS ordering of the vertices in $\mathcal{O}(log_2(n))$ time in the EREW-PRAM model, over $\mathcal{O}(n^3)$ processors. This

algorithm is not suitable for traversing sparse large-scale graphs. Prior work on large-scale BFS implementations are mainly based on the extensions of two parallel BFS algorithms. In the first approach, vertices are visited level by level (*top-down* BFS), and the graph (vertex or edges) is partitioned (either implicitly or explicitly) among the processors. The running time increases linearly with the number of traversed levels. Notice that concerning the correctness, a given vertex v can have more predecessors therefore the algorithm can return different (valid) BFS-tree as output (*idempotent property*) according to the architecture consistency model implemented. This no-deterministic behavior can be avoided by a careful management of the memory contention. In the second one, refereed as *bottom-up* each unvisited vertex tries finding any parent among its neighbors, unlike top-down approach where the vertices in the current frontier looks for their neighbors. The advantage is that once a vertex has found a parent, it does not need to check the rest of its neighbors. As a result, this approach is very effective on short diameter graphs. Furthermore, the bottom-up algorithm also removes the need for some atomic operations in parallel implementations [BAP13]. A hybrid approach, called direction-optimizing, switches between the *top-down* and the *bottom-up* traversal. When the frontier is small, the top-down approach is faster than the bottom-up since it will struggle to find valid parents and thus do unnecessary work. On the other hand, when the frontier is large since the bottom-up results to be more efficient [BAP13, BBM+15]. Detailed discussions are provided in [DNM14, YBD14, HT13]. Finally, a different algorithm was proposed by Ullman and Yannakakis [UY91]. Instead of a level-synchronized search, the graph is explored using multiple path-limited searches, and these searches are finally merged together to obtain the properly BFS-tree from the source vertex. However, the implementation is more complicated than the simple level-synchronous approach, and thus there are not experimental evidences about performance.

A number of studies have focused on the BFS algorithm both on shared and distributed architectures. Agarwal *et al.* [APPB10] demonstrated poor

scaling due to atomic operations on multi-socket systems. To reduce this, the authors proposed a combination of the fine-grained approach (edge partitioning among the sockets) and the accumulation-based approach in edge traversal. In details, each socket atomically updates only the information of the local vertices into a bitmap. They achieve good scaling going with up to four sockets. On single GPU, several studies addressed the problem related to the data-thread mapping strategy. The easiest approach assign to each thread one element of the BFS queue. On power-law graphs, such approach suffers from thread unbalancing and poor performance [JLH+11, MB13, MB14a].

Hong *et al.* overcame the difficulties due to the vertex-parallelism by adopting a warp centric programming model [HKOO11, HOO11]. In their implementation each warp is responsible of a subset of the vertices in the BFS queue.

The approach proposed by Merrill *et al.* [MGG12] assigned a chunk of data to a CTA (a CUDA block). The CTA works in parallel to inspect the vertices in its chunk. Furthermore, they uses heuristics for avoiding redundant vertex discovery (*warp culling*). Similar approach was efficiently adopted by Gunrock for their direction-optimizing BFS implementation [WDP+16].

On distributed memory systems several works based on an algebraic approach have been proposed [BM11, CPW+12, US13, BBM+15]. Satish *et al.* [SKCD12] implemented a distributed BFS with 1-D partitioning that shows good scaling with up to 1024 nodes. They described a technique to postpone the exchange of predecessors at the end of the traversal step. Similar technique was also reported and validated in several works [BCMV15, BCM+15]. Ueno *et al.* [US13] presented a hybrid CPU-GPU implementation of the Graph500 benchmark, using the 2D partitioning proposed by Yoo *et al.* [YCH+05b]. Their implementation uses the technique introduced by Merrill *et al.* [MGG12] to create the edge frontier. Furthermore, in order to reduce the size of the messages, they used a novel compression technique. Finally, they also implemented a sophisticated method to overlap communi-

cation and computation in order to reduce the working memory size of the GPUs.

Recently, Edmonds *et al.* provided the first hybrid-parallel 1D BFS implementation that uses active messages [EWHL10].

Petrini *et al.* [CP14] implemented a distributed-memory parallelization of BFS for BlueGene/P architectures. Their results show that the combination of the underlaying architecture and the System Programming Interface (SPI) allows achieving significant performance on R-MAT graphs. They also reported that the SPI implementation outperforms the MPI one by a factor of 5. Bisson *et al.* [BBM16] developed a fast 2D BFS implementation which exploits Nvidia Kepler capabilities. Their code achieved 830 billion edges per second on an R-MAT graph by exploiting 4096 GPUs.

Bernaschi *et al.* [BCM+15] provided an efficient BFS implementation on Multi-GPU system based on:

- a modified CSR data structure which allow to use an efficient data-thread mapping strategies based on prefix-sum and binary search. This technique allows exploiting the idempotent property in order to avoid atomic operations;
- 1-D partitioning of the graph among GPUs;
- a new communication pattern for the predecessors exchanging, similarly to what proposed in [SKCD12].

Our *st*-connectivity implementation are based on such techniques. Formally, given an undirected graph G and two vertices s and t, the *st*-connectivity determines whether or not there is a path from s to t in G. If this path exist, it is also the shortest path between s and t. Often, in real cases, the corresponding search problem of finding a path from s to t is required as well. All the BFS implementations described before can be easily extended to solve *st*-connectivity. There are not many studies about bi-directional search for the solution of ST-CON on large-scale graphs.

The first ST-CON implementation based on concurrent bi-directional search on Cray MTA-2 architecture was provided by Bader *et al.* [BM06]. Their approach introduced a race condition when the frontiers of the both BFSs expend the same vertex. To solve this, they also provided a different strategies which perform one BFS at time. The algorithm first performs the BFS with the smaller current frontier. Besta *et al.* [BH15] proposed to use Atomic Active Messages (AAM) based on hardware transactional memory to solve the race condition on Intel Haswell and IBM Blue Gene/Q.

3.2 ST-CON on Multi-GPU Systems

We can determine if there is a path from vertex s to t by starting a BFS from s and terminating it when the target vertex t is reached. Otherwise the procedure will visit all the graph. By storing the predecessors the algorithm also solve the corresponding search problem.

As mentioned before, a different strategy would be to run two BFS concurrently, starting from both vertices s and t and expanding their frontiers [Poh69, BM06] The algorithm ends when the same vertex is in both frontiers. Vertices on both frontiers represent the ST-CON Matching Set (ST-CON MS).

Using the BFS algorithm in this way to solve the ST-CON problem poses the following main issues:

- in a BFS there is a single frontier, whereas in ST-CON there are two frontiers: one of the vertices from s and another of the vertices from t. This increases the memory requirements;

- in a BFS, a vertex can be either unvisited or visited, whereas in ST-CON, a vertex can be unvisited, visited from s, visited from t, or visited from both s and t (in which case, a matching vertex is found and the problem is solved);

- in a BFS, a vertex can have at most one predecessor, whereas in ST-CON, vertices within the matching set have two predecessors, one from the s path and another from the t path. As a consequence, the idempotent property is not valid anymore.

Algorithm 1 Parallel ST-CON with atomic operations

Input Input: graph G(V,E), s, t
Input Output: matching node v, its parents u and w, path s to t
Input Data: CQ and NQ; $pred$
Input Macro: $GetColor(u)$ get the color of u, $SetColor(c, u)$ set the color c to u

```
1:  CQ, NQ ← ∅
2:  pred[v_j] = −1, ∀ v_j ∈ V
3:  enqueue(CQ, SetColor(Red, s), SetColor(Blue, t))
4:  pred[s] = s; pred[t] = t
5:  while CQ ≠ ∅ do
6:      for each u_i in CQ in parallel do
7:          for each v_j neighbor of u_i (in parallel) do
8:              ——critical section——
9:                  if pred[v_j] == −1 then
10:                     pred[v_j] = u_i
11:                     MyColor = GetColor(u_i)
12:                     enqueue(NQ, SetColor(MyColor, v_j))
13:                 else if GetColor(pred[v_j])! = GetColor(u_i) then
14:                     return v_j, u_i, w_i
15:                 end if
16:             end ——critical section——
17:         end for
18:     end for
19: end while
```

In Algorithm 1 we present the pseudo-code for the ST-CON. At the beginning of the algorithm, both s and t are enqueued in the same queue, the first colored RED and the second BLUE, and their predecessors are set (lines 3-4). The frontier is then expanded and each new vertex, for which the predecessor is not set (line 9), is enqueued with the color of its parent (lines 11-12). If a vertex has already been visited from another color, then the algorithm ends, returning the matching vertex along with its parents (lines 13-14). The whole section between lines 8-16 is critical for concurrency. A straightforward way to maintain the coherence is based on the usage of atomic operations. More precisely, it can be implemented through the use of a compare and

swap operation that resolves the race condition among threads accessing the predecessor array (we refer to this implementation as *atomic-stcon*). For the simple BFS, this race condition is benign because the predecessors are *idempotent*.

Algorithm 2 Parallel ST-CON without atomic operations

```
Input Input: graph G(V,E), s, t
Input Output: Matching Node Set (MNS), each path from s to t
Input Data: CQ and NQ; pred_s, pred_t
Input Macro: GetColor(u) get the color of u, SetColor(c,u) set the color c to u, UnSetColor(u)
    return u discolored
 1: CQ, NQ, MNS ← ∅
 2: pred_s[v_j] = pred_t[v_j] = −1, ∀ v_j ∈ V
 3: enqueue(CQ, SetColor(Red, s), SetColor(Blue, t))
 4: pred_s[s] = s pred_t[t] = t
 5: while CQ ≠ ∅ & MNS == ∅ do
 6:     for each u_i in CQ in parallel do
 7:         for each v_j neighbor of u_i (in parallel) do
 8:             if GetColor(u_i) == RED then
 9:                 if pred_s[v_j] == −1 then
10:                     pred_s[v_j] = UnSetColor(u_i)
11:                     enqueue(NQ, SetColor(Red, v_j))
12:                 end if
13:             else if GetColor(u_i) == BLUE then
14:                 if pred_t[v_j] == −1 then
15:                     pred_t[v_j] = UnSetColor(u_i)
16:                     enqueue(NQ, SetColor(Blue, v_j))
17:                 end if
18:             end if
19:         end for
20:     end for
21:     for each v_i ∈ V in parallel do                 ◁ Find matching node
22:         if pred_s[v_i] ≠ −1 & pred_s[v_i] == pred_t[v_i] then
23:             enqueue(MNS, v_i)
24:         end if
25:     end for
26:     CQ ← NQ
27:     NQ ← ∅
28: end while
```

The need to control access to the critical section can be avoided by removing the race condition. To that purpose, it is necessary to use distinct memory locations to store the predecessors and visited vertices for the two subset of vertices with additional usage of the GPU memory. We refer to

this implementation as *no-atomic-stcon* (see Algorithm 2). More in detail, to keep track of vertices visited from s and t, we use different arrays. Predecessors and visited vertices from s are stored in $pred_s$ and $mask_s$, respectively, whereas predecessors and visited vertices from t are stored in $pred_t$ and $mask_t$, respectively. Filtering already-seen vertices depends on the vertex color (lines 8-17) and thus the critical section is removed. At the end of each BFS level (lines 21-25), we compare $pred_s$ and $pred_t$ to determine whether a matching vertex has been found. In this way, we also calculate all vertices in the matching set and all paths between s and t.

3.3 Performance Metric for st-connectivity

Most of recent work uses the TEPS metric to evaluate and compare BFS performance. Papers dealing with different graph algorithms such as Betweenness Centrality or All-Pairs Shortest-Path still use the running time [KKJ08, SKSc13, HN07, MBBC07].

There is, at least, one major drawback to using the TEPS metric to evaluate the performance of a solution to the ST-Connectivity problem and, more generally, for other graph algorithms (as far as we know TEPS has been used only for BFS). The problem is that it counts all the edges in the connected component that includes the starting vertex (root) in addition to those actually traversed by the algorithm. By doing this the TEPS metric does not account for the actual work done.

We argue that for ST-CON, a simple but effective metric can be represented by the mean value of the number of s-t paths (NSTPS) found in one second, averaged over a suitable set of extracted pairs:

$$< NSTPS >_{NE} = \frac{1}{< \text{s-t } time >_{NE}}$$

where NE is the Number of Extracted pairs. The number NE and set of s-t pairs must be selected carefully.

For the special case of R-MAT graphs, it turns out that it is possible

to choose at random a relatively small set compared to the total number of nodes in the graph ($NE << N$). This is a consequence of two properties of a R-MAT graphs: the power law distribution and small diameter. Such properties give rise to a sharp distribution of the shortest path lengths. For example, we computed the mean and variance of the shortest path lengths found by extracting 100, 1000 and 10000 s-t pairs for different instances of an R-MAT graph (mean and variance were computed only for connected pairs). The results are shown in Table 3.1.

R-MAT GRAPHS								
S=25, NP=16, d=8			S=27, NP=64, d=9			S=29, NP=256, d=9		
NE	Mean	Variance	NE	Mean	Variance	NE	Mean	Variance
9989	1.9994	0.0280	9946	2.110	0.1006	9983	2.022	0.0311
999	2.002	0.0240	995	2.110	0.1024	999	2.024	0.037
100	2.01	0.01	100	2.090	0.0827	100	2.01	0.01

Table 3.1: Mean and variance of the length of shortest paths for three different instances of an R-MAT graph with different values of SCALE S. NP is the number of GPUs and d the diameter of the graph. For each instance, NE is the number of extracted vertices.

REAL GRAPHS					
Live-journal1 S=22, EF 14, d=15			com-Orkut S=22, EF 38, d=8		
NE	Mean	Variance	NE	Mean	Variance
9988	3.027	0.298	10000	2.361	0.240
1000	3.021	0.302	1000	2.365	0.244
100	2.84	0.297	100	2.25	0.209

Table 3.2: As in Table 3.1, we report the mean and variance of the length of the shortest paths for two real graphs.

It is apparent that the mean and variance are reasonably stable with respect to the number of extractions. By invoking the Law of Large Numbers, we can state that 1000 pairs are a representative set of the whole graph. We

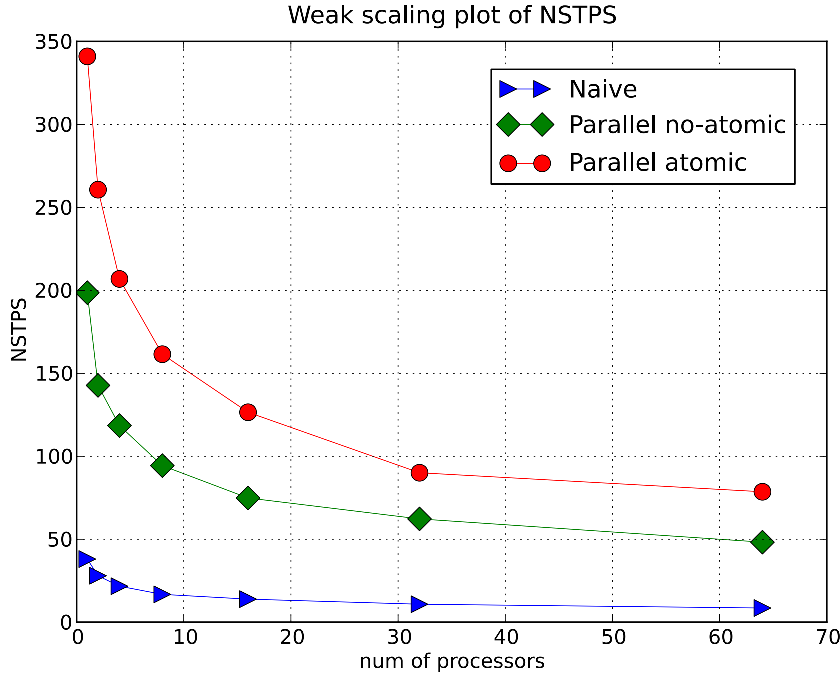


Figure 3.1: Weak scaling plot of the number of ST-CON problems solved within a second (*NSTPS*) for the two implementations described in the text: *atomic-stcon* and *no-atomic-stcon*. For comparison, we also show the performance of the naive (single BFS) implementation. The SCALE of the R-MAT graph ranges from $21 - 27$ for $1 - 64$ GPUs, respectively, with EF equal to 16.

can repeat the same argument for the real graphs that we analyzed. In those cases, the variance is higher and thus we decided to increase the number of pairs to 10000 (although 1000 would suffice, see Table 3.2).

3.4 Experimental Results

We report performance evaluation for both *atomic-stcon* and *no-atomic-stcon* solutions for the ST-Connectivity problem on a Multi-GPU.

Figure 3.1 shows the performance, in NSTPS, of our implementations. For

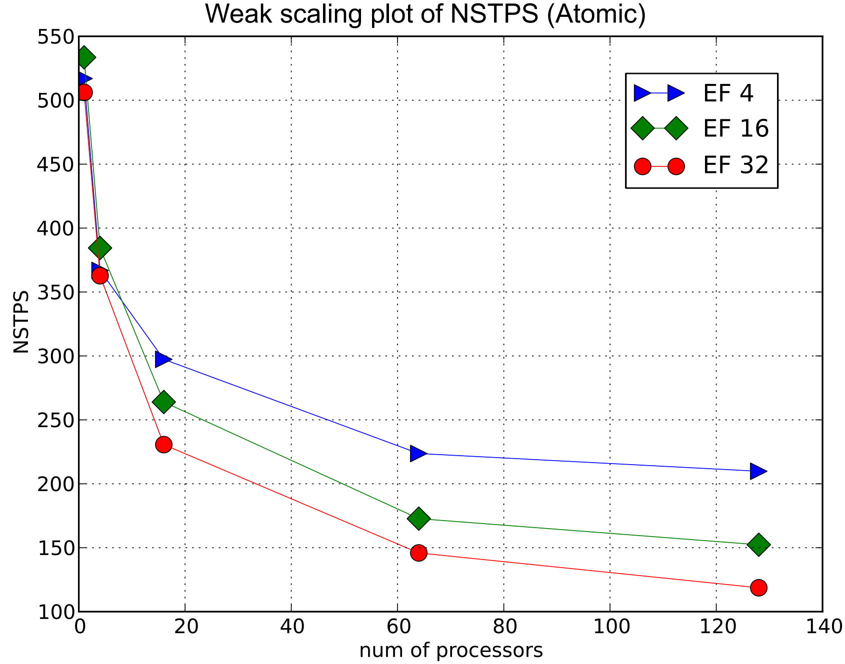


Figure 3.2: Weak scaling plot of *NSTPS* using the *atomic-stcon* implementation for three different values of EF. The SCALE of the R-MAT graph ranges from $19 - 26$ for $1 - 128$ GPUs, respectively.

comparison, the performance of a *naive* implementation is also plotted. The *naive* implementation simply starts a BFS from s and stops if t is reached. As expected, both the *atomic-stcon* and *no-atomic-stcon* outperform the *naive* implementation (Figure 3.1). The weak scaling plot is consistent with our basic intuition: the ST-Connectivity problem is harder when the scale of the graph is larger. Searching a path within a larger component using a BFS algorithm requires the traversal of more edges. This is also apparent in Figure 3.2, where by varying the EF parameter, we change the number of edges given a number of vertices. The code performs better when there are fewer edges to be traversed (the plot refers to the *atomic-stcon* implementation, but we obtained the same behavior for the *no-atomic-stcon* implementation).

It is apparent that the *atomic-stcon* implementation performs better than

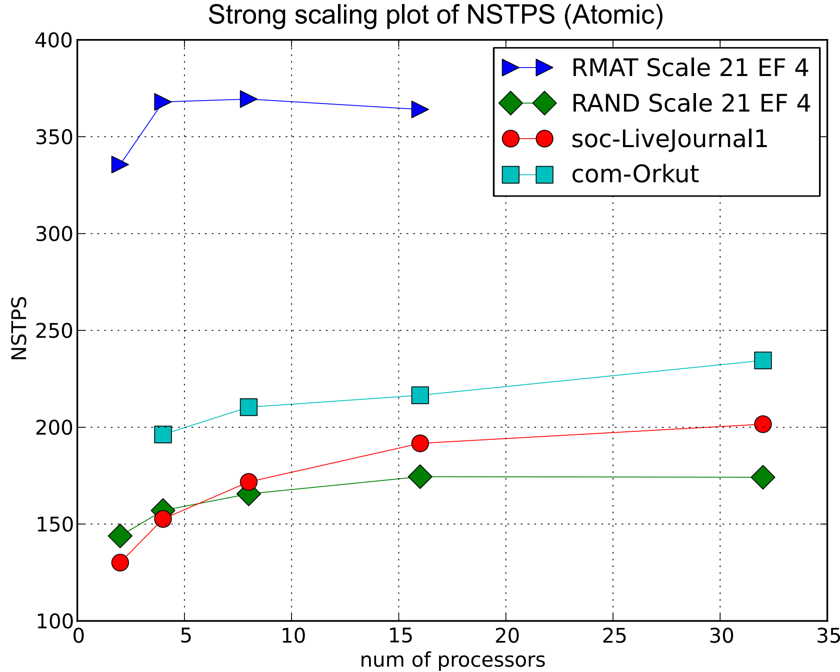


Figure 3.3: Strong scaling plot of *NSTPS* using the *atomic-stcon* implementation.

the *no-atomic-stcon*. This is mainly because the *atomic-stcon* implementation visits a very small fraction of the vertices in the graph, as shown in Table 3.3. Moreover, atomic primitives have been significantly improved in the latest Nvidia GPU "Kepler" [US13].

A strong scaling plot of the *atomic-stcon* implementation is shown in Figure 3.3. We gain some benefits only by using a small number of computing nodes. This is because there is not enough work to be done in parallel, as becomes apparent by looking at Table 3.3. For the graphs under investigation, the matching vertex (MV-lvl) is found after, at most, three levels. At that level, hub vertices are usually enqueued but not yet visited [MGG12, BAP12]. As a consequence, only a small fraction of vertices are actually visited (columns "V naive", "V atomic", and "V no-atomic"). This under-utilizes the CUDA threads and, in turn, explains the lack of

Data Set Name	V naive	V atomic	V no-atomic	SCALE	EF	MV-lvl
R-MAT	31.59%	<1%	1.94%	22	16	1.98
RANDOM	81.77%	<1%	1.24%	22	16	2.76
soc-LiveJournal1	68.27%	< 1%	6.05%	$\sim$ 22	$\sim$ 14	3.0
com-Orkut	74.52%	<1%	9.75%	$\sim$ 22	$\sim$ 38	2.36

Table 3.3: Columns 2, 3, and 4 show, for different graphs, the percentage of vertices in the graph visited by the *naive*, *atomic*, and *no-atomic* implementations, respectively. Columns 4 and 5 specify the size of each graph in terms of SCALE and EF (for real datasets, SCALE and EF are approximated from the number of vertices and edges). Column 6 shows at which BFS level (MV-lvl) the matching vertex is found. The level is computed as the average with respect to 10000 random instances of the ST-CON problem for the same graph.

scalability.

Our implementation can also output the path (or the set of paths) between s and t. This part has been implemented in a straightforward way by collecting all the predecessors on one computing node and then traversing the whole set backward from the matching vertex and its predecessors. We did not include the time needed for that task in the reported NSTPS.

To the best of our knowledge, one paper deals with a parallel implementation for the ST-Connectivity problem [BM06]. The authors exploited the idea of starting two BFS concurrently from both s and t. The proposed solution relied on atomic operations and achieved a performance of 0.3s to solve an ST-CON problem on a scale-free graph with 134 million vertices and 805 million edges (EF $\sim$ 6) on a Cray MTA-2. On a problem of comparable size, an R-MAT graph with 134 million vertices and 4 billion edges (EF=16), our *atomic-stcon* implementation running on 64 GPUs is about 25 times faster (it takes 0.012s).

Figure 3.4 shows a time breakdown of the main computational and communication components of the two implementations for a R-MAT graph, while in Figure 3.5, we compare the overall computation and communica-

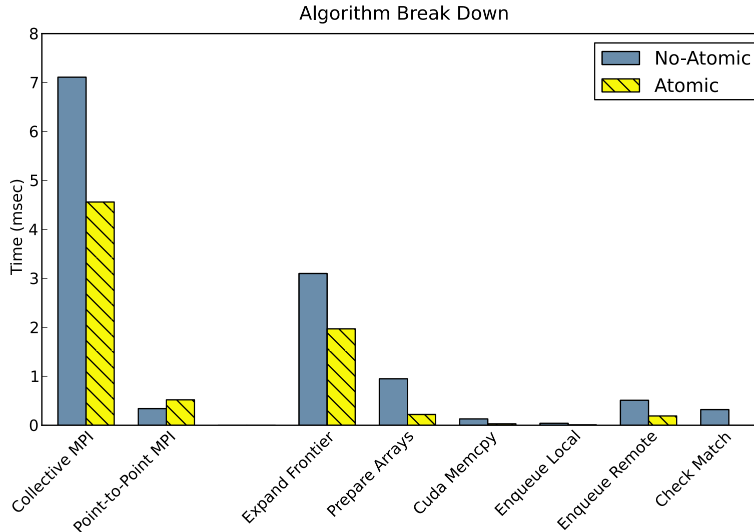


Figure 3.4: Time breakdown for *no-atomic-stcon* and *atomic-stcon* on 16 GPUs. The scale of the R-MAT graph is equal to 25, and EF is equal to 16.

tion time for three different graphs. Data were collected, using 16 GPUs to solve 2000 s-t pairs for an R-MAT graph and a RANDOM graph with SCALE equal to 25 and EF equal to 16, and using only 4 GPUs for the *com-Orkut* real dataset, since it is smaller than the synthetic graphs. Each timing is averaged over the 2000 runs and over the number of GPUs. We first discuss the computational parts: *expand frontier*, *prepare arrays*, *enqueue local*, *enqueue remote*, and *check match*. The expansion of the frontier, where the NLFS is built starting from the *Current Queue*, is the most time consuming part. The *atomic-stcon* implementation performs better because it stops exactly when the first matching vertex is found. Thus, on average, only a subset of the vertices in the frontier are actually visited. The *prepare array* part, which organizes data for communications, is more expensive in the *no-atomic-stcon* implementation, because of the redundancy of the data structure used. However, at this point of the BFS, there are fewer elements to be processed, because of the filtering in the previous step, and thus the

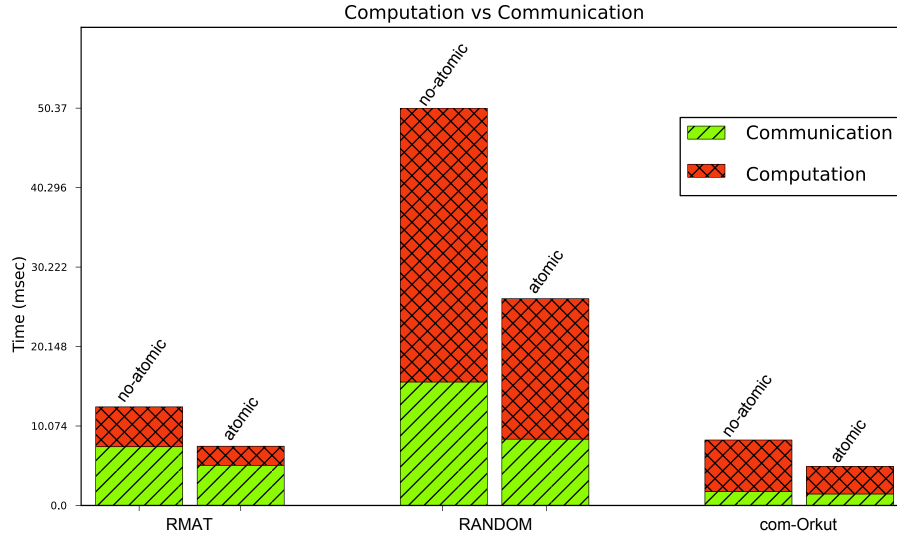


Figure 3.5: Computation and communication time breakdown for *no-atomic-stcon* and *atomic-stcon* for three graphs. The scale of the R-MAT and RANDOM graphs is equal to 25, and EF is equal to 16. The scale of the com-Orkut graph is approximately 22, and EF is approximately 38.

time required by *prepare array* is only a small fraction of the time required by *expand frontier*. Once again, as the number of visited vertices is, on average, smaller in the *atomic-stcon* implementation, it is less expensive to use atomic primitives to build the queue. The *check matching* is present only in *no-atomic-stcon*, where the check for a matching node is performed right before starting a new iteration, whereas in *atomic-stcon*, this check is performed in both *expand frontier* and *enqueue remote*. MPI communications can be further divided into collective and point-to-point primitives. The first group includes the `MPIallgather` and `MPIallreduce`, operations. Collective MPI operations require a significant amount of time because they are implicit synchronization points, that is, all processes wait for the slowest computing node. In the first BFS levels, computation may be unbalanced

among computing nodes because queues contain relatively few elements not uniformly distributed among them. This unbalance occurs both in the ST-Connectivity and BFS algorithms, but the former terminates after very few levels (see Table 3.3), and therefore the unbalance is more evident.

3.5 Summary and Discussion

ST-CON is one of the most fundamental query in network analysis. The traditional implementations of ST-CON are based on atomics or fine locks may entail significant overheads. We presented two solution for the ST-connectivity problem on large-scale graphs. Both implementations are based on a distributed bi-directional search on Multi-GPU system. However this approach introduces race condition on distributed and parallel implementations. Atomic primitives can be used for the control of critical sections where data structures, shared by the concurrent BFS, need to be accessed. The efficiency of the atomic primitives available using NVIDIA "Kepler" architecture are crucial under this points and they allow to solve about 340 ST-CON problems per second on a graph having about 2 million vertices and 32 million edges. Experiments on previous generation of GPUs show that atomic operations have significant impact on performance especially on Multi-GPU system. In this case, the solution based on data-structure replication provides better performance over the same instances.

Concerning the comparison against state-of-the-art, our solution is 25× faster than the best existing parallel implementation.

Chapter 4

Betweenness Centrality

One of the main goals of graph analysis is to rank the nodes in a network according to a centrality measure. In general, centrality measures play an important role in several graph applications including transport networks [WMWJ11], social and biological networks [Fre77, BS09] and and terrorism prevention as well [BKMM07]. One of the most popular metrics is the Betweenness Centrality (BC) [Fre77] and basically it requires the solution of the all shortest-paths in the graphs. The Brandes BC algorithm [PP13,Bra01] provides a work-efficient way to obtain centrality scores without needing to store all shortest-paths simultaneously, achieving a quintessential reduction in the memory footprint. It requires $\mathcal{O}(n + m)$ space and solves the exact BC in $\mathcal{O}(nm)$ and $\mathcal{O}(nm + n^2 log(n))$ time steps on unweighted and weighted graph respectively [Bra01], where n is the number of vertices and m is the number of edges. Therefore, the exact computation is infeasible for very large networks. To date, most parallelizations of BC have leveraged the breadth first search (BFS) primitive [DWM09, TTS09, TSG11, MEJ+09, GB13, MB14b, Sar15], which can be used to implement Brandes algorithm on unweighted graphs [KG11]. However, in those solutions, the size of the graph is limited by the space complexity of Brandes' algorithm [Bra01]. Pioneering approaches [BG11, BCV15] overcome the memory limitations by distributing the graph among more computational nodes. Inter-node com-

munication remains a major bottleneck since the times of early distributed implementations of BFS [BM13, YCH+05a, LGHB07]. Although advanced techniques overcome many of the difficulties (e.g., the use of a bit-mask improves the scalability [SKCD12]), a distributed BC implementation requires further/specific techniques due to Brandes' algorithm time-complexity and for the different memory requirements with respect to a simple BFS. For instance, a distributed BC implementation requires information exchange about the shortest paths so there is no benefit from using a bit-mask during the communication [BCV15]. In the present section, we provide several solutions to overcome and mitigate all these difficulties.

In detail, the key contribution of this chapter are as follows:

- A new distributed algorithm for the computation of Betweenness Centrality on unweighted graphs based on three different levels of parallelism.

- Two different heuristics to reduce the time and space requirements of BC computation.

- A new algebraic-based distributed algorithm for the solution of Betweenness Centrality on weighted graphs.

- A dynamic technique to perform efficiently the algraic operation between operands with different sparsity.

4.1 Background and notation

The first formal definition of the betweenness centrality metrics was proposed by Freeman and Linton [Fre77] (see also the Wasserman's work for further details [Was94]). Let σ_{st} be the number of shortest paths between vertices s, t, whereas $\sigma_{st}(v)$ represents the number of those shortest paths that pass through v with $s, t, v \in V$. We define the pair-dependency on v of a pair s,t,

the ratio $\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}}$. The betweenness centrality of a vertex v is defined as the sum of the pair-dependencies of all pairs on v,

$$BC(v) = \sum_{s \neq t \neq v} \delta_{st}(v) \tag{4.1.1}$$

Before Brandes' work, a simple algorithm computed the BC score by solving the all-pairs-shortest-path problem and then by counting the paths. That solution requires $\mathcal{O}(n^3)$ time by using the Floyd–Warshall algorithm and $\Theta(n^2)$ space for pair-dependencies. In order to remove the explicit summation of all pair-dependencies and thus exploiting the natural sparsity of real-world graphs, Brandes introduced the dependency of a vertex v with respect to a source vertex s:

$$\delta_s(v) = \sum_{w:v \in pred(w)} \frac{\sigma_{sv}}{\sigma_{sw}} \cdot (1 + \delta_s(w)) \tag{4.1.2}$$

Formula 4.1.1 can be re-defined as sum of dependencies:

$$BC(v) = \sum_{s \neq v} \delta_s(v) \tag{4.1.3}$$

To summarize, Brandes' algorithm (see Algorithm 1 in [Bra01]) computes BC scores in $\mathcal{O}(nm)$ on unweighted graphs and consists of:

1. computing the single source-shortest-path σ from a single root vertex s;

2. summing all dependencies δ from s and update BC score;

3. repeating steps 1) and 2) for each vertex in G.

Algorithm 3 Brandes' algorithm

```
Input G(V, E)                                         ◁ G is unweighted graph
Output BC[v], v ∈ V
 1: BC[v] ← 0
 2: for s ∈ V do
 3:     S ← empty stack
 4:     Pred[v] ← NULL ∀v ∈ V
 5:     σ[v] ← 0, ∀v ∈ V, σ[s] = 1
 6:     d[v] ← −1, ∀v ∈ V, d[s] = 0
 7:     Q ← empty queue
 8:     enqueue s → Q
 9:     while Q not empty do                          ◁ Path counting via BFS
10:         dequeue v ← Q
11:         push v → S
12:         for each neighbor w of v do
13:             if d[w] < 0 then
14:                 enqueue w → Q
15:                 d[w] ← d[v] + 1
16:             end if
17:             if d[w] = d[v] + 1 then
18:                 σ[w] ← σ[w] + σ[v]
19:                 append v → Pred[w]
20:             end if
21:         end for
22:     end while
23:     δ[v] ← 0, ∀v ∈ V                              ◁ Dependency
24:     while S not empty do
25:         pop w ← S
26:         for v ∈ Pred[w] do
27:             δ[v] ← δ[v] + σ[v]/σ[w] × (1 + δ[w])
28:         end for
29:         if w ≠ s then                             ◁ Update BC
30:             BC[w] ← BC[w] + δ[w]
31:         end if
32:     end while
33: end for
```

4.2 Betweenness Centrality on Unweighted Graphs

Several authors tackled the problem of speeding up the exact BC computation by parallelizing Brandes' algorithm for unweighted graphs. That approach requires a fast and memory-efficient traversal algorithm, like Breadth First Search. BC computation on GPU suffers from both the irregular access pattern and the workload unbalance due to traversal steps of the graph (counting of shortest paths and dependency accumulation).

4.2.1 Related Work

Jia et al. [JLH+11] evaluated two types of data-thread mapping: *vertex-parallel* and *edge-parallel*. Briefly, the former approach assigns a thread to each vertex during graph traversal. The number of edges traversed per thread depends on the degree of the vertex assigned to each thread. The difference in the degree among vertices causes a load imbalance among threads. In particular, since the degree distribution of typical scale-free networks (like the social networks) follows a power-law [BA99], there is a severe load imbalance that explains the poor performance obtained with that approach on GPU systems. The *edge-parallel* approach solves that problem by assigning edges to threads during the frontier expansion. However, this assignment of threads can also result in a waste of work because the edges that do not originate from vertices in the current frontier do not need to be inspected. The *edge-parallel* approach is not well-suited for graphs with low average degree, as well as dense graphs [JLH+11]. The vertex-based parallelism is affected by workload unbalance, whereas the edge-based parallelism uses more memory and more atomic operations [JLH+11, SKSc13]. Several authors proposed different strategies in order to exploit the advantages of both the methods [SKSc13,Sar15,MB14b]. In detail, McLaughlin and Bader discussed two hybrid methods for the selection of the parallelization strategy. Sarıyüce *et al.*, introduced the vertex virtualization technique based on a relabeling of the data structure (e.g., CSR, Compressed Sparse Row) [SKSc13] [Sar15].

Their solution is able to compute 32 concurrent BFS on an Nvidia Tesla K20 before decreasing in performance. The technique replaces a high-degree vertex v with $n_v = \lceil \texttt{adj(v)} \rceil / \Delta$ virtual vertices having at most Δ neighbours. Vertex virtualization technique is not very effective for graphs with low average degree. Moreover, it requires a careful tuning of its parameters. The authors also proposed a coarse-grained approach in which a single GPU executes multiple BFS at the same time with an increase of memory requirements. Madduri *et al.* [MEJ+09] propose to check successors instead of predecessors in the dependency accumulation step. In that way, the dependency accumulation procedure can start from one depth-level closer to the root vertex of the BFS tree and generally it does not require atomic operations. On distributed systems, betweenness computations exhibits both coarse- and fine-grained parallelism. In the coarse-grained parallelism, the entire graph and additional data structures are replicated so that each computing node has its own local copy. Since each root vertex can be processed independently, each computing node processes a subset of the vertices of the graph. At the end of the procedure, a *Reduce* operation is also required to update the final BC scores. For graphs that have a single connected component, the amount of work will be balanced among computational nodes. In this case, a nearly perfect scaling can be expected [MB14b]. However, this approach does not work in case of large scale graphs which cannot be stored in the memory of a single GPU. On the contrary, in the fine-grained approach all processing units are involved concurrently on the same computation starting from a single root vertex. On distributed systems, this requires a partitioning of the graph and data structures among the computational nodes. Edmonds *et al.* proposed a space efficient distributed algorithm where the vertices are randomly assigned to each processor [EHL10]. On unweighted synthetic graphs, the authors showed a satisfactory scalability up to 16 nodes. Gunrock library [PWW+15] also provides an implementation of Brandes' algorithm on a single Multi-GPU computing node. Their BC implementation is 2.5 faster than the single GPU version by exploiting 6 GPUs

and 1-D partitioning [WDP+15]. The 2-D partitioning is already adopted for improve the scalability of the betweenness algorithm [BG11]. Their solution solved the exact BC computation exploiting a Multi-Source BFS algorithm for the shortest path counting based on the linear algebra approach [KG11]. In particular, the Multi-Source BFS is implemented as the multiplication of the transpose of the adjacency matrix of the graph (M') with a rectangular matrix F, where each ith column of F represents the current frontier of the ith concurrent BFSs. However their solution did not exploit heuristics, thus the performance is limited. To the best of our knowledge, there are no solutions, based on a linear algebra formulation, which exploit heuristics to speed-up the BC computation. Bernaschi et al. [BCV15] proposed the first fully distributed BC on Multi-GPU systems. Their solution scales well up to 64 GPUs on the Friendster graph [LK15]. The authors also compared two different partitioning strategies.

4.2.2 Betweenness Centrality on Multi-GPU systems

Our Multi-GPU Betweenness Centrality (**MGBC**) algorithm consists in a sophisticated parallelization of Brandes' algorithm that exploits three different and complementary levels of parallelism. In particular,

1. at node-level: CUDA threads work on a subset of edges (fine-grained parallelism on shared memory system) according to a suitable strategy of data-threads mapping.

2. at cluster-level: a set of processors (or accelerators like GPUs) works concurrently following a graph partitioning strategy. At this level, the performance depends on the communication network as well.

3. at subcluster-level: multiple sub-clusters work over replicas of the same graph. Each sub-cluster performs the BC procedure on a subset of vertices concurrently to other sub-clusters. This further level of parallelism can be introduced since the betweenness equation is additive.

On a distributed-memory system, the last two levels enable fine and coarse-grained parallelism.

4.2.3 Active-Edge Parallelism

In this section we present the our data-thread mapping strategy extending the solution introduced by Bisson *et al.* [BBM16]. The goal is to provide a general data-thread mapping strategy that does not depend on specific characteristics (e.g., the degree distribution) of the graph. This strategy, we called Active-Edge parallelism, extends the edge-parallel approach by assigning a thread to each outgoing edge from the vertices in the current frontier (CQ). In this way, we do not need to inspect each edge in the graph as the original edge-parallelism strategy does. To this aim, it is necessary to count the total number of outgoing edges from the vertices in the frontier and then map each vertex to its neighbors. In detail, the degree of each vertex in the current frontier is stored into a contiguous array CD. A prefix-sum of the CD array is performed afterwards. At the end of the prefix-sum, CD contains the information required to identify the predecessor vertex associated with the ith thread. In order to determine the predecessor, a binary search over the CD array is also required. A simple example of this approach is illustrated in Figure 4.1. This mapping achieves a pretty good load balancing among threads by introducing extra computation. The prefix-sum and the binary search may represent a significant overhead during the traversal steps. However, it is possible to reduce that cost by noting that the dependency accumulation procedure visits the same frontiers of the first phase of Brandes in reverse order. With the active-edge parallelism strategy, in both steps the scan operation is performed over the degrees of the same vertices belonging to the same frontiers. By storing and accumulating the offset array CD during the shortest path counting, we can avoid to perform again the prefix scan. This solution allows reducing the computation time during dependency accumulation by reading CD stored in the previous step. By exploiting such symmetry the binary search results can be reused as well.

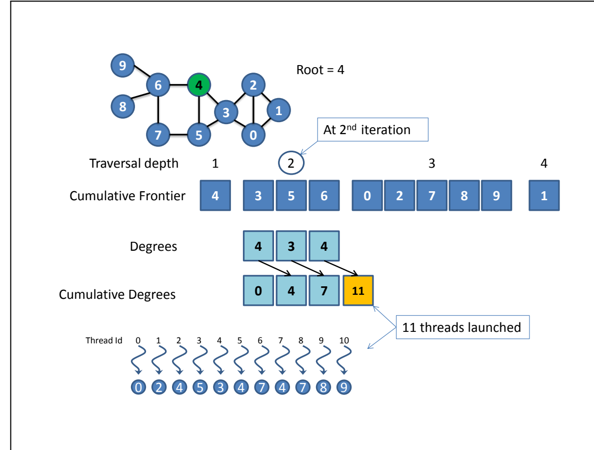


Figure 4.1: Example of Active-Edge Parallelism.

Obviously, this time-saving has an extra memory cost that is, at most, $\mathcal{O}(n)$.

4.2.4 Algorithm Description

Multi-GPU Betweenness Centrality algorithm (MGBC) is a novel parallelization of Brandes' algorithm. Like Brandes' algorithm, MGBC is composed of three main steps: *i*) shortest paths counting, *ii*) dependency accumulation and *iii*) update of BC scores. Algorithm 4 describes the shortest path counting procedure implemented in MGBC. At the beginning of each step, each processor has its own subset of the frontier. The processors on the same column of the mesh exchange frontier vertices (vertical communication). After this operation, all processors on the same column share the same frontier. During the frontier expansion (see Algorithm 5), new discovered vertices are marked as visited. Their σ values are updated by an atomic operation. The edges belonging to other processors are communicated together with the partial σ score (horizontal communication). At the end, the new frontier and σ values are updated.

After shortest path counting, the depth/level array of each discovered vertex (d) is exchanged as well. This operation is performed once for each BC

Algorithm 4 Shortest Path Counting on Multi-GPU

Input $G(V, E)$ ◁ G is unweighted graph
Input Processor P_{ij}
Input Root vertex s

1: $\sigma[v] \leftarrow 0, \forall v \in V$
2: $d[v] \leftarrow -1, \forall v \in V$
3: $Q \leftarrow$ `empty queue`
4: $lvl \leftarrow 0$ ◁ BFS level or depth
5: $nq \leftarrow 1$
6: $Q_{off}[0] \leftarrow 0$
7: **if** s belongs to P_{ij} **then**
8: $\quad \sigma[s] \leftarrow 1$
9: $\quad bmap[s] \leftarrow 1$
10: $\quad d[s] = 0$
11: $\quad$ `enqueue` $s \rightarrow Q$
12: **end if**
13: **while** true **do**
14: $\quad lvl \leftarrow lvl + 1$
15: $\quad$ `gather` Q and σ from column j ◁ Vertical communication
16: $\quad Q_{off}[lvl] \leftarrow Q_{off}[lvl - 1] + nq$
17: $\quad nq \leftarrow 0$
18: $\quad Q_r \leftarrow$ **expandFrt** $(lvl, bmap, Q, Q_{off}, d, \sigma)$
19: $\quad$ `exchange` Q_r and σ for row i ◁ Horizontal communication
20: $\quad$ `append` $Q_j \rightarrow Q$
21: $\quad$ `append` **updateFrt** $(lvl, bmap, Q, Qoff, d, \sigma) \rightarrow Q$
22: $\quad nq \leftarrow$ `number of vertices added to` Q
23: $\quad$ **if** $nq = 0$ **for all** processors **then**
24: $\quad\quad$ **break**
25: $\quad$ **end if**
26: **end while**

round between shortest-path counting and dependency accumulation phases. In contrast to the 2-D BFS case, during each *fold* phase in the 2-D BC algorithm, the σ values must be sent among processes. This limits the scalability of the BC algorithm with respect to the 2-D BFS building block. Rather than exchanging the predecessors list during the traversal step as usual in a distributed implementation of the Brandes' algorithm, we keep track of the local frontiers computed from each computing node. The combination of the local frontier and the distance array allows building the predecessor/successors information of the BFS tree without paying additional communication costs. As a consequence, the communication cost depends on the number

Algorithm 5 expandFrt

```
Input G(V, E)                                        ◁ G is unweighted graph
Output BC[v], v ∈ V
 1: for each v ∈ CQ in parallel do                   ◁ CQ is the current frontier
 2:     for each neighbor w of v in parallel do
 3:         if bmap[w] = 0 then
 4:             bmap[w] ← 1
 5:             d[w] ← lvl
 6:             r ← row of w's owner
 7:             atomically enqueue w → Q_r
 8:         end if
 9:         if d[w] = lvl then
10:             atomically σ[w] ← σ[w] + σ[v]
11:         end if
12:     end for
13: end for
```

of vertices in the graph (i.e., the distance array) instead of the number of the edges. Furthermore, this solution also allows reducing both the memory requirements of the local data structures from $\mathcal{O}(m)$ to $\mathcal{O}(n)$ and, as consequence, the number of `read` and `write` operations on GPUs memory.

Dependency accumulation is described in Algorithm 6. The approach we describe is based on the checking successor technique originally proposed by Madduri *et al.* [MEJ+09]. The intuition behind is that the leaves of the BFS tree do not have successors, therefore the algorithm computes the dependencies from the second to last level by checking successors instead of predecessors. As mentioned before, in line 1 of Algorithm 6, both the distance information d and the σ of the vertices are exchanged among computing nodes in the same row. The dependency $\delta[w]$ is calculated by the shortest path count $\sigma[v]$ and dependency value $\delta[v]$ of all its successors. Each processor accumulates the local contributions to $\delta[w]$ for those successors for which it holds the edge (w, v) (**accumulateDep** procedure on line 5). All the local dependency contributions are exchanged and summed by a reduce operation among the processors having the same index column of the mesh afterwards. The final dependency value $\delta[w]$ is obtained by multiplying the accumulated dependencies over $\sigma[w]$ (procedure **updateDep**). Finally, $\delta[w]$

Algorithm 6 Dependency Accumulation on Multi-GPU

Input $G(V, E)$ ◃ G is unweighted graph
Input Processor P_{ij}
1: exchange d and σ for row i ◃ Horizontal communication
2: $\delta[v] \leftarrow 0, \forall v \in V$
3: $depth \leftarrow lvl - 1$
4: **while** depth > 0 **do**
5: **accumulateDep** $(depth, Q, Q_{off}, d, \sigma, \delta)$ ◃ Accumulate dependencies
6: all reduce δ among column j ◃ Vertical communication
7: **updateDep** $(lvl, Q, Qoff, d, \sigma)$ ◃ Update dependencies
8: exchange δ among row i ◃ Horizontal communication
9: $depth \leftarrow depth - 1$
10: **end while**

values are exchanged among processors on the same row (line 8) since they are required for the next iteration. The procedure **accumulateDep** is described in Algorithm 7. In detail, the algorithm first selects the vertices in the local frontier Q (line 1) in order to verify if their neighbors are successors (line 4). An atomic operation is performed to update the local dependency $\delta[w]$.

Algorithm 7 AccumulateDep

1: $CQ \leftarrow Q[Q_{off}[depth]]...Q[Q_{off}[depth - 1]]$
2: **for each** $w \in CQ$ in parallel **do**
3: **for each** neighbor v of w in parallel **do**
4: **if** $d[v] = d[w] + 1$ **then**
5: atomically $\delta[w] \leftarrow \frac{1+\delta[v]}{\sigma[v]}$
6: **end if**
7: **end for**
8: **end for**

4.2.4.1 Communication optimization

Finally, the proposed distributed algorithm is amenable to the overlap of MPI communication and CPU-GPU data transfer. Although Nvidia provides several techniques to reduce communication overhead such as GPUDirect RDMA, we adopt a simple overlap mechanism between two consecutive communications, whereby the cost of the communication through the PCI bus can be hidden. In particular, right after the shortest path counting phase,

both the distance vector d and σ values are exchanged among processors in the same grid row. Since the computation is totally delegated to GPU, usually two consecutive independent communications comply with the following pattern:

1. synchronous-copy of σ from GPU to CPU;

2. exchange of σ among processors in the same grid row;

3. synchronous-copy of σ from CPU to GPU.

4. synchronous-copy of d from GPU to CPU;

5. exchange of d among processors in the same grid row;

6. synchronous-copy of d from CPU to GPU.

In this naive pattern, data transfer procedure ends after six synchronous steps. However, by exploiting Cuda Asynchronous Copy operations and Cuda Streams, the two communications can be completed in four steps (see Figure 4.2):

1. asynchronous-copy of σ from GPU to CPU; asynchronous-copy of d from GPU to CPU;

2. exchange of σ among processors in the same grid row;

3. asynchronous-copy of σ from CPU to GPU; exchange of d among processors in the same grid row;

4. asynchronous-copy of d from CPU to GPU.

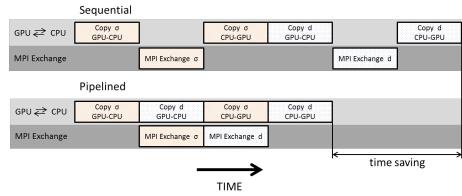


Figure 4.2: Overlapping of GPU - CPU data transfer with MPI communication.

4.2.5 Sub-clustering

A Multi-Source approach for the BC computation offers a significant speed-up on a single-GPU, provided that enough extra-memory (e.g., for the replication of σ and δ arrays) is available [Sar15]. In addition, on distributed systems the replication of data-structures may increase the communication among computing nodes and increase the synchronization requirements. For example, the approach proposed by Buluc *et al.* [BG11] encapsulates three levels of parallelism: columns of F provide parallelism over starting vertices, columns in M' and rows of F provide parallelism over the vertices in each frontier. Finally, rows of M' encapsulate edge (adjacency) parallelism of each frontier vertex. However all the processors in the mesh are involved in the communication during traversal steps. Therefore, to the best of our knowledge, using the single-GPU Multi-Source approach as a basis for a fully distributed BC algorithm does not appear to be the best option. On the other hand, on distributed systems, a coarse-grained approach enables to obtain a very good speed-up by replicating the data structures among computing nodes in order to work on multiple vertices at the same time. As mentioned above, this approach limits the maximum size of the graph that can be processed (e.g., the Twitter graph [KLPM10] cannot be stored in the memory of a single GPU). However, it is possible to obtain a significant improvement of performance by combining fine- and coarse-grained approaches at cluster level abstraction. Within this context, we propose a new solution which

combines graph distribution and graph replication on a Multi-GPU system. Although the present work is focused on BC and Multi-GPUs systems, the approach is more general and can be followed for most problems (i.e., diameter computation, all-pairs-shortest-paths, transitive closure, etc...) that require multiple, independent breadth-first search executions on graphs too large to fit in a single computing node. A set of processors is split into sub-clusters. Each sub-cluster, in turn, is organized as a bi-dimensional grid of processors. Processing nodes in the same sub-cluster work at the fine-grained level: the graph is distributed among the nodes according to a 2-D partitioning, and partial BC values are calculated starting from a subset of vertices. Independent sub-clusters work at the coarse-grained level: the whole graph and additional data structures are replicated in each sub-cluster. In the end, a reduce operation updates the final BC scores. Even if the amount of work in each sub-cluster can be different when processing graphs with multiple connected components, with the sub-clusters solution it is possible to take advantage of both fine- and coarse-grained approaches (see Section 4.4.3). Let p be the number of processors available/requested in the cluster, and let fd be the factor of graph distribution (indicating the size of the mesh of the sub-cluster). The factor of replication of the graph (fr) is defined by $fr = \frac{p}{fd}$ and, in our implementation, it determines the number of sub-clusters. A simple example is shown in Figure 4.3. On the contrary to existing solutions [BG11], which involves all p processors in the communication, sub-clustering technique involves only fd processors in a subcluster during traversal steps (except for the final reduction operation). Furthermore, our approach is not limited to a 2-D partitioning, so other partitioning strategies can be adopted. Both the fd and fr factors must be taken into account to achieve the best performance. Concerning practical aspects, we implement this solution by creating a hierarchy among processes managed by different MPI communicators.

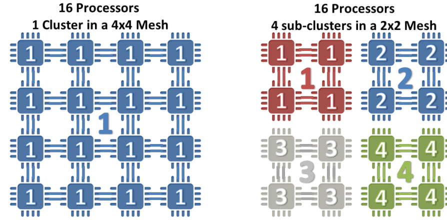


Figure 4.3: Sub-clustering. On the left side the configuration ($p = 16$, $fd = 1$ and $fr = 1$) enables a pure fine-grained strategy. On the right side, a sub-cluster configuration with $p = 16$, $fd = 4$ and $fr = 4$.

4.3 Heuristics for Exact Betweenness Centrality

An exhaustive evaluation of betweenness centrality scores requires solving the SSSP problem starting from each vertex. For large-scale graphs with millions of vertices, computing all SSSPs is a formidable challenge. Nevertheless, in some cases, the betweenness centrality of some sub-structures of the graph, or of vertices with specific properties, can be analytically computed with no need to execute Brandes' algorithm [SSKC13, PEZ+15]. For example, Puzis et al. proposed two heuristics to speed-up the BC computation [PEZ+15]. The first one, contracts structurally-equivalent nodes (nodes that have the same neighbours) into one "special" node. The second heuristics relies on finding the biconnected components of the graph and contracting them as well. The BADIOS framework [SSKC13], tries reducing the computation by shattering (bridges and articulation vertices) and compressing (side and identical vertices). Moreover, focusing on compression based techniques, vertices with exactly one neighbor (degree-1 vertices) have BC score 0, since they are end points and cannot be crossed by any shortest path. As a matter of fact, a careful handling of degree-1 vertices improves overall performance of the implementation of Brandes' algorithm: a) by skipping the execution of

Brandes' algorithm rooted from degree-1 vertices; b) by reducing the number of vertices to traverse. Formally, let $G = (V, E)$ be an undirected and unweighted graph with $n = |V|$ vertices and $m = |E|$ unordered pairs, let $(u, v) \in E : deg(u)$. Since all the shortest paths terminating into a degree-1 vertex have to pass through its neighbor, the contribution $\delta_{sv}(w)$ could be not necessarily equal to 0 (where w is a successor of v). From the algorithm point of view, degree-1 reduction extends Brandes' algorithm by adding a preprocessing procedure and by employing a different formulation for dependencies computation. In detail, the preprocessing step computes $\forall (u, v) \in E : deg(u) = 1$:

$$\begin{aligned} \omega(v) &= \omega(v) + 1; \\ BC(v) &= BC(v) + 2 \cdot (n - \omega(v) - 2) \end{aligned} \tag{4.3.1}$$

where $\omega(v)$ represents the contribution of u to v and initially is set equal to 0. When a degree-1 vertex u is detected, the value $\omega(v)$ of its neighbor v is incremented, and u is removed from the graph. When u is removed from the graph, the value $BC(v)$ needs to be updated in order to consider the contribution of paths starting from all other vertices connected to v and terminating in u. Notice that, n does not correspond to the number of vertices in the graph, but to the number of vertices in the connected component of v. After the preprocessing step, Brandes' algorithm is executed over the residual graph $G'(V', E')$ obtained by the degree-1 removal procedure. Concerning dependency accumulation, Formulas 4.1.2 and 4.1.3 can be re-defined as follows:

$$\begin{aligned} \delta_s(v) &= \sum_{w : v \in pred(w)} \frac{\sigma_{sv}}{\sigma_{sw}} (1 + \delta_s(w) + \omega(w)) \\ BC(v) &= \sum_{s \neq v} \delta_s(w) \cdot (\omega(s) + 1) \end{aligned} \tag{4.3.2}$$

The first formulation of the degree-1 reduction was described by Baglioni *et al.* [BGPL12]. Several works described the degree-1 reduction implementa-

tion on Single GPU [SKSc13,Sar15]. Moreover, Bernaschi *et.al* proposed and evaluated a distributed degree-1 reduction preprocessing based on a 1-D partitioning [BCV15]. Another approach, based on degree-1, consists in re-using the shortest path tree from the vertex adjacent to a degree-1 vertex to cut down on computation [BHM11]. Indeed the traversal step from a degree-1 vertex is not required if the shortest path tree from its adjacency is stored. That solution does not require the preprocessing step, but, at the same time, it does not take advantage of graph compression.

4.3.0.1 Degree-1 reduction

In this Section, we discuss our algorithm for the removal of degree-1 vertices[a,b]. Unlike previous approaches, we provide a distributed preprocessing algorithm described by the pseudo-code in Algorithm 8. Degree-1 reduction requires to identify vertices with degree one and this task is easier to accomplish if each vertex, along with all its edges, is stored on the same processor. This can be easily obtained with a 1-D partitioning. First, the edges are sorted by the antecedent vertex u and processed sequentially: when a degree-1 vertex u is discovered, $\omega[v]$ is incremented and the edge (u, v) is added to the list R of the removed edges. Otherwise, all the edges from u are appended to the new edge list E' of the residual graph. In an undirected graph, for each edge (u, v) the symmetric edge (v, u) must be removed as well. The contribution of degree-1 vertices to BC scores cannot be computed during preprocessing since we support graphs with multiple connected components, on the contrary to previous solutions. By observing the formula $BC(v) = BC(v) + 2 \cdot (n - \omega(v) - 2)$, we already highlighted that n corresponds to the number of vertices in the same connected component of v, including degree-1 vertices. For any vertex s, we can compute n_s, the number of vertices of its connected component, during the shortest paths counting.

[a] For the sake of simplicity we do not remove tree vertices from the graph by calling repeatedly the preprocessing (tree vertices removal).

[b] The preprocessing is implemented only on CPU.

When a new vertex v is discovered during graph traversal from root vertex s, n_s is updated as follows: $n_s = n_s + \omega[v]$. Computing n_s is required whenever $\omega[s] \neq 0$, in other words, only if vertex s is connected to a degree-1 vertex. There are two alternatives for the computation of n_s: *i*) using atomic operations during shortest paths counting; *ii*) using a parallel reduction of the distances array before the update of the betweenness centrality score. In both the cases, the procedure should not consider the contribution $\omega(v)$ of unvisited vertices. As to the performance, the best solution depends on the cost of atomic operations. Finally, since our approach does not require information about the connected components of the graph, the computing time of the preprocessing step decreases.

Algorithm 8 Degree-1 Preprocessing

```
Output ω[v], G'(V', E')
 1: R ← empty List
 2: E' ← empty List
 3: if u mod #P = P_i : (u, v) ∈ E then            ◁ P_i is processor i^th
 4:     assign (u,v) to E_i
 5: end if
 6: sorting E_i by u
 7: for (u, v) ∈ E_i do
 8:     if ∄(w, z) ∈ E_i : u = w then               ◁ (w,z) the successor or predecessor in E
 9:         append (v, u) → R
10:         ω[v] = ω[v] + 1
11:     else
12:         append (u, v) → E'
13:     end if
14: end for
```

4.3.0.2 Deriving BC of degree-2 vertex

We also propose a new technique based on dynamic programming, to compute the BC score of degree-2 vertices without executing Brandes' algorithm from them explicitly. In contrast to the degree-1 reduction or other techniques which modify the topology of the graph, the degree-2 heuristics exploits the information of the shortest-path tree of the two neighbors to derive both the shortest-path tree and the dependency of the degree-2 vertex. A

similar intuition is barely sketched in literature [Mad08]. The author proved that it is possible to build the shortest path tree from an arbitrary vertex in the graph without re-traversing the graph when the shortest-path trees from all its adjacencies are known. However, the author did not provide either an algorithm or related results. The following notation is used in the rest of the Section. We denote with $lvl_s(v)$ the discovery depth or level or unary distance of v during a traversal step from a source vertex $s \neq v$. Let c be a vertex with $deg(c) = 2$ and a and b its own neighbors. We also use the symbols frt_s and frt_s^k to denote the BFS tree (or set of frontiers) of a vertex s and the set of vertices discovered at level k respectively. Our goal is to compute the shortest path and the dependencies of a degree-2 vertex by re-using shrewdly the information provided by the execution of Brandes' algorithm for its adjacencies. To that purpose, we need to determine:

1. the frt_c from the frontiers of its own adjacencies;
2. the number of the shortest paths of each vertex from c;
3. the dependencies of c.

Concerning the first point, the key idea behind the degree-2 heuristics is that the frontiers of a degree-2 vertex can be obtained by merging the frontiers of the two neighbors. To do that, the BFS trees of its own adjacencies must be stored. Furthermore, it is apparent that the shortest paths from c to a vertex $v \in frt_c$ must pass through either a, b or both; indeed we may determine the relation between the level at which a vertex is discovered starting from c and the level of the same vertex discovered starting from the degree-2 neighbors.

Lemma 4.3.1 *Let c be a degree-2 vertex, and let a and b be neighbors of c such that a, b and $c \in V$ in an unweighted graph $G = (V, E)$. Each vertex v in the frontiers of a and b obeys the following rules: i) $v \in frt_c$. ii) v is discovered in frt_c at level $lvl_c(v) = min\{lvl_a(v), lvl_b(v)\} + 1$*

The number of shortest paths passing through vertex v in frt_c depends on which shortest-path is followed when v is discovered at $lvl_c(v)$, a path through a or b. In other words, $\sigma_c(v)$ is equal to $\sigma_a(v)$ iff $lvl_a(v) < lvl_b(v)$; likewise, $\sigma_c(v)$ is equal to $\sigma_b(v)$ iff $lvl_b(v) < lvl_a(v)$. If v is discovered at the same level from both a and b (i.e., $lvl_a(v) = lvl_b(v)$), then $\sigma_c(v)$ is defined by the shortest paths passing via a and b. More in detail, we first recall the Bellman's observation.

Lemma 4.3.2 *(Bellman criterion) A vertex $v \in V$ lies on a shortest path between vertices s, $t \in V$, if and only if $d(s,t) = d(s,v) + d(v,t)$.*

By properly applying Bellman criterion we find that:

$$\sigma_c(v) = \begin{cases} \sigma_a(v) & \text{if } lvl_a(v) < lvl_b(v) \\ \sigma_b(v) & \text{if } lvl_a(v) > lvl_b(v) \\ \sigma_a(v) + \sigma_b(v) & \text{if } lvl_a(v) = lvl_b(v) \end{cases} \tag{4.3.3}$$

We prove Lemma 4.3.1 by induction. At level 1, frt_c^1 is composed by a and b by definition of degree-2 vertex. At level 2, frt_c^2 is composed by $frt_a^1 \bigcup frt_b^1$. At level 1, frt_a^1 is composed by c (by definition of a), a set of vertices $v \neq c \neq b$ iff $\exists(a,v) \in E$ (case 1) and b if there exists an edge (a,b) (case 2). Likewise, frt_b^1 is defined by c, a set of vertices $v \neq c \neq a$ iff $\exists(b,v) \in E$ (case 1) and a if there exists an edge (a,b) (case 2) which we omit. If a generic vertex v is discovered at lvl^{i-th} from a, then the path to reach v from b is longer or equal at most. With respect to c, v is reachable from the path passing through a or b. A naive implementation is described by the pseudo-code in Algorithm 9. The algorithm makes it possible to get rid of the computation of the shortest path from c only, so the dependency accumulation is still required.

The BFS tree rooted in c can be derived by simply sorting lvl_c. This solution only saves the time for the graph traversal. We may achieve a greater benefit if betweenness contributions from c are directly added on-the-fly while the dependency accumulation steps for its two neighbors a and b are performed. This solution avoids both the execution of the Brandes' algorithm

Algorithm 9 Shortest-path tree computation of a degree-2 vertex from its own neighbors

Input $G(V, E)$, $sigma_a[]$, $sigma_b[]$, $lvl_a[]$ and $lvl_b[]$
Output $\sigma_c[]$, $lvl_c[]$

```
1:  sigma_c[v] ← 0, lvl_c[v] ← ∞   ∀v ∈ V
2:  for each v ∈ V in parallel do
3:      if lvl_a[v] = lvl_b[v] then
4:          σ_c[v] ← σ_a[v] + σ_b[v]
5:          lvl_c[v] ← lvl_a[v] + 1
6:      end if
7:      if lvl_a[v] < lvl_b[v] then
8:          σ_c[v] ← σ_a[v]
9:          lvl_c[v] ← lvl_a[v] + 1
10:     else
11:         σ_c[v] ← σ_b[v]
12:         lvl_c[v] ← lvl_b[v] + 1
13:     end if
14: end for
```

from the vertex c and the explicit evaluation of lvl_c and σ_c. As explained before, the BC contributions δ_s of a vertex s are computed recursively by re-traversing the BFS tree rooted in s according to Formula 4.1.2. As a matter of fact, δ_s at each level depends on the contributions at the deeper level. The first problem to be considered is when the vertices contributions of c should be added to δ_c since the order of visit may be different between its own neighbours. This is accomplished by modifying Brandes procedure so that dependency accumulation steps for a, b and c are performed together "level by level". During this step, the frontiers of a and b are dynamically merged (without storing them in a new BFS tree of c explicitly) and contributions of c dependencies are added as well. We call this technique "Dynamic Merging of Frontiers (DMF)". Algorithm 10 and Algorithm 11 modify the procedure described in Algorithm 1 in [Bra01] (at lines 24 - 28) by implementing DMF. We first compute σ_a, lvl_a, σ_b and lvl_b (following the procedure described in Algorithm 5). At line 1 of Algorithm 10, the deeper BFS tree between a and b is evaluated. The vertices in the leaves of a and b contribute to the δ_c iff they are both discovered at the same level. For instance, let w be a vertex belonging to the leaves of the BFS tree of a. It may be discovered

two levels before by b (if $(a, b) \notin V$). In this case, the contribution of the predecessors of w should be taken into account in $\delta_c(w)$ when w is visited in the dependency accumulation of b. Moreover, we have to consider the shortest path tree of b in the dependency accumulation equation. When both the current depths of the BFS trees are synchronized, the procedure simultaneously computes, level-by-level, the dependencies for a, b and c. Algorithm 11 shows the dependency accumulation of a child of a degree-2 vertex c according to Formula 4.3.3. In detail, within each iteration of the dependency accumulation, for each vertex in the frontier of a, we calculate the dependency accumulation as in the original algorithm but we check, in addition, if the vertex should be considered for c. We do the same for each vertex in the frontier of b. Notice that, when a predecessor v of w is discovered at the same level in a and b, the $\sigma_c(v)$ is defined for both $\sigma_a(v)$ and $\sigma_b(v)$ (line 6). Like in Algorithm 7, the procedure exploits atomic operations to update δ_c.

Finally, we can conclude with the following result.

Theorem 4.3.3 *Let c be a degree-2 vertex, and let a and b be neighbors of c such that a, b and $c \in V$ in an unweighted graph $G = (V, E)$. The shortest path tree of c can be derived iff the level of each vertex discovered in BFS trees rooted at a and b is given respectively.*

The effectiveness of the degree-2 heuristics depends on the order in which the vertices are processed in the main loop of Brandes' algorithm. When dealing with a degree-2 vertex, first we have to perform the shortest paths counting steps from its own adjacencies. At the same time, a degree-2 vertex should be processed together with its two neighbors. Moreover, we cannot execute Brandes' procedure of a generic vertex v without knowing if v is a neighbor of a degree-2 vertex. The solution proposed allows computing the dependency values of a, b and degree-2 vertex c in a single computation by concurrent execution of the dependency accumulation of a and b level-by-level. This happens when the adjacencies of a degree-2 vertex do not belong to the adjacencies of other degree-2 vertices. As a matter of fact, we

Algorithm 10 Dependency Accumulation steps based on Dynamic Merging of Frontiers

Input $G(V,E)$, $sigma_a[]$, $sigma_b[]$, $lvl_a[]$ and $lvl_b[]$
Output $\delta_a[]$, $\delta_b[]$ and $\delta_c[]$
1: $depth \leftarrow \max\{depth_a, depth_b\}$
2: **while** $depth > 0$ **do**
3: **if** $depth = depth_a$ **then**
4: DepAcc-deg2 ($\text{DepInfo}_a, \sigma_a, lvl_a, \sigma_b, lvl_b$)
5: ◃ DepInfo_a denotes the information required in Alg. 6 at line 5 related a vertex a.
6: $depth_a \leftarrow depth_a - 1$
7: **end if**
8: **if** $depth = depth_b$ **then**
9: DepAcc-deg2 ($\text{DepInfo}_b, \sigma_a, lvl_a, \sigma_b, lvl_b$)
10: $depth_b \leftarrow depth_b - 1$
11: **end if**
12: $depth \leftarrow depth - 1$
13: **end while**

cannot solve all degree-2 vertices by applying Algorithm 10 even if the graph is composed of degree-2 vertices only. For instance, let $C = (V, E)$ be a cycle graph where $|V| = n$ and each vertex has degree 2. The algorithm computes the BC score of, at most, $\frac{n}{2}$ or $\lfloor \frac{n}{2} \rfloor - 1$ (if n is odd) vertices without performing Brandes algorithm explicitly. As to the memory requirements, the heuristics requires $\mathcal{O}(n)$ extra memory-space since both σ_a and lvl_a depend on the number of vertices of the graph. In the present work, we do not address the problem of how to find out the minimal set of vertices for which we need to store the shortest path trees. However, for experimental validation, we simply check if a vertex v is a child of a degree-2 vertex. If this occurs, the algorithm performs shortest paths counting from both v and the other child of its own predecessor. On the other hand, if v is a degree-2 vertex, we execute the shortest paths counting of its own adjacencies and then Algorithm 10 is used in order to derive the contribution of v to the BC score.

Algorithm 11 Augmenting the betweenness centrality accumulation from a left-child of a degree-2 vertex

Input $DepInfo_a, Q_a, \sigma_a, lvl_a, \sigma_b, lvl_b, \delta_c$
1: $CQ_a \leftarrow Q_a[Q_{off}[depth]]...Q_a[Q_{off}[depth_a - 1]]$
2: **for each** $w \in CQ_a$ in parallel **do**
3:     **for each** neighbor v of w in parallel **do**
4:         **if** $d[v] = d[w] + 1$ **then**
5:             `atomically` $\delta_a[w] \leftarrow \frac{1+\delta_a[v]}{\sigma_a[v]}$
6:             **if** $lvl_a[v] = lvl_b[v]$ **then** atomically $\delta_c[w] \leftarrow \frac{1+\delta_c[v]}{\sigma_a[v]+\sigma_b[v]}$
7:             **end if**
8:             **if** $lvl_a[v] < lvl_b[v]$ **then** atomically $\delta_c[w] \leftarrow \frac{1+\delta_c[v]}{\sigma_a[v]}$
9:             **end if**
10:         **end if**
11:     **end for**
12: **end for**

4.4 Experimental Result on Unweighted Graphs

In the present section we provide both the comparison against state-of-the-art the performance our the proposed solutions. We first compare MGBC with other implementations on a single GPU. Actually, most of them do not offer full support for a distributed Multi-GPU configuration. Some of them [MB14b] on distributed systems, support only coarse-grained parallelism, where each GPU works independently on a replica of the same graph. All those solutions cannot be used for very large graphs, like Friendster or Twitter [KLPM10] since those graphs do not fit in the memory of a single system, therefore some graphs of our dataset are omitted for this experiment. Then we study weak and strong scalability of MGBC in order to evaluate the ratio between computation and communication on different graphs. We also evaluate the impact of the optimizations techniques above described. Finally, we measure the speedup provided by heuristics with respect to our base (heuristics-free) implementation. In particular, we evaluate the speedup of the degree-2 heuristics and its impact on graphs having a long diameter like road networks.

4.4.1 Evaluation Platforms and Data Sets

Numerical experiments have been carried out on two different systems: *Piz Daint* at Centro Svizzero di Calcolo Scientifico (CSCS) and *Drake*, a server equipped with two K80 GPU available at National Research Council of Italy. Piz Daint is a hybrid Cray XC30 system with 5272 computing nodes interconnected by an Aries network with Dragonfly topology [KDSA08]. Each node is powered by an Intel Xeon E5-2670 CPU and a Tesla K20X Nvidia GPU and it is equipped with 32 GB of DDR3 host memory and 6 GB of DDR5 GPU memory. The code has been generated using the GNU C compiler version 4.8.2, CUDA C compiler version 6.5 and Cray MPICH version 7.2.2 on Piz Daint and OpenMPI 1.8.4 on Drake. We employ the exclusive prefix sum implemented in the Thrust Library [HB]. The code uses 32-bit data structures except for graph generation. We usually report the time (in seconds) for total BC computation. Sometimes, in order to compare our results with those reported in state-of-the-art literature, we also report the *traversed edges per second* (TEPS) defined as $TEPS_{bc} = \frac{m \times n}{t}$, where n is the number of vertices or a subset of them, m is the number of (undirected) edges, and t is the execution time of the BC computation[c]. However, for very large graphs we measure the time only for a representative subset of source vertices [d]. In this case, the time to compute the BC score of the whole graph is estimated. We measured the performance for both R-MAT [CZF04] and real-world graphs (see Table 4.1) [LK15]. The R-MAT graphs are generated setting parameters a, b, c, and d equal to 0.57, 0.19, 0.19, 0.05 respectively.

4.4.2 Single-GPU

We compare our solution on single GPU (without any heuristics or optimization) to those proposed in McLaughlin and Bader [MB14b], Sarıyüce *et al.* [SKSc13] and Gunrock [WDP^{+}15][e] on Drake system. The solution pro-

[c] We do not consider disconnected vertices with respect to a source.
[d] Source vertices are selected randomly among non-isolated vertices.
[e] The version of Gunrock used for the experiments is dated 2015.

Graph	Vertices	Edges	dg-1 %	dg-2 %	d
com-amazon	334863	925872	4.68	10.5	44
com-youtube	1134890	2987624	53.00	15.4	20
RoadNet-CA	1965206	2766607	16.27	10.4	849
RoadNet-PA	1088092	1541898	17.30	7.1	786
com-LiveJournal	3997962	34681189	19.20	10.7	17
com-Orkut	3072441	117185083	2.21	1.4	9
Friendster	65608366	1806067135	1.20	9.0	32
Twitter	41652240	1468365033	4.50	5.5	18

Table 4.1: Features of the real-world graphs used for the tests. **d** represents the diameter whereas dg-1 and dg-2 represent the percentage of vertices having one and two neighbors respectively

.

posed in [Sar15] employs a multi-source approach and extends the solution reported in Table 4.2 (Sariyüce mode-4) by using coarse-grained parallelism in shared memory systems achieving a further speed-up. Unfortunately, the implementation described in [Sar15] is not available for a direct comparison. The main goal of these tests is to show the impact of the data-thread mapping strategy. The MGBC implementation for a single GPU is derived by turning off network and related host-device communications. We report and focus the performance of BC implementations on graphs with different characteristics. In Table 4.2, we report the mean time (in seconds) for each implementation. Since other codes do not allow a random selection of the vertices, the mean time is computed over the first 10000 vertices of the biggest connected component. Concerning Sarıyüce's implementations, we evaluated two of their data-mapping strategies. The first one, called *mode-2* employs edge-based GPU parallelism, whereas the second one, *mode-4*, uses virtual-vertices with stride access [SKSc13]. Experiments show that the hybrid approach of McLaughlin performs better than others on graphs with a pretty small edge factor and long diameter, like road networks. Instead on other kinds of graphs, the performance of their approach is not satisfactory. On the other hand, the vertex-virtualization technique achieves very good performance on more dense graphs. However, such an approach re-

Graph	Mclaugh-lin	Sarıyüce mode-2	Sarıyüce mode-4	Gunrock	MGBC
com-Amazon	0.008	0.009	0.006	0.007	**0.005**
RoadNet-CA	**0.067**	0.371	0.184	0.298	0.085
RoadNet-PA	**0.035**	0.210	0.114	0.212	0.071
com-LiveJournal	0.210	0.143	**0.084**	*nt*	0.100
com-Orkut	0.552	0.358	**0.256**	*nt*	0.314

Table 4.2: Comparison with other single GPU implementations on real-world graphs. The time reported is expressed in seconds. The acronym `nt` stands for "execution does not terminate".

quires an *a priori* tuning of the virtual-vertex parameter. By changing the virtualization parameter the performance may decrease. Although the design is focused on distributed systems, our BC implementation achieves good performance without requiring any specific tuning.

4.4.3 Multi-GPU and Sub-Clustering

We first evaluated MGBC performance in strong scaling experiments. In particular, we ran MGBC from 10000 vertices on both R-MAT and real-world graphs without using neither heuristics nor the prefix-sum optimization. In particular, Figure 4.4 shows the scalability for an R-MAT graphs with $S = 23$ and $EF = 16, 32$. Our solution has a very good scaling up to 128 GPUs with EF 32. Moving from 1 to 2 nodes, there is only a $\sim 40\%$ improvement due to communication overhead. In Figure 4.5, we report a breakdown of the total time in computation, communication and *sigma-delta* communication which measures the time spent in exchanging σ, d and δ among the processors. From 2 to 8 nodes, the scalability is almost linear. In those cases, the communication represents a small fraction of the total time ($\sim 16\%$). From 16 to 32 nodes, we observe that the computation decreases linearly whereas the communication remains almost the same. By employing 64 GPUs, both the computation and the communication decrease, however the computation represents one third of the total time. At 128 nodes the computation and the communication are the same, then with more than 256 GPUs, the communi-

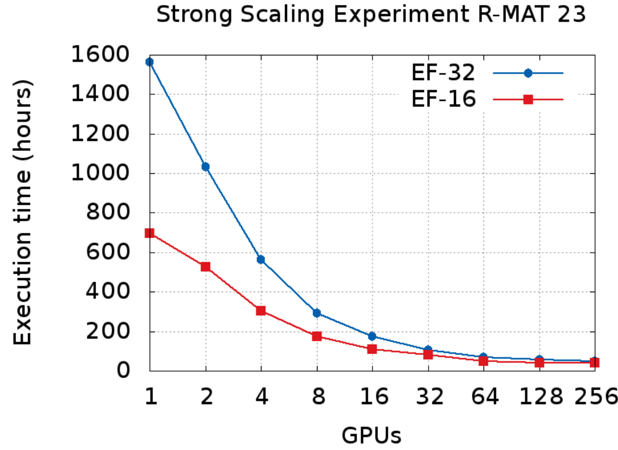


Figure 4.4: Strong scaling experiments for R-MAT graphs with $S = 23$ and $EF = 16, 32$.

cation dominates the computation and the algorithm does not scale anymore. The *sigma-delta* communications, in the worst case (256 GPUs), represent $\sim 9\%$ of the total time. The black curve denotes the mean time for a BC round. Concerning real graphs, the strong scaling for *Friendster* and *Twitter* graphs is evaluated in Figure 4.6. The mean time of a BC round is figured out by looking at the y2-axis. Notice that the minimum number of GPUs required to store the entire graph is 16 for both graphs. Although we observe a good scalability (up to 256 GPUs for *Friendster*), the mean time of a BC round is still pretty high (i.e., 0.601 seconds). As a consequence, the exact computation of the betweenness centrality for both graphs is not feasible in a reasonable amount of time.

Figure 4.7a, 4.7b illustrate the performance of MGBC for graphs that a single GPU can not handle due to memory limits in weak scale experiments. Although the amount of data is the same for each GPU, the time required to compute BC is not constant. Contrary to the BFS algorithm, where only one vertex is marked as predecessor, during the traversal steps, the evaluation of the BC requires the counting of all shortest paths. When in

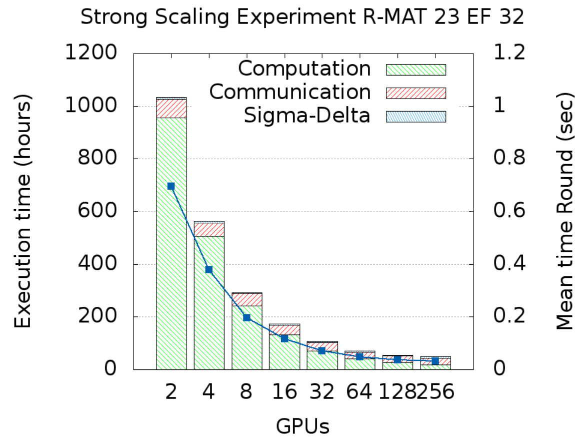


Figure 4.5: Strong scaling experiments for R-MAT graphs with $S = 23$ and $EF = 32$.

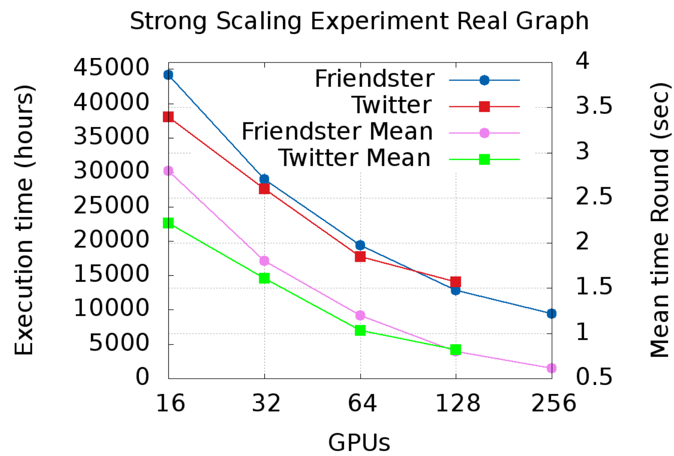
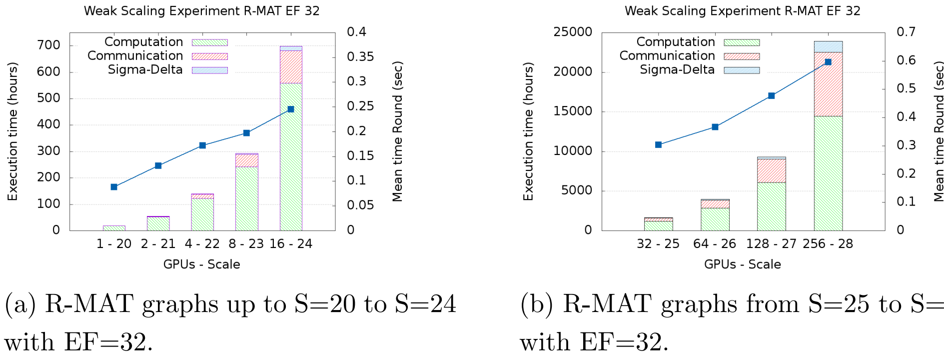


Figure 4.6: Strong scaling experiments for Twitter and Friendster graphs.



(a) R-MAT graphs up to S=20 to S=24 with EF=32.

(b) R-MAT graphs from S=25 to S=28 with EF=32.

Figure 4.7: Weak Scaling Experiments

the current frontier there are more vertices which expand the same successor, the time required to count the shortest paths and to accumulate dependency contributions increases due to atomic operations (see line 10 of Algorithm 5 and line 5 of Algorithm 7). In particular, the time increases linearly from $S = 20$ up to $S = 28$ with $EF = 32$. The mean time for a BC round for an R-MAT $S = 28$ is 0.590 seconds (29 GTEPS). To the best of our knowledge, there are no studies of BC on R-MAT graphs with scale greater than 24.

4.4.3.1 Sub-clustering

Table 4.3 shows the total time required to compute the BC for the Orkut graph when the replication factor fr increases. The results confirm the intuition that, for a fixed number of processors p, a smaller factor of distribution offers the best performance. With $p = 256$ ($fr = 64$), the time required for the full BC computation of the Orkut graph is 2.3 hours. Concerning the scalability, the sub-clustering technique requires that each sub-cluster had a balanced workload. The partial BC scores are stored locally for all of the GPUs into a sub-cluster. Finally, the scores at sub-cluster level are aggregated into the global BC scores by a reduce operation. On *Orkut* graph, the workload among sub-clusters is balanced since *Orkut* unveils only one connected component. We also report a comparison between MGBC and the

fr	1	64	128
Time (hours)	211	3.5	1.8
GTEPS	0.9	56.2	111.6

Table 4.3: Total time to compute exact BC for the Orkut graph with $fd = 2$.

BC implementation	Computing nodes (N. of MPI tasks)	Concurrent vertices	Time (sec)
BC-CombBLAS-2D	128 (1024)	128	362.89
BC-CombBLAS-2D	512 (4096)	128	147.55
MGBC-2D	64 (64)	1	300.97
MGBC-2D	128 (128)	1	196.44
MGBC-2D	256 (256)	1	130.68
MGBC-2D	512 (512)	1	94.59
MGBC-SC	512 (512)	8	39.50

Table 4.4: Performance comparison between the BC implementation provided by CombBLAS and different MGBC configurations on an RMAT graph $S = 29$ and $EF = 16$. We report the time required to complete the BC computation from 128 source vertices. In MGBC, the number of concurrent vertices corresponds to fr.

BC implementation provided by CombBLAS [BG11]. The benchmark is on an R-MAT graph with $S = 29$ and $EF = 16$. In Table 4.4, we evaluate MGBC and BC-CombBLAS-2D performance on different mesh configurations. For BC-CombBLAS-2D we exploited all the cores (8) of each compute node on Piz Daint. Notice that BC-CombBLAS-2D only works on a square (at logical level) grid of processors. MGBC-SC denotes MGBC in subcluster configuration with $fd = 64$ and $fr = 8$. The data reported also confirm that our approach in 2-D configuration scales well up to the point when the communication starts to dominate the computation.

4.4.3.2 Optimizations impact

Finally, in Figure 4.8 and Figure 4.9, we report the impact of the prefix-sum-free optimization and overlap technique respectively. In order to evaluate the impact of the prefix-sum optimization, we compared both implementations on graphs with different diameter and density. Figure 4.8 shows

the performance increment both on R-MAT and real graphs. In general, the improvement is more significant on graphs with long diameter since the prefix-sum is performed for each level and those graphs require many iterations. The R-MAT graphs are characterized by short diameter, furthermore when the graph becomes denser the prefix-sum implemented in the Thrust library is more efficient since it tends to achieve the maximum throughput. More details about the performance of scan functions are reported in literature [Mer15]. By observing the results on R-MAT with $S = 16$, we obtain a 14% improvement due mainly to the low throughput of the prefix-sum implemented in Thrust. On the other hand, RoadNet-PA graph is characterized by a long diameter and low density. In this (best) case, our technique offers the highest improvement ($\sim$ 30%). The experiment on the *Orkut* graph (EF $\sim$ 38 and diameter 9) represents the case where the prefix-sum is efficient but its cost is relevant. As a matter of fact, the maximum cost of the prefix-sum is achieved when the algorithm traverses the levels (middle) where the maximum number of vertices is discovered. In the latter case we obtain a 10% improvement. In order to evaluate the overlap technique, we compare the result of strong scaling experiments previously reported with the result obtained when the overlap is off. In the strong scaling experiment, the amount of data stored in a single node varies. In this way, we can measure the overall time for *sigma-delta* exchange. It turns out that by increasing the number of GPUs the cost of the host-device communication decreases. Figure 4.9a and Figure 4.9b remark the effectiveness of our solution both on synthetic and real-world graphs. In particular, the communication of *sigma-delta* can be reduced by a factor of 2.5 when the overlap is enabled. Notice that when the communication dominates the computation, the overlap benefit decreases.

4.4.4 Heuristics

For the degree-1 reduction, we evaluated, first of all, the strong scalability of the preprocessing step. Figure 4.10 illustrates the strong scaling of Algorithm 8 applied to an R-MAT graph with $S = 22$ and $EF = 16$ on

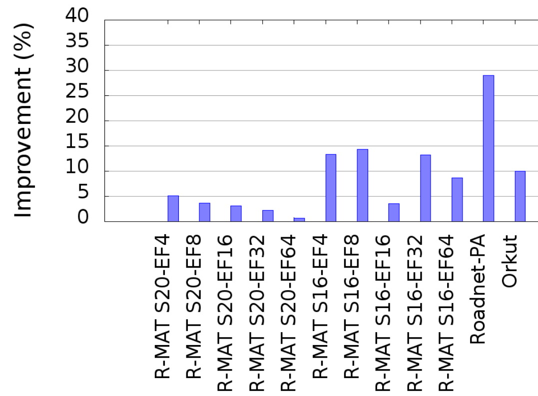
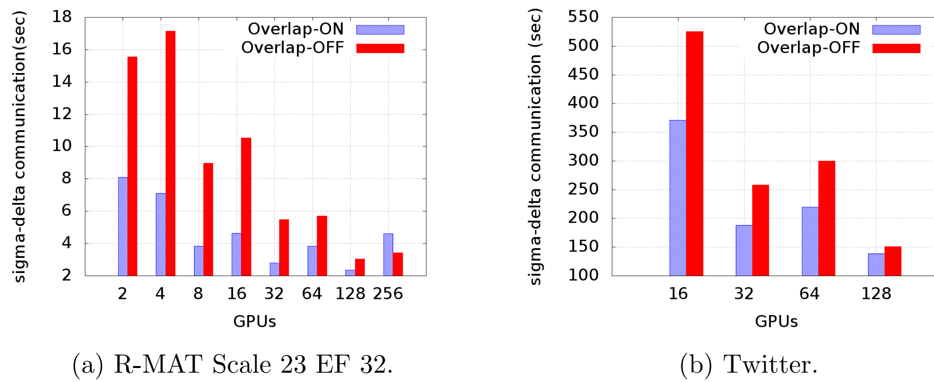


Figure 4.8: Impact of the prefix-sum optimization on single GPU system.



(a) R-MAT Scale 23 EF 32.

(b) Twitter.

Figure 4.9: Impact of the overlapping on synthetic and real-world graphs.

Piz Daint. The algorithm exhibits a near-linear speedup suggesting that the communication does not represent a bottleneck during the preprocessing step. The experiments reported below have been performed on the Drake system. Concerning synthetic graphs, we computed the BC scores of all vertices of a R-MAT graph with $S = 20$ and different EFs exploiting a 2x2 grid of GPUs. More in detail, Table 4.5 shows the mean time of an iteration of MGBC[f], the total time and the preprocessing time when the degree-1 reduction is applied. On an R-MAT graph with EF 16, the preprocessing takes less than 0.02% of the total time offering an increment of performance of 30% compared to the execution with degree-1 off. A more significant improvement can be achieved when the edge factor decreases since the number of degree-1 vertices increases. For example, the execution of MGBC with degree-1 reduction on the *com-youtube* graph is $\sim$ 3 times faster than an execution with degree-1 reduction off. On the contrary to previous works which show only the speed-up of the degree-1 reduction on single GPU, in Figure 4.11 we compare the impact of degree-1 reduction on computation and communication times on distributed systems. It is worth noting that with 4 GPUs the problem is computation-bound therefore the reduction of the total execution time is limited to the gain obtained on the computation. The improvement on the communication is more evident, for example, on an R-MAT graph with $S = 20$ $EF = 4$, where the communication time with degree-1 enabled is halved with respect to the case with degree-1 turned off (see the second bar chart on Figure 4.11).

Graph	**Degree-1 (%)**	**Total time (hour)**	**Mean time (sec)**	**Preprocessing (sec)**	Speed-up
com-Youtube	53.0	1.4 (3.9)	0.009 (0.013)	0.620	2.8x
R-MAT EF4	13.6	1.1 (1.8)	0.012 (0.015)	0.312	1.8x
R-MAT EF16	13.3	2.9 (4.1)	0.020 (0.022)	1.237	1.4x
R-MAT EF32	12.1	5.0 (6.6)	0.029 (0.032)	2.449	1.3x

Table 4.5: Impact on BC processing time due to degree-1 reduction. The value reported in parenthesis are referred to MGBC with degree-1 off.

[f] The mean time is computed considering only connected vertices.

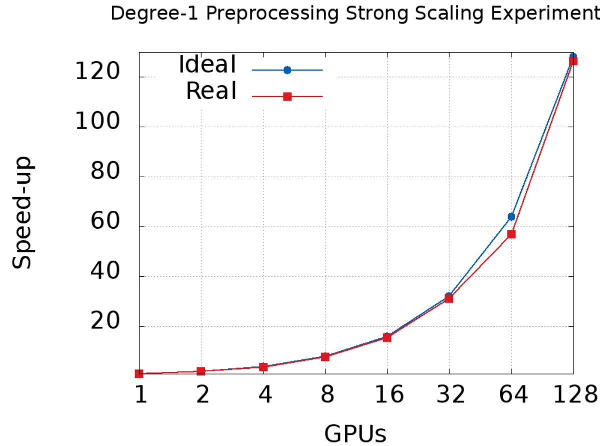


Figure 4.10: Strong scaling experiment of the preprocessing algorithm for a R-MAT graph with $S = 23$ and $EF = 32$.

In Figure 4.12, we show the performance of the degree-2 heuristics presented in Section 4.3.0.2 and in general the impact of heuristics in betweenness computation. We focused on road networks since they present a significant number of degree-1 and degree-2 vertices. In the y-axis, we report the number of vertices processed exploiting the techniques proposed in the present work. For example, with no heuristics enabled, all the vertices of the graph must be processed by MGBC (blue bar). On the other hand, the red and transparent stacks represent the vertices processed by degree-1 and degree-2 heuristics without computing the BC explicitly. The sum of the stacks must be equal to the total number of vertices of the graph (for RoadNet-PA $n = 1090920$). In the y2-axis, we report the total execution time (expressed in hours) for each heuristics. In particular,

- MGBC-H0 represents MGBC without any heuristics turned on.
- MGBC-H1 exploits the degree-1 reduction.
- MGBC-H2 performs MGBC with degree-2 heuristics based on DMF techniques.

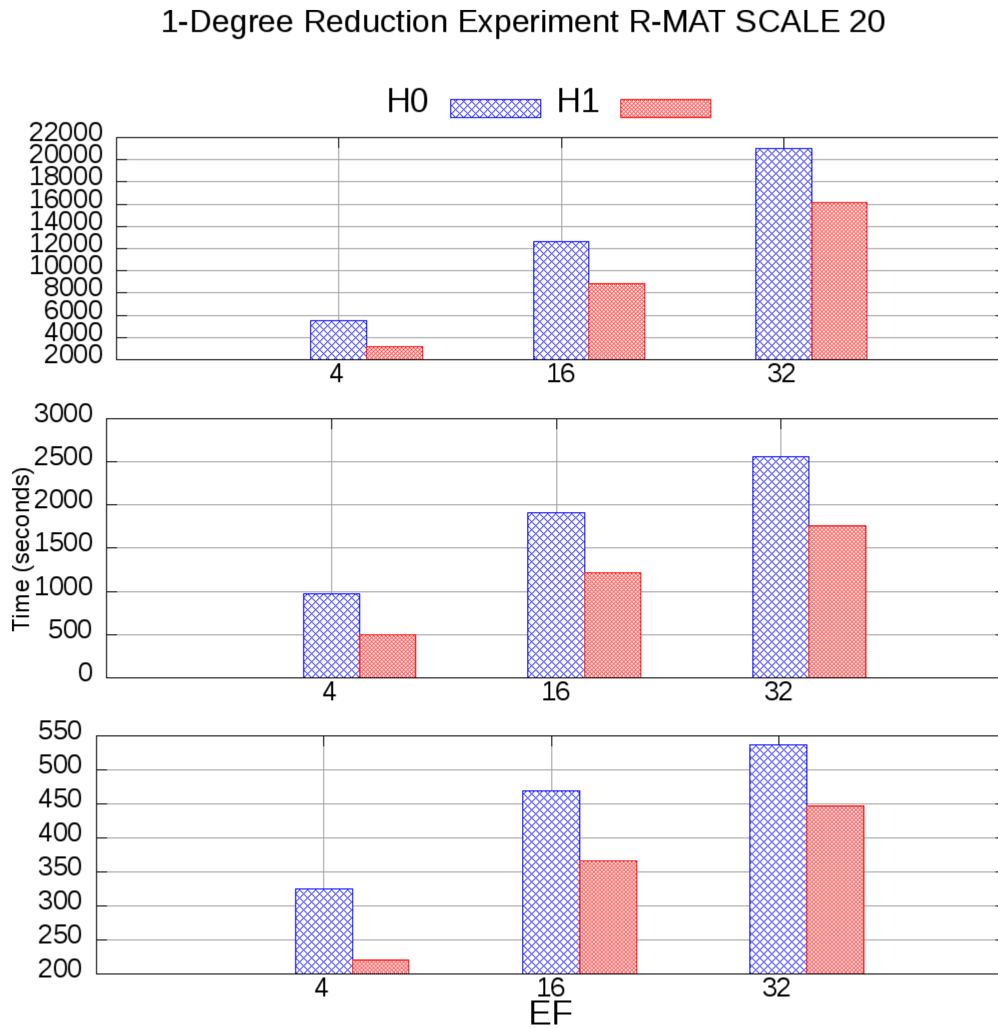


Figure 4.11: Impact of degree-1 reduction on a R-MAT graph with $S = 20$. The bars show the time in seconds of the computation (top), communication (middle) and overlap (bottom) respectively.

- MGBC-H3 combines degree-1 reduction and degree-2 heuristics.

The data reported are obtained running the experiments on Drake in single GPU configuration. With H0, MGBC performs shortest paths counting, dependency accumulation and betweenness update procedure for each vertex in the graph[g]. For RoadNet-PA, the average time to perform these steps is 0.071 seconds whereas the time to solution is about 21 hours. With degree-1 turned on, $\sim$ 17% of vertices are removed from the graph and their BC score contributions are directly computed from their neighbors. We remark that the procedure reduces both the number of vertices to traverse and the number of the vertices to perform MGBC. MGBC-H1 is 17% faster than MGBC-H0, in line with the percentage of degree-1 vertices. In this case, the improvement is mainly due to the reduction in the number of MGBC execution. On networks with a different topology, like the *com-youtube* graph, the improvement may be greater due to the significant reduction in the total number of vertices to be visited. Although the percentage of degree-2 vertices is 7%, we are able to handle only 5% of them with a 5% improvement in terms of MGBC performance (see H2 bar in Figure 4.12). The reason is that 2% of degree-2 vertices share one or both neighbors. In this case, due to our implementation of DMF, we cannot augment the betweenness score of all degree-2 vertices. As a matter of fact, on the contrary to degree-1 reduction, the degree-2 heuristics allows achieving a linear improvement depending only on the number of skipped Brandes' computations. By combining H2 and H3 heuristics, we can achieve an improvement that is not just their sum, since the preprocessing of the degree-1 reduction increases the number of degree-2 vertices. Basically 3-degree vertices which have a degree-1 neighbor become degree-2 after the degree-1 preprocessing step. In our experiment we have $\sim$ 8% of degree-2 vertices added. Although the number of degree-1 vertices processed in H3 configuration is the same, the betweenness score of degree-2 vertices is twice (10%) if compared to the H2 case. The total number of vertices for which we avoid performing a round of MGBC is composed

[g]The disconnected vertices are also taken into account.

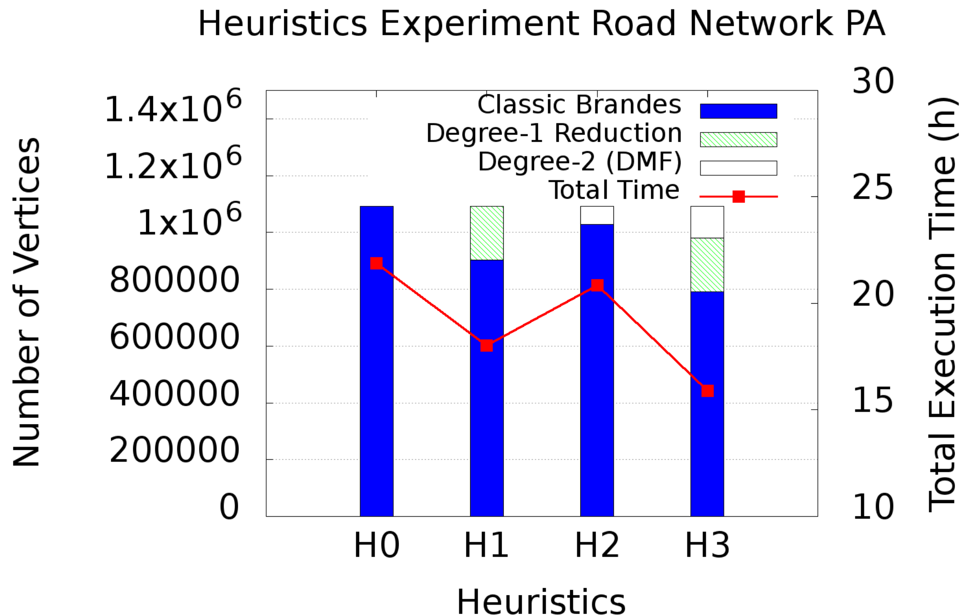


Figure 4.12: Heuristics comparison on RoadNet-PA.

as follows: 17% (due to degree-1 reduction) and 10% computed by degree-2 heuristics. By comparing with MGBC-H0, as expected the total improvement in terms of performance of MGBC-H3 is about 27%. The improvement of MGBC-H3 on the other graphs reported in the dataset ranges from 10% on *com-amazon* up to 310.0% on *com-youtube*.

4.5 Betweenness Centrality on Weighted Graphs

Most graph algorithms can be expressed via matrix-vector or matrix-matrix products [KAB+16]. As an introductory example, we consider BFS [CLR90]. BFS starts at a root vertex r and traverses all nodes connected to r by one edge, then the set of nodes two edges away from r, etc. BFS can be used to compute shortest paths in an unweighted graph, which we can represent by

Graph	Total time (hour)	Mean time (sec)	Traditional Execution	degree-1	degree-2
MGBC-H0	21.8	0.071	1090920 (1090920)	0 (188317)	0 (77265)
MGBC-H1	18.0	0.070	902603 (1090920)	188317 (188317)	0 (77265)
MGBC-H2	20.8	0.068	1029219 (1090920)	0 (188317)	61701 (77265)
MGBC-H3	15.9	0.062	791294 (1090920)	188317 (188317)	165788 (111309)

Table 4.6: Impact of the heuristics on the exact Betweenness Computation on RoadNet-PA. The numbers in parenthesis represent the total number of vertices that may be computed by heuristics.

an adjacency matrix with elements $A_{ij} \in \{1, \infty\}$. In this case, BFS would visit each vertex v and derive its distance $\tau(r, v)$ from the root vertex r.

Algebraically, BFS can be expressed as iterative multiplication of the sparse adjacency matrix A with a sparse vector x_i over the tropical semiring, with i denoting the iteration number. The tropical semiring is a commutative monoid $(\mathbb{W}, \min)$ combined with the addition operator, which replace the monoid $(\mathbb{R}, +)$ and multiplication operator that are usually used within the matrix-vector product. The BFS algorithm would initialize $x_0^r = 0$ (an initial distance to r is 0) and any other element of x_0 is ∞ (i.e., an initial distance to any other element is ∞). Each BFS iteration computes $x_{i+1} = x_i \bullet_{\langle \min,+ \rangle} A$, then screens x_{i+1} retaining only elements that were ∞ in all x_j for $j < i$. The sparsity of the vector is given by all entries which are not equal to ∞, the additive identity of the tropical semiring.

Instead of using semirings, as commonly done in algebraic graph algorithms, we employ commutative monoids. Semirings permit multiplicative operators only on elements within the same set, while our algorithms require operators on members of different sets. We use monoids to define generalized matrix-vector and matrix-matrix multiplication operators. A monoid $(S, \oplus)$ is a set closed under an associative binary operation $\oplus$ with an identity element. A commutative monoid $(S, \oplus)$ is a monoid where $\oplus$ is commutative,

General BC		
	$\lambda(v)$	betweenness centrality of v
	$\tau(s,t)$	shortest path distance between s, t
	$\bar{\sigma}(s,t)$	number of shortest paths between s, t
	$\sigma(s,t,v)$	number of shortest paths between s, t leading via v
	$\delta(s,v)$	dependency of s on v; $\delta(s,v) = \sum_{t \in V} \sigma(s,t,v)/\bar{\sigma}(s,t)$
	$\pi(s,v)$	set of immediate predecessors of v in the shortest paths from s to v

Table 4.7: Symbols used in the section; $v, s, t \in V$ are vertices of a graph G

and

$$\bigoplus_{i=j}^{k} s(i) = s(j) \oplus s(j+1) \oplus \cdots \oplus s(k)$$

for any $k \geq j$ with $s(i) \in S$ for each $i \in j : k$. We denote the elementwise application of a monoid operator to a pair of matrices as $A \oplus B$ for any $A, B \in S^{m \times n}$. To define a suitable matrix multiplication primitive for our algorithms, we permit different domains for the matrices and replace elementwise multiplication with an arbitrary function that is a suitable map between the domains. Specifically, consider two input matrices $A \in D_A^{m \times k}$ and $B \in D_B^{k \times n}$, a bivariate function $f : D_A \times D_B \to D_C$, and a commutative monoid $(D_C, \oplus)$. Then, we denote matrix multiplication (MM) as $C = A \bullet_{\langle \oplus, f \rangle} B$, where each element of the output matrix, $C \in D_C^{m \times n}$, is $C(i,j) = \bigoplus_{k=1}^{n} f(A(i,k), B(k,j))$, $\forall i \in 1 : m, j \in 1 : n$. This MM notation enables a unified description of the main steps of our BC algorithm.

4.5.1 Notation

The variable we employ are summarized in Table 4.7. Notice that we use a different notation with respect to the previous sections just to provide more readable formulas in the lemmas we will show.

4.5.2 Related Work

The only distributed memory BC implementation done using matrix primitives we are aware of exists in CombBLAS [BG11]. To the best of our knowledge, we provide the first communication cost analysis of a BC algorithm, and the first implementation leveraging 3D sparse matrix multiplication. A mix of graph replication and blocking have been previously used for BC computation [BCV16], but the communication complexity of the scheme was not analyzed. Furthermore, previous parallel codes and algebraic BC formulations have been limited to unweighted graphs.

4.5.3 Maximal Frontier Algorithm

We now describe our algebraic maximal frontier BC algorithm for weighted graphs (MFBC), which

1. uses Bellman-Ford with multiplicity counting to obtain shortest distances for n_b starting vertices at a time,

2. propagates partial centrality scores from leaves of shortest path-trees to all other nodes,

3. outputs a vector λ of BC scores.

The operations dominating the cost of MFBC are expressed as generalized matrix multiplications of the adjacency matrix and sparse rectangular $n \times n_b$ matrices with elements of two specially-defined monoids, *multpaths* and *centpaths*. We first provide Maximal Frontier Bellman Ford (MFBF): an algorithm for computing distances and multiplicities. We then describe Maximal Frontier Brandes (MFBr): an algorithm for computing partial centrality factors. Finally, we combine these to obtain MFBC.

4.5.3.1 MFBF: Computing Shortest Paths

To determine the shortest distances and the multiplicities of each shortest path, we extend the Bellman-Ford algorithm. Rather than working purely with weights, we define the MFBF algorithm in terms of multpaths, which carry both weights and multiplicities. The maximal frontier approach propagates partial multiplicity counts through the graph, relaxing the edges of all nodes whose multiplicity was updated in the previous iteration.

The first key element of MFBF is the multpath monoid $(\mathbb{M}, \oplus)$. A multpath $x = (x.w, x.m) \in \mathbb{M} = \mathbb{W} \times \mathbb{N}$ is a path in G with a weight $p.w$ and a multiplicity $x.m$. The multpath monoid provides an operator that acts on any two multpaths x and y, and returns the one with lower weight; if the weights of x and y are equal, then a multpath with the same weight and the sum of multiplicities is returned. Specifically, for any pair $x, y \in \mathbb{M}$, we have

$$x \oplus y = \begin{cases} x & : x.w < y.w \\ y & : x.w > y.w \\ (x.w, x.m + y.m) & : x.w = y.w. \end{cases}$$

Our MFBF algorithm (Algorithm 12) iteratively updates a matrix T of multpaths via forward traversals from each source vertex. This is done in the inner loop (lines 3-7) where the $\mathcal{T}$ tensor with partial multiplicity scores is updated in each iteration using Bellman-Ford Action $\bullet_{\langle \oplus, f \rangle}$, where f is defined as

$$f : \mathbb{M} \times \mathbb{W} \to \mathbb{M}, \qquad f(a, w) = (a.w + w, a.m).$$

The function f may be interpreted as an action of the monoid $(\mathbb{W}, +)$ on the set $\mathbb{M}$. This concept generalizes to $n \times k$ matrices, where we have a monoid action $\bullet_{\langle \oplus, f \rangle}$ with monoid $(\mathbb{W}^{n \times n}, \bullet_{\langle \min, + \rangle})$ on the set $\mathbb{M}^{n \times k}$.

We obtain shortest path distances and multiplicities via MFBF (Algorithm 12): a Bellman-Ford variant that relaxes all edges adjacent to vertices whose path information changed in the previous iteration. The edge re-

laxation is done via matrix multiplication of the adjacency matrix and a multpath matrix, which appends edges to the existing frontier of vertices via function f, then uses the multpath operator $\oplus$ to select the minimum distance new set of paths, along with the number of such new paths. This partial multiplicity score is subsequently accumulated to T if it corresponds to a minimum distance path from the starting vertex.

Algorithm 12 $[T] = \text{MFBF}(A, \vec{s})$

Input A: $n \times n$ adjacency matrix, $\vec{s}$: list of starting n_b vertices
Output T: multpath matrix of distances and multiplicities from vertices $\vec{s}$ ◃ Existential qualifiers $\forall s \in 1 : n_b$ (denoting starting vertices) and $\forall v \in 1 : n$ (denoting destination vertices) are implicit.

1: $T(s, v) := (A(\vec{s}(s), v), 1)$ ◃ Initialize multpaths
2: $\mathcal{T} := T$ ◃ Initialize multpath frontier
3: **while** $\mathcal{T}(s, v) \neq (\infty, 0)$ for some s, v **do**
4: $\quad \mathcal{T} := \mathcal{T} \bullet_{\langle \oplus, f \rangle} A$ ◃ Explore nodes adjacent to frontier
5: $\quad T := T \oplus \mathcal{T}$ ◃ Accumulate multiplicities
6: $\quad$ **if** $\mathcal{T}(s, v).m{=}0 \vee \mathcal{T}(s, v).w{>}T(s, v).w$ ◃ Determine new frontier based on multiplicity updates **then** $\mathcal{T}(s, v){:=}(\infty, 0)$
7: **end while**

Lemma 4.5.1 *For any adjacency matrix A and vertex set $\vec{s}$, $T = MFBF(A, \vec{s})$ satisfies $T(s, v) = (\tau(s, v), \bar{\sigma}(s, v))$.*

Proof 4.5.1 *Let the maximum number of edges in any shortest path from node s be d. For $j \in 1 : d, v \in V \setminus \{s\}$, let each $h_j(s, v) \in \mathbb{M}$ be a multpath corresponding to the weight and multiplicity of all shortest paths from vertex s to vertex v containing up to j edges (if there are no such paths, $h_j(s, v) = (\infty, 0)$). Further, let each $\hat{h}_j(s, v) \in \mathbb{M}$ be a multpath corresponding to the weight and multiplicity of all shortest paths from vertex s to vertex v containing exactly j edges (if there are no such paths, $\hat{h}_j(s, v) = (\infty, 0)$). Note that $h_d(s, v)$ contains the weight and multiplicity of all shortest paths from vertex s to vertex v, since no shortest path can contain more than d edges, therefore $h_d(s, v) = (\tau(s, v), \bar{\sigma}(s, v))$. Further, by the definition of $\oplus$, we have $h_j(s, v) = \bigoplus_{q=1}^{j} \hat{h}_j(s, v)$.*

Let $T_j(s, v)$ be the state of $T(s, v)$ after the completion of $j - 1$ iterations of the loop from line 3. We show by induction on j that $T_j(s, v) = h_j(s, v)$

and $\mathcal{T}_j(s,v) = \hat{h}_j(s,v)$*, and subsequently that after* $d-1$ *while loop iterations,* $T(s,v) = T_d(s,v) = h_d(s,v)$*. For* $j = 1$*, no iterations have completed and we have* $T(s,v) = (A(s,v), 1)$*, as desired. For the inductive step, we show that given* $T_j(s,v) = h_j(s,v)$ *one iteration of the loop on line 3 will yield* $T_{j+1}(s,v) = h_{j+1}(v)$*. We note that by definition of* $\oplus$*, only paths with a minimal weight* $\mathcal{T}_j(s,u).w$ *contribute to* $\mathcal{T}_{j+1}(s,v)$*, and furthermore (again by definition of* $\oplus$*),*

$$\mathcal{T}_{j+1}(s,v).m = \sum_{w \in P} w.m, \quad \text{where}$$
$$P = \{\mathcal{T}_j(s,u) : \mathcal{T}_j(s,u).w + A(u,v) = \mathcal{T}_{j+1}(s,v).w\},$$

i.e., $\mathcal{T}_{j+1}(s,v).m$ *is the sum of the multiplicities of all the minimal weight paths from vertex* s *to* v *consisting of* $j+1$ *edges. Our expression for* P *is valid, since each must consist of a minimal weight path of* k *edges from vertex* s *to some vertex* u*, which is given by* $\mathcal{T}_j(s,u)$ *and some other edge* (u,v)*, whose weight is given by* $A(u,v)$*.*

4.5.3.2 MFBr: Computing Centrality Scores

Once we have obtained the distances and multiplicities of shortest paths from a set of starting vertices via MFBF, we can begin computing the partial centrality scores. However, rather than working with partial centrality scores $\delta(s,v)$ we work with partial centrality factors:

$$\zeta(s,v) = \delta(s,v)/\bar{\sigma}(s,v) = \sum_{w \in \pi(s,v)} \Big(\frac{1}{\bar{\sigma}(s,w)} + \zeta(s,w)\Big).$$

Computing ζ rather than δ simplifies the algebraic steps done by the algorithm and leads to a simpler proof of correctness. Once we have computed ζ, it is cheap to construct δ, simply by multiplying by corresponding elements of $\bar{\sigma}$, which we have already computed via MFBF.

To propagate partial centrality scores, we use centpaths, which store a distance, a contribution to the centrality score, and a counter. A centpath

$x = (x.w, x.p, x.c) \in \mathbb{C} = \mathbb{W} \times \mathbb{R} \times \mathbb{Z}$ is a path with a weight $x.w \in \mathbb{W}$, partial centrality score $x.p \in \mathbb{R}$, and a counter $x.c \in \mathbb{Z}$. Our algorithm will converge to a centpath x for each pair of starting and destination nodes s, v, where the partial dependency factor $x.p = \zeta(s, v) = x.p$.

The counter $x.c$ is used to keep track of the number of predecessors who have not propagated a partial dependency factor up to the node v in a previous iteration. The counter is decremented until reaching zero, at which point the final centrality scores of all predecessors have been integrated into $x.p$ and it is then propagated from v up to the root s.

Similarly to the multpath monoid, we define a centpath monoid $(\mathbb{C}, \otimes)$ with an operator that acts on any two centpaths x and y, and returns the one with lower weight; if the weights of x and y are equal, then the partial centrality factors and counter values of the two centpaths are added. Specifically, for any pair $x, y \in \mathbb{C}$, we have

$$x \otimes y = \begin{cases} x & ; x.w > y.w \\ y & ; x.w < y.w \\ (x.w, x.p + y.p, x.c + y.c) & ; x.w = y.w. \end{cases}$$

Our MFBr algorithm (Algorithm 13) iteratively updates a matrix Z of centpaths via backward propagation of partial centrality factors from the leaves of the shortest path tree. In the inner loop (lines 5-12), Z is computed with $\bullet_{\langle \otimes, g \rangle}$, where g is defined as

$$g : \mathbb{C} \times \mathbb{W} \to \mathbb{C}, \qquad g(a, w) = (a.w - w, a.p, a.c).$$

The function g may be interpreted as an action of the monoid $(\mathbb{W}, +)$ on the set $\mathbb{C}$.

For weighted graphs, a single vertex may appear many times in the frontier as its shortest path information and multiplicity is corrected, unlike in a BFS or Dijkstra's algorithm-based traversal, where the total number of nonzeros in the matrix multiplication operand $\mathcal{T}$ sums to $(n-1)n_b$ over all iterations. For the Brandes step, we can avoid propagating unfinalized

Algorithm 13 $[Z] = \text{MFBr}(A, T)$

Input A: $n \times n$ adjacency matrix, T: matrix of distances and multiplicities
Output Z: centpath matrix of partial centrality factors ζ ◃ Existential qualifiers $\forall s \in 1 : n_b$ (denoting starting vertices) are implicit and $\forall v \in 1 : n$ (denoting intermediate vertices). ◃ Initialize centpaths by finding counting predecessors

1: $\mathcal{Z}(s, v) := (T(s, v).w, 0, 1)$
2: $Z := \mathcal{Z} \otimes (\mathcal{Z} \bullet_{\langle \otimes, g \rangle} A^{\mathsf{T}})$ ◃ Initialize centpath frontier
3: **if** $Z(s, v).c = 0$ **then** $\mathcal{Z}(s, v) := (T(s, v).w, 1/T(s, v).m, -1)$
4: **else** $\mathcal{Z}(s, v) = (\infty, 0, 0)$
5: **while** $\mathcal{Z}(s, v) \neq (\infty, 0, 0)$ for some s, v **do**
6: $\quad \mathcal{Z} := \mathcal{Z} \bullet_{\langle \otimes, g \rangle} A^{\mathsf{T}}$ ◃ Back-propagate frontier of centralities ◃ Turn off counters for nodes that already appeared in a frontier
7: $\quad$ **if** $Z(s, v).c = 0$ **then** $Z(s, v).c = -1$
8: $\quad Z := Z \otimes \mathcal{Z}$ ◃ Accumulate centralities and increment counters ◃ Determine new frontier based on counters
9: $\quad$ **if** $Z(s, v).c = 0$ **then**
10: $\quad\quad \mathcal{Z}(s, v) := (T(s, v).w, Z(s, v).p + 1/T(s, v).m, -1)$
11: $\quad$ **else** $\mathcal{Z}(s, v) = (\infty, 0, 0)$
12: **end while**

information as we already know the structure of the shortest path trees.

MFBr (Algorithm 13) propagates centrality factors optimally via the counter kept by each centpath, putting vertices in the frontier only when all of their predecessors have already appeared in previous frontiers. The counter is initialized to the number of predecessors, is decremented until reaching 0, added to a frontier and set to -1 to avoid re-adding the vertex to another frontier. This approach is strictly better than propagating partial centrality scores, which does not contribute to overall progress. Simultaneously, this scheme is much faster than doing Dijkstra to compute shortest-paths, since it requires the same number of iterations as Bellman Ford (Dijkstra's algorithm requires $n - 1$ matrix multiplications).

Lemma 4.5.2 *For any adjacency matrix A and a multpath matrix T containing shortest path distances and multiplicities, $Z = MFBr(A, T)$ satisfies $Z(s, v).p = \zeta(s, v)$.*

Proof 4.5.2 *We prove that the partial BC scores are computed correctly after $d - 1$ iterations of the loop in line 5 if all shortest paths from $\vec{s}$ in G consist of at most d edges. As before, we denote the shortest distance from node s*

to v as $\tau(s,v)$ and the multiplicity as $\bar{\omega}(s,v)$. We define $k_j(s,v) \in \mathcal{Z}$ as the sum of all minimal distance paths of at most $j-1$ edges from s ending at u that are on the minimal distance path between 1 and v,

$$\begin{aligned} k_j(s,v) &= \sum_{(u,\star)\in P_j(s,v)} \frac{1}{\bar{\sigma}(s,u)}, \quad \textit{where} \\ P_j(s,v) &= \{(u,\vec{w})|l \in 1:j-1, \vec{w} \in (1:n)^l, \\ &\ \tau(s,u) + A(u,w_1) + A(w_1,w_2) + \ldots + A(w_l,v) = \tau(s,v)\} \end{aligned}$$

Since $P_d(s,v)$ is the set of all shortest paths between s and v that are parts of shortest paths between s and u, for each u there are $\sigma(s,u,v)/\bar{\sigma}(u,v)$ such paths, and therefore,

$$k_d(s,v) = \sum_{(u,\star)\in P_d(s,v)} \frac{1}{\bar{\sigma}(s,u)} = \sum_{u=1}^{n} \frac{\sigma(s,u,v)}{\bar{\sigma}(s,u)\bar{\sigma}(u,v)}.$$

We now show that $k_j(s,v)$ can be expressed in terms of $k_{j-1}(s,u)$ for all $u \in P_1(v)$ (the 1-edge shortest-path neighborhood of v from s). We accomplish this by disjointly partitioning $P_j(s,v)$ into $P_1(s,v)$ and $\bigcup_{u\in P_1(s,v)} P_{j-1}(s,u)$, which yields,

$$\begin{aligned} k_j(s,v) &= \sum_{(u,\star)\in P_1(s,v)} \left(\frac{1}{\bar{\sigma}(s,u)} + \sum_{(w,\star)\in P_{j-1}(s,u)} \frac{1}{\bar{\sigma}(s,w)} \right) \\ &= \sum_{(u,\star)\in P_1(s,v)} \left(\frac{1}{\bar{\sigma}(s,u)} + k_{j-1}(s,u) \right) \end{aligned}$$

Let $Z_j(s,v)$ be the state of $Z(s,v)$ and $\mathcal{Z}_j(s,v)$ be the state of $\mathcal{Z}(s,v)$ after the completion of $j-1$ iterations of the loop on line 5. We argue by induction on j, that for all $j \in 1:d$, $Z_j(s,v).p = k_j(s,v) = k_d(s,v)$ and

$$\mathcal{Z}_j(s,v) = (\tau(s,v), 1/\bar{\sigma}(s,v) + \zeta(s,v), -1)$$

if and only if the largest number of edges in any shortest path from any node u to v, such that $\tau(s,v) = \tau(s,u) + \tau(u,v)$, is $j-1$. In the base case, $j=1$

Algorithm 14 $[\lambda] = \text{MFBC}(A)$

Input A: $n \times n$ adjacency matrix, n_b: the batch size
Output λ: a vector of BC scores

1: $\forall v \in 1:n, \quad \lambda(v) := 0$ ◃ Initialize the BC scores
2: **for** $i \in 1 : n/n_b$ **do**
3: $[T] = \text{MFBF}(A, (i-1)n_b + 1 : in_b)$
4: $[Z] = \text{MFBr}(A, T)$ ◃ Accumulate partial centralities: $\delta(s, v) = \zeta(s, v) \cdot \bar{\sigma}(s, v)$
5: $\forall v \in 1:n, \quad \lambda(v) := \lambda(v) + \sum_{s=1}^{n_b} Z(s, v).p \cdot T(s, v).m$
6: **end for**

and we have $Z_1(s, v).p = k_j(s, v) = k_d(s, v) = \zeta(s, v) = 0$ for all vertices v that have no predecessors (are leaves in the shortest path tree), further these vertices are set appropriately in $\mathcal{Z}_1$.

For the inductive step, we note that the update on line 5 is contributing the appropriate factor of $\frac{1}{\bar{\sigma}(s,u)} + k_{j-1}(u)$ from each predecessor vertex u. Furthermore, each such predecessor vertex u must have been a member of a single frontier by iteration j, since the larger number of edges in any shortest path from u to any node v must be no greater than $j - 1$. Therefore, the counter $Z_j(s, v).c = 0$, which means $\mathcal{Z}_j(s, v)$ is set appropriately (for subsequent iterations $k > j$, $\mathcal{Z}_k(s, v) = (\infty, 0, 0)$ since we set $Z_j(s, v).c = -1$ at iteration $j + 1$).

4.5.3.3 Combined BC Algorithm

To obtain a complete algorithm for BC, we combine MFBF and MFBr in MFBC (Algorithm 14). MFBC is parametrized with a batch size n_b and proceeds by computing MFBF and MFBr to obtain partial centrality factors for n_b starting vertices at a time. These factors are then appropriately scaled by multiplicities ($\bar{\sigma}(s, v)$) and accumulated into a vector of total centrality scores.

Theorem 4.5.3 *For any adjacency matrix A and $n_b \in 1 : n$, $\lambda = MFBC(A, n_b)$ satisfies $\lambda(v) = \sum_{s,t \in V} \frac{\sigma(s,t,v)}{\bar{\sigma}(s,t)}$.*

Proof 4.5.3 *We assume $n \mod n_b = 0$, if it does not hold then $n \mod n_b$ disconnected vertices can be added to G without changing λ. For each batch of*

vertices, MFBF computes the correct shortest distances and multiplicities T by Lemma 4.5.1. For each T, MFBr computes the correct partial centrality scores Z by Lemma 4.5.2. Therefore, at iteration i, we have $T(s,v).m = \bar{\sigma}((i-1)n_b + s, v)$ and $Z(s,v).p = \zeta((i-1)n_b + s, v)$. Furthermore, over all iterations, line 5 extrapolates to

$$\begin{aligned}\lambda(v) &= \sum_{s=1}^{n} Z(s,v).p \cdot T(s,v).m = \sum_{s \in V} \zeta(s,v) \cdot \bar{\sigma}(s,v) \\ &= \sum_{s \in V} \delta(s,v) = \sum_{s \in V} \sum_{t \in V} \frac{\sigma(s,t,v)}{\bar{\sigma}(s,t)}.\end{aligned}$$

4.5.4 Communication Complexity

Our maximal frontier betweenness centrality scheme is designed to leverage all available parallelism in the problem to accelerate overall progress. We now formally study the scalability of the algorithm by bounding its communication complexity. We consider a parallel execution model with p processors and count the number of messages and amount of data communicated by any processor. We do not keep track of the number of operations performed, because for sparse matrix multiplication, all algorithms we consider have an optimal computation cost, and for betweenness centrality our algorithm is work-optimal in the unweighted case. The computation cost in the weighted case depends on the number of times each vertex appears in a frontier during the execution of MFBF, which is dependent on the connectivity of the graph as well as the edge weights.

First, we derive the communication costs for sparse matrix multiplication, by far the most expensive operation performed within MFBC. We present a communication cost model for the matrix multiplication (tensor contraction) routine in CTF, for sparse matrices with arbitrary dimensions and nonzero count. CTF employs a larger space of sparse matrix-multiplication variants than considered in any previous study. Our results provide a communication bound that is substantially lower than previous results for sparse matrix

multiplication when the number of nonzeros is imbalanced between matrices. This theoretical result is of stand-alone importance, as sparse matrix multiplication is a critical primitive not only in graph algorithms, but in numerical algorithms such as multi-grid.

We then express the communication cost of MFBC in terms of the communication cost of the sparse matrix multiplications it execute. Overall, the MFBC algorithm attains a communication bandwidth cost that is the same or better than the communication cost of the best-known all-pairs shortest-path algorithms, which are much less memory-efficient. However, the latency (synchronization) cost of MFBC may be higher, by a factor proportional to the number of batches used in MFBC, which in turn depends on the available memory.

4.5.4.1 Cost Model

We use the $\alpha - \beta$ model [SCKD14] where the latency of sending a message is α and the inverse bandwidth is β. We assume that $\alpha \geq \beta$. There are p processes and M (number of words) is the size of a local memory at every processor.

The cost of collective communication routines (scatter, gather, broadcast, reduction, and allreduction) on p processors in the $\alpha - \beta$ model is $O(\beta \cdot x + \alpha \cdot \log p)$ [ABB$^+$15]; x is the maximum number of words that each processor owns at the start or end of the collective. Furthermore, the cost of a *sparse reduction* where each processor inputs a sparse array and the resulting array has x nonzeros is also $O(\beta \cdot x + \alpha \cdot \log p)$.

We use $\text{nnz}(X)$ to denote the number of non-zeros in any matrix X and $\text{flops}(X, Y)$ to denote the number of nonzero products when multiplying sparse matrices X and Y.

4.5.5 Parallel Sparse Matrix Multiplication

We first analyze the product of sparse matrices $A^{m\times k}$ and $B^{k\times n}$ that produce matrix $C^{m\times n}$. We include algorithms that use 1-, 2-, and 3-dimensional matrix decomposition. All considered algorithms have a computation cost of $O(\text{flops}(A,B)/p)$, which we omit.

All the considered algorithms and implementations use matrix blocks that correspond to the cross product of a subset of columns and a subset of rows of the matrices. These blocks are chosen obliviously of the structure of matrix. For sparse matrices with a sufficiently large number of nonzeros, randomizing the row and column order implies that the number of nonzeros of each such block is proportional to the size of the block. Therefore, we assume that the number of nonzeros in any block of dimensions $b_1 \times b_2$ of a sparse matrix $A^{m\times k}$ has $O(\text{nnz}(A)b_1b_2/(mk))$ nonzeros so long as $mk/(b_1b_2) \leq p$.

Further, we assume that multiplication of any two blocks of equal dimensions yields about the same number of nonzero products and output nonzero. Thus, when multiplying blocks of size $b_1 \times b_2$ of matrix A and $b_2 \times b_3$ of matrix B, the number of nonzero operations is $O(\text{flops}(A,B)b_1b_2b_3/(mnk))$ and the number of nonzeros in the output block contribution to matrix C is

$$O\bigg(\min\Big[\frac{\text{nnz}(C)}{mn}b_1b_3, \frac{\text{flops}(A,B)}{mnk}b_1b_2b_3\Big]\bigg).$$

For a sparse matrix corresponding to uniform random graphs, the respective numbers are

$$\text{flops}(A,B) \approx \frac{\text{nnz}(A)}{mk}\frac{\text{nnz}(B)}{kn}mnk = \text{nnz}(A)\,\text{nnz}(B)/k$$

and $\text{nnz}(C) \approx \min(mn, \text{flops}(A,B))$.

4.5.5.1 1D and 2D Algorithms

We first recall the communication analysis of the graphs decomposition strategies described in 2.2. There are three variants of 1D decomposition for MM operation, each of which replicates one of the matrices and blocks the others

into p pieces. Variant A replicates A via broadcast and assigns each processor a set of columns of B and C. Variant B broadcasts B and assigns each processor a set of rows of A and C. Variant C assigns each processor a set of columns of A and rows of B, computes their product, and uses a reduce to obtain C. The communication cost of version X of 1D algorithm for $\mathrm{X} \in \{\mathrm{A}, \mathrm{B}, \mathrm{C}\}$ is

$$W_{\mathrm{X}}(X, p) = O(\alpha \cdot \log p + \beta \cdot \mathrm{nnz}(X)).$$

2D algorithms [Can69, VDGW97, ABG+95] block all matrices on a processor grid of $p_r \times p_c$ processors and move the data in steps to perform the matrix multiplication. 2D algorithms can be based on point-to-point or collective communication. The former are up to $O(\log p)$ faster in latency, but the latter generalize easier to rectangular processor grids. The algorithms are naturally extended to handle sparse matrices, by treating the matrix blocks as sparse [BG12,BBD+13]. One of the simplest 2D algorithms is Cannon's algorithm, which shifts blocks of A and B on a square processor grid, achieving a communication cost of

$$O\left(\alpha \cdot \sqrt{p} + \beta \cdot \frac{\mathrm{nnz}(A) + \mathrm{nnz}(B)}{\sqrt{p}}\right).$$

The algorithm is optimal for square matrices, but other variants achieve lower communication cost when the number of nonzeros in the two operand matrices are different.

Our implementation leverages three variants of 2D algorithms using broadcasts and (sparse) reductions. The variant AB broadcasts blocks of A and B along processor grid rows and columns, while the variants AC and BC involve reducing C and broadcasting A and B, respectively. CTF uses $\mathrm{lcm}(p_r, p_c)$ (where lcm is the least common multiple) broadcasts/reductions and adjusts p_r and p_c so that $\mathrm{lcm}(p_r, p_c) \approx \max(p_r, p_c)$ steps of collective communication are performed. When each matrix block is sparse with the specified load balance assumptions, the costs achieved by these 2D algorithms are given in

general by W_{YZ} for variants $\text{YZ} \in \{\text{AB}, \text{AC}, \text{BC}\}$ as $W_{\text{YZ}}(Y, Z, p_r, p_c) =$

$$O\bigg(\alpha \cdot \max(p_r, p_c)\log(p) + \beta \cdot \Big(\frac{\text{nnz}(Y)}{p_r} + \frac{\text{nnz}(Z)}{p_c}\Big)\bigg)$$

4.5.5.2 3D Algorithms

While 2D algorithms are natural from a matrix-distribution perspective, the dimensionality of the computation suggests the use of 3D decompositions [DNS81,ABG⁺95,ACS90,Ber89,MT99], where each processor computes a subvolume of the mnk dense products. 3D algorithms have been adapted to sparse matrices, in particular by the Split-3D-SpGEMM scheme [ABB⁺15], which obtains a cost of

$$O\left(\alpha \cdot \sqrt{cp}\log p + \beta \cdot \left(\frac{\text{nnz}(A) + \text{nnz}(B)}{\sqrt{cp}} + \frac{\text{flops}(A, B)}{p}\right)\right)$$

by using a the grid of processes that is $\sqrt{p/c} \times \sqrt{p/c} \times c$.

We derive 3D algorithms (and implement within CTF) by nesting the three 1D algorithm variants with the three 2D algorithm variants. The cost of the resulting nine 3D variants on a $p_1 \times p_2 \times p_3$ processor grid with the 1D algorithm applied over the first dimension and the is

$$\begin{aligned} W_{\text{X,YZ}}(X, Y, Z, p_1, p_2, p_3) &= W_{\text{X}}(X[p_2, p_3]) \\ &+ \begin{cases} W_{\text{YZ}}(Y, Z[p_1], p_2, p_3) & : X = Y, \\ W_{\text{YZ}}(Y[p_1], Z, p_2, p_3) & : X = Z, \\ W_{\text{YZ}}(Y[p_1], Z[p_1], p_2, p_3) & : X \notin \{Y, Z\}, \end{cases} \end{aligned}$$

for $(\text{X}, \text{YZ}) \in \{\text{A}, \text{B}, \text{C}\} \times \{\text{AB}, \text{AC}, \text{BC}\}$, with notation $X[p_2, p_3]$ denoting that the 1D algorithm operates on blocks of X given from a $p_2 \times p_3$ distribution, while $Y[p_1]$ and $Z[p_1]$ refer to 1D distributions. This cost simplifies

to

$$W_{\mathrm{X,YZ}}(X,Y,Z,p_1,p_2,p_3) =$$
$$O\bigg(\alpha\cdot\max(p_1,p_2)\log(\min(p_1,p_2)) + \beta\cdot\frac{\mathrm{nnz}(X)}{p_2p_3}\bigg)$$
$$+\begin{cases} O\Big(\beta\cdot(\frac{\mathrm{nnz}(X)}{p_2}+\frac{\mathrm{nnz}(Z)}{p_1p_3})\Big) & : X = Y,\\ O\Big(\beta\cdot(\frac{\mathrm{nnz}(Y)}{p_1p_2}+\frac{\mathrm{nnz}(X)}{p_3})\Big) & : X = Z,\\ O\Big(\beta\cdot(\frac{\mathrm{nnz}(Y)}{p_1p_2}+\frac{\mathrm{nnz}(Z)}{p_2p_3})\Big) & : X \notin \{Y,Z\}.\end{cases}$$

The amount of memory used by this algorithm is

$$M_{\mathrm{X,YZ}}(X,Y,Z,p,p_1) = O\bigg(\frac{\mathrm{nnz}(X)p_1}{p} + \frac{\mathrm{nnz}(Y)+\mathrm{nnz}(Z)}{p}\bigg).$$

As we additionally consider pure 1D and 2D algorithms, then pick the 1D, 2D, or 3D variant of least cost. Provided unlimited memory, the execution time of our sparse matrix multiplication scheme is no greater than

$$W_{\mathrm{MM}}(A,B,C,p) = O\bigg(\min_{\substack{p_1,p_2,p_3\in\mathbb{N}\\ p_1p_2p_3=p}}\bigg[\alpha\cdot\max(p_1,p_2,p_3)\log p$$
$$+\beta\cdot\Big(\frac{\mathrm{nnz}(A)}{p_1p_2}\delta(p_3) + \frac{\mathrm{nnz}(B)}{p_2p_3}\delta(p_1) + \frac{\mathrm{nnz}(C)}{p_1p_3}\delta(p_2)\Big)\bigg]\bigg),$$

where $\delta(x) = 1$ when $x \geq 1$ and $\delta(1) = 0$.

4.5.5.3 Parallel Betweenness Centrality

Provided the communication analysis of sparse matrix multiplication, it is easy to ascertain a communication cost bound for MFBF, which performs the bulk of the computation via generalized matrix multiplication. We focus our cost analysis on unweighted graphs, as it is difficult to ascertain bounds on the size of each frontier in weighted graphs. For unweighted graphs, each vertex appears in a unique frontier.

Theorem 4.5.4 *For any unweighted n-node m-edge graph G with adjacency matrix A and diameter d, Given a machine with p processors each with $M = \Omega(cm/p)$ words of memory for any $c \in [1, p]$, MFBC (Algorithm 14) can execute with communication cost,*

$$W_{MFBC}(n,m,p,c) = O\bigg(\alpha \cdot \frac{dn^2}{m}\sqrt{p/c^3}\log p + \beta \cdot \Big(\frac{n^2}{\sqrt{cp}} + \frac{n\sqrt{m}}{p^{2/3}}\Big)\bigg).$$

Proof 4.5.4 *MFBC is dominated in computation and communication cost by multiplications of sparse matrices, triggered by the operator $\bullet_{\langle\oplus,f\rangle}$ within MFBF and $\bullet_{\langle\otimes,g\rangle}$ within MFBr. There are up to $2d+1$ such matrix multiplications in total. Without loss of generality, we consider only the d within MFBF, letting F_i be the frontier ($\mathcal{T}$) at iteration i and G_i be the output of $\mathcal{T} \bullet_{\langle\oplus,f\rangle} A$, which include F_{i+1} but can be much denser. We can then bound the cost of MFBC as*

$$\begin{aligned}W_{MFBC} &= O\bigg(\min_{n_b\in 1:n}\Big[\frac{n}{n_b}\sum_{i=1}^{d} W_{MM}(A, F_i, G_i, p)\Big]\bigg)\\ &= O\bigg(\min_{n_b\in 1:n}\frac{n}{n_b}\sum_{i=1}^{d}\min_{\substack{p_1,p_2,p_3\in\mathbb{N}\\ p_1p_2p_3=p}}\Big[\alpha\cdot\max(p_1,p_2,p_3)\log p\\ &\quad + \beta\cdot\Big(\frac{m}{p_1p_2}\delta(p_3) + \frac{\text{nnz}(F_i)}{p_2p_3}\delta(p_1) + \frac{\text{nnz}(G_i)}{p_1p_3}\delta(p_2)\Big)\Big]\bigg).\end{aligned}$$

The MFBC algorithm requires $O(nn_b/p)$ memory to store T, therefore, we have $nn_b/p = O(M)$, and select $n_b = cm/n$,

$$\begin{aligned}W_{MFBC} = O\bigg(\frac{n^2}{cm}\sum_{i=1}^{d}\min_{\substack{p_1,p_2,p_3\in\mathbb{N}\\ p_1p_2p_3=p}}\Big[\alpha\cdot\max(p_1,p_2,p_3)\log p\\ + \beta\cdot\Big(\frac{m}{p_1p_2}\delta(p_3) + \frac{\text{nnz}(F_i)}{p_2p_3}\delta(p_1) + \frac{\text{nnz}(G_i)}{p_1p_3}\delta(p_2)\Big)\Big]\bigg).\end{aligned}$$

We will use a 3D algorithm with $p_1 = p_2 = \sqrt{p/c}$, $p_3 = c$, which replicates A via a 1D algorithm, then employs the BC variant of a 2D algorithm, with

a memory usage of $O(cm/p)$. The replication of A can be amortized over (up to d) sparse matrix multiplications and over the $\frac{n^2}{cm}$ batches, since A is always the same adjacency matrix. Thus,

$$\begin{aligned} W_{MFBC} =& O\bigg(\beta \cdot \frac{cm}{p} + \frac{n^2}{cm}\bigg(\Big(\sum_{i=1}^{d}\Big[\alpha \cdot \sqrt{p/c}\log p \\ &+ \beta \cdot \Big(\frac{\text{nnz}(F_i)}{\sqrt{pc}} + \frac{\text{nnz}(G_i)}{\sqrt{pc}}\Big)\Big]\Big)\bigg)\bigg), \end{aligned}$$

and furthermore, over all n_b batches the total cost is

$$\begin{aligned} W_{MFBC} =& O\bigg(\alpha \cdot \frac{dn^2}{m}\sqrt{p/c^3}\log p \\ &+ \beta \cdot \bigg[\frac{cm}{p} + \sum_{i=1}^{d}\frac{n^2(\text{nnz}(F_i) + \text{nnz}(G_i))}{m\sqrt{pc^3}}\bigg]\bigg). \end{aligned}$$

Now, since the graph is unweighted, we know that each vertex appears in a unique frontier, so $\sum_{i-1}^{d}\text{nnz}(F_i) \leq nn_b = cm$. Therefore, each node can be reached from 3 frontiers (the one it is a part of, the previous one, and the subsequent one), therefore $\sum_{i-1}^{d}\text{nnz}(G_i) \leq 3cm$. Then, the total bandwidth cost over all d iterations and cm/n batches is $O(\beta \cdot (n^2/\sqrt{cp} + \frac{cm}{p}))$. This cost is minimized for $c = p^{1/3}n^2/m$, so with $M = \Omega(n^2/p^{2/3})$ memory, the cost $O(\beta \cdot n\sqrt{m}/p^{2/3})$ can be achieved.

Our communication cost analysis can be extended to weighted graphs, provided bounds on $\sum_i \text{nnz}(F_i)$ and $\sum_i \text{nnz}(G_i)$ for each batch. The quantity $\sum_i \text{nnz}(F_i)$ can be bounded given an amplification factor bounding the number of Bellman-Ford iterations in which the shortest path distance between any given pair of source/distance vertices is changed. However, we do not see a clear way to bound $\sum_i \text{nnz}(G_i)$ for weighted graphs. We evaluate MFBC for weighted graphs in the subsequent section, observing a slowdown proportional to the factor of increase in the number of iterations with respect to the unweighted case (in the unweighted case it is the diameter d).

We are not aware of other communication cost studies of betweenness centrality algorithms, but we can compare our approach to those computing the full distance matrix via all-pairs shortest-paths (APSP) algorithms, requiring at least n^2/p memory, regardless of m. The best-known algorithms for the APSP problem, leverage 3D matrix multiplication to obtain a bandwidth cost of $O(\beta \cdot n^2/\sqrt{cp})$ using $O(cn^2/p)$ memory for any $c \in [1, p^{1/3}]$ [Tis01]. MFBC matches this bandwidth cost, while using only $O(cm/p)$ memory. Further, given sufficient memory $M = \Omega(n^2/p^{2/3})$, our algorithm is up to $\min(n/\sqrt{m}, p^{2/3})$ faster. When also considering an algorithm that replicates the graph as an alternative, the best speed-up achievable by MFBC is for $M = \Theta(n^2/p^{2/3})$ memory with $n/\sqrt{m} = p^{1/3}$, and when $\beta \gg \alpha$, in which case $W_{\text{MFBC}}(n, n^2/p^{2/3}, p, p^{2/3}) = O(\beta \cdot n^2/p)$ is $p^{1/3}$ times faster than Floyd-Warshall, path doubling, or Dijkstra with a replicated graph.

The capability of our algorithm to employ large replication factors c gives it good strong scalability properties. If each processor has $M = O(m/p_0)$ memory, it is possible to achieve perfect strong scalability in bandwidth cost using up to $p_0^{3/2} n^3/m^{3/2}$ processors, while for up to $p_0^{3/2} n^2/m$,

$$W_{\text{MFBC}}(n, m, cp_0, c) = \frac{1}{c} W_{\text{MFBC}}(n, m, p_0, 1)$$

is satisfied, so strong scalability is achieved in all costs from p_0 to $p_0^{3/2} n^2/m$ processors. This range in strong scalability is be better than that achieved by the best known square dense matrix multiplication algorithms, p_0 to $p_0^{3/2}$ [SD11].

The Floyd-Warshall APSP algorithm has latency cost $O(\alpha \cdot \sqrt{cp})$, but a path-doubling scheme can achieve $O(\alpha \cdot \log p)$ [Tis01] using $O(n^2/p^{2/3})$ memory. Given this amount of memory, MFBC can achieve a latency cost of

$$O\bigg(\alpha \cdot d \log p \Big(\frac{n^2}{\sqrt{p}m} + \sqrt{n/\sqrt{m}}\Big)\bigg).$$

It might be possible to improve this latency cost by using different sparse matrix multiplication algorithms.

4.5.6 Implementation

We implement two parallel versions of MFBC using CTF. The first, CTF-MFBC, uses CTF to dynamically select data layouts without guidance from the developer. The second, CA-MFBC, predefines the 3D processor grid layout that we used to minimize theoretical communication cost in the proof of Theorem 4.5.4. We first summarize the functionality of CTF and explain how it provides the sparse matrix operations necessary for MFBC. We then give more details on how CTF handles data distribution and communication.

4.5.6.1 From Algebra to Code

CTF permits definition of all well-known algebraic structures and implements tensor contractions with user-defined addition and multiplication operators [SH15]. Matrices are sufficient to encode graphs and subgraphs (frontiers); tensors of order higher than two can represent hypergraphs. Since graphs are sufficient for the purposes of this paper, we refer only to CTF matrix operations. An $n \times n$ CTF matrix is distributed across a **World** (an MPI communicator), and has attributes for symmetry, sparsity, and the algebraic structure of its elements. We work with adjacency matrices with weights in a set **W**, defined as

```
Matrix<W> A(n,n,SP,D,Y);
```

where **D** is a **World** and **Y** defines the **Monoid<W>** of weights with minimum as the operator.

CTF permits operations on one, two, or three matrices at a time, each of which is executed bulk synchronously. To define an operation, the user assigns a pair of indices (character labels) to each matrix (generally, an index for each mode of the tensor). An example function inverting all elements of a matrix **A** and storing them in **B** is expressed as

```
Function<int,float>([](int x){ return 1./x; })
B["ij"] = f(A["ij"]);
```

All CTF operations may be interpreted as nested loops, where one operation is performed on elements of multidimensional arrays in the innermost loop. For instance, in terms of loops on arrays **A** and **B**, the above example is

```
for (int i=0; i<n; i++)
  for (int j=0; j<n; j++)
    B[i,j] = 1./A[i,j];
```

For contractions, we can define functions with two operands.

We can express $\bullet_{\langle\oplus,f\rangle}$ by defining a functions **u** for operation $\oplus$ and **f** for f, then defining a **Kernel** corresponding to $\bullet_{\langle\oplus,f\rangle}$

```
Kernel<W,M,M,u,f> BF;
Z["ij"]=BF(A["ik"],Z["kj"]);
```

Provided that **Z** is a matrix with each element in **M**, while **A** is the adjacency matrix with elements in **W**, the above CTF operation executes $Z = A \bullet_{\langle\oplus,f\rangle} Z$. One could always supply the algebraic structure in a **Monoid** when defining the matrix, then use **Function** in place of a **Kernel**. However, the latter construct parses the needed user-defined functions as template arguments rather than function arguments, enabling generation of more efficient (sparse) matrix multiplication kernels for blocks. Having these alternatives enables the user to specify which kernels are intensive and should be optimized thoroughly at compile time, while avoiding unnecessary additional template instantiations.

Other CTF constructs employed by our MFBC code are

- **Tensor::write()** to input graphs bulk synchronously,
- **Tensor::slice()** to extract subgraphs,
- **Tensor::sparsify()** to filter the next frontier,
- **Transform** to modify matrix elements with a function.

More information on the scope of operations provided by CTF is detailed in [SH15] and previous papers.

4.5.6.2 Data Distribution Management

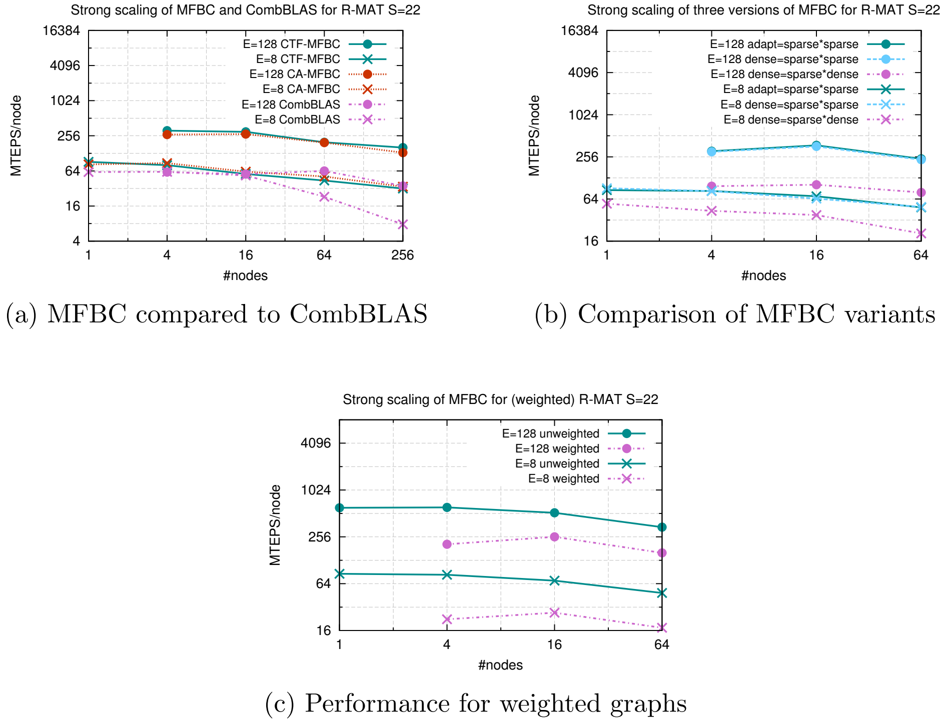
CTF is designed to permit the user to work obliviously of the data distribution of matrices. When created, each matrix is distributed over all processors using a processor grid that makes the block dimensions owned by each processor as close to a square as possible. For each operation (e.g., sparse matrix multiplication), CTF seeks an optimal processor grid, considering the space of algorithms described in Section 4.5.5 as well as overheads, such as redistributing the matrices.

Transitioning between processor grids and other data redistributions are achieved using three kernels: (1) block-to-block redistribution, (2) dense-to-dense redistribution, (3) sparse-to-sparse redistribution. Kernel (1) is used for reassigning blocks of a dense matrix to processors on a new grid, (2) is used for redistributing dense matrices between any pair of distributions, and (3) is used for reshuffling sparse matrices and data input. After redistribution, the matrix/tensor data is transformed to a format suitable for summation, multiplication, or contraction. For dense matrices, this involves only a transposition, but for sparse matrices, CTF additionally converts data stored as index–value pairs (coordinate format) to a compressed-sparse-row (CSR) matrix format.

CTF uses BLAS [LHKK79] routines for block-wise operations whenever possible (for the data-types and algebraic operations provided by BLAS). The Intel MKL library is additionally used for multiplication of sparse matrices, including three variants: one sparse operand, two sparse operands, and two sparse operands with a sparse output. Substitutes for these routines are provided in case MKL is not available. Further, for special algebraic structures or mixed-type contractions, all block multiplication and summation routines are implemented manually in CTF. Currently, these routines are threaded and lightly optimized, but do not contain hardware-specific optimizations.

CTF predicts the cost of communication routines, redistributions, and block-wise operations based on linear cost models. Aside from terms for latency α and bandwidth β, CTF additionally considers the memory bandwidth cost and computation cost of redistribution and block-wise operations. The dimensions of the sub-matrices on which all kernels are executed for a given mapping can be derived at low cost a priori. To determine sparsity of blocks, we scale by either the nonzero fraction of the operand matrix or the estimated nonzero fraction of the output matrix. Automatic tuning of the models allows the cost expressions of different kernels to be comparable on any given architecture. CTF employs a model tuner that executes a wide set of benchmarks on a range of processors, designed to make use of all kernels for various input sizes. Tuning is done once per architecture or whenever a kernel is added or significantly modified.

While having clear practical advantages and generality, the current CTF implementation cannot automatically exploit persistence of replication of the adjacency matrix, a technique utilized in the communication cost analysis of MFBC. This implies that CTF-MFBC incurs an additional bandwidth cost term of $O(\beta \cdot dn^2/p)$. When $c < p/d^2$, which holds for low-diameter graphs, this term does not increase the communication complexity of MFBC. On the other hand, for high diameter graphs, replicating the matrix redundantly can become an overhead. However, we leverage the ability of CTF to have user-defined initial processor grid mappings for each matrix to achieve persistent decompositions manually. In particular, we implemented a code we refer to as CA-MFBC, which maps all matrices to a $\sqrt{p/c} \times \sqrt{p/c} \times c$ processor grid. In this way, we ensure that the mapping proposed in Theorem 4.5.4 is used for every sparse matrix multiplication. The implementation still performs a broadcast to replicate the adjacency matrix for every sparse matrix multiplication, but the cost of data remapping, which is more expensive, is avoided.



(a) MFBC compared to CombBLAS

(b) Comparison of MFBC variants

(c) Performance for weighted graphs

Figure 4.13: Strong scaling of MFBC and CombBLAS for R-MAT graphs, which have roughly 2^S vertices and average degree E (weights are selected randomly between 1 and 100 for weighted graphs in Figure 4.13c)

4.5.7 Experimental Result

We present performance results for the two CTF-based implementations of MFBC (CTF-MFBC and CA-MFBC) on a Cray XC40 architecture. To validate the quality of our absolute performance, we compare these to the BC implementation in CombBLAS. We additionally compare variants of our implementation, which leverage a mix of sparse and dense matrices. We present strong scaling results on R-MAT graphs [CZF04], but focus on uniform-random graphs for weak scaling, as these are cheaper to generate. We evaluate the algorithms for two different weak scaling modes.

CTF-MFBC achieves the best strong and weak scalability, performing

consistently well for all graphs. In strong scaling for R-MAT with parameters $S = 22$ and edge factor $E = 8$ (yielding roughly 4 million vertices and an average degree close to 8), CTF achieves an 89× speedup from 1 to 256 nodes, increasing its performance advantage over CombBLAS from a factor of 1.5× to 4.1×. In weak scaling with an adjacency matrix of 1 percent density, on 256 nodes, CTF-MFBC reaches a performance rate of 20×10^{12} edge traversals per second, 8× faster than CombBLAS.

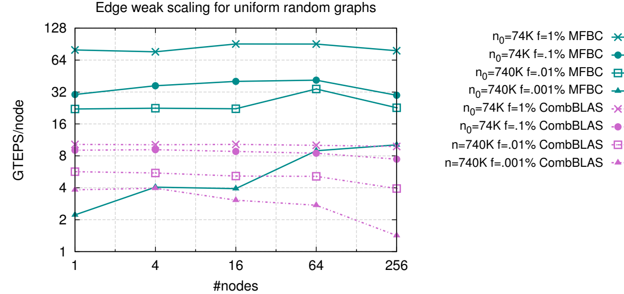
4.5.7.1 Experimental Setup

We debugged and tuned the implementation using the NERSC Edison supercomputer, a Cray XC30. Each Edison compute node has two 12-core HT-enabled Intel Ivy Bridge sockets with 64 GiB DDR3-1866 RAM. However, we collected final performance data using CSCS Piz Dora, a Cray XC40. Each node of Piz Dora has two 18-core Intel Broadwell CPUs (Intel® Xeon® E5-2695 v4). The network is the same on both machines, a Cray's Aries implementation of the Dragonfly topology [KDSA08]. The primary difference between these two Cray installations is the on-node setup and the scheduling system. We retuned our code to work with 36 cores per node rather than 24. The decision to collect the final results on Piz Dora was motivated by the relatively low level of performance variation we observed there.

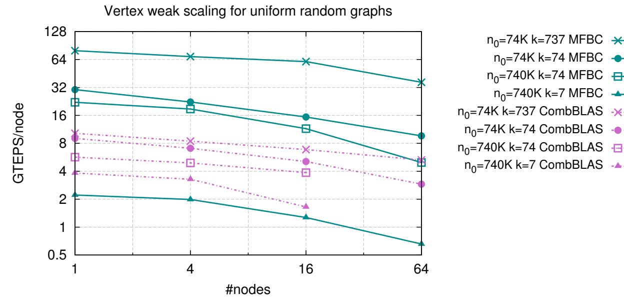
We report performance measurements of the minimum execution time for each configuration, which are predominantly collected by benchmarking a single batch of starting vertices. This benchmarking strategy is statistically questionable in the presence of significant machine noise or job interference, but there were no noticeable performance spikes on Piz Dora and our results show consistent trends. Collecting more performance data points was not possible under our allocation constraints without sacrificing performance, as CTF achieves highest performance when executed with the largest starting batch size possible. We used values of n_b of up to 6044, with each batch taking at least a minute to execute in all cases, which is sufficiently large for performance noise to be absorbed.

For comparisons with CombBLAS: Figures 4.13a, 4.14a, and 4.14b, we used CTF v1.4.1 and CombBLAS v1.5.0. For these runs, double precision floating point numbers were used for centrality scores. The performance data in Figures 4.13b and 4.13c were gathered for a slightly older CTF version, with centrality scores kept in single rather than double precision. The older set of results in these two plots suffices for comparing variants of our own code.

We use the metric of edge traversals per second (TEPS) to quantify perfor-



(a) constant $n^2/p = n_0^2$ and edge percentage $f = 100 \cdot m/n^2$



(b) constant $n/p = n_0$ and vertex degree $k = m/n$

Figure 4.14: Weak scaling of MFBC and CombBLAS for uniform random graphs

mance. The number of edge traversals scales with the size of the graph. For betweenness centrality on a connected unweighted n-vertex m-node graph, the total number of edge traversals is mn, as each edge is traversed to consider shortest paths from every starting node.

4.5.7.2 Strong Scaling

We begin our performance study by testing the ability of MFBC to lower time to solution by using extra nodes (strong scaling). We work with two R-MAT graphs, for both of which $\log_2(n) \approx S = 22$, while the average degree is controlled by $k \approx E \in \{8, 128\}$. The CTF implementations preprocess

R-MAT graphs by removing all disconnected vertices. R-MAT graphs have a low diameter, so a small number of matrix multiplications is done in the unweighted case.

Figure 4.13a compares the performance of CTF-MFBC, CA-MFBC, and CombBLAS betweenness centrality. For the R-MAT graph with a smaller average vertex degree ($E = 8$), CA-MFBC performs best, leveraging replication factors as large as $c = 16$ to achieve good strong scalability. However, CTF-MFBC always stays within 15% of the performance of CA-MFBC and is slightly faster for $E = 128$, indicating that CTF selects good processor grids automatically and does not incur too much overhead in redistribution.

CTF-MFBC outperforms CombBLAS for all data-points we gathered in the strong scaling tests. For $E = 8$, the margin between the implementations grows from 1.5× to 4.1× as the number of nodes is increased from 1 to 256. For $E = 128$, CTF is faster than CombBLAS by a factor of 5.1× on 4 nodes and a factor of 4.5× on 256 nodes. While CTF-MFBC improves its TEPS rate by a factor of 4 from this increase in edge factor, CombBLAS stays at roughly the same TEPS rate, but becomes communication-bound earlier for $E = 8$. This behavior is consistent with the communication cost of MFBC. The expected dominant communication term, $O(\beta \cdot n^2/\sqrt{cp})$ grows as $O(\sqrt{k})$ for graphs with average degree $k = m/n$ (through the dependence of c on m), while the number of operations grows with k.

Figure 4.13b compares the performance of three variants of CTF-MFBC. The first version, which is used for all results in other plots, represents both the graph and the matrix storing the frontiers (e.g., $\mathcal{T}$ in Algorithm 12) as sparse CTF matrices, and adaptively chooses between a sparse and a dense output matrix using a simple heuristic estimate of the output nonzero fraction. The second version always uses a dense output, while the third also uses dense representations of frontiers throughout. Figure 4.13b shows that using (selective) output sparsity does not significantly improve performance (the first two versions get roughly the same performance). R-MAT MFBC computation is dominated by sparse matrix multiplications whose output is

nearly dense. However, a substantial performance improvement is obtained by using a sparse matrix representation of the frontier (comparing the first two versions to the third).

Figure 4.13c tests the performance of CTF-MFBC for R-MAT graphs with edge weights randomly selected as integers in the range $[1, 100]$. In these tests, the number of sparse matrix multiplications doubles and the frontier stays relatively dense for several steps of Algorithm 12, thus the overall performance of MFBC decreases by more than a factor of two with the inclusion of weights.

4.5.7.3 Weak Scaling

We now test the parallel scalability of MFBC, while keeping m/p constant (weak scaling). Our weak scaling experiments work with uniform random graphs, in which all nodes have the same vertex degree, and every edge exists with a uniform probability. We consider "edge weak scaling" where n^2/p is kept constant and "vertex weak scaling" where n/p is kept constant. Aside from CombBLAS, weak scaling results are given only for our CTF-MFBC variant (from here-on just MFBC), which achieves good edge weak scaling, but deteriorates in efficiency for vertex weak scaling, a discrepancy justified by our theoretical analysis.

Figure 4.14a provides "edge weak scaling" results, in which the sparsity percentage of the adjacency matrix, $f = 100 \cdot m/n^2$, stays constant. The data confirms the observation that MFBC performs best for denser graphs. MFBC scales well in these scaling experiments, which is expected, since the communication cost term $O(\beta \cdot n^2/\sqrt{cp})$ grows in proportion with $\sqrt{p}$, while the amount of computation per node $O(mn/p)$ also grows in proportion with $\sqrt{p}$.

Figure 4.14b provides "vertex weak scaling" results, in which the the vertex degree k stays constant. We were unable to get CombBLAS to execute successfully on 64 nodes for the graphs with $n = 740K$ vertices. MFBC performs better than CombBLAS for all except that smallest degree graph, but

both implementations deteriorate in performance rate with increasing node count. This deterioration is predicted by our communication cost analysis, since in this weak scaling mode, the term $O(\beta \cdot n^2/\sqrt{cp})$ grows in proportion with $p^{3/2}$, while the amount of work per node $O(mn/p)$ grows in proportion with p. Therefore, unlike edge weak scaling, vertex weak scaling is not sustainable, the number of words communicated per unit of work grows with $\sqrt{p}$.

4.6 Summary and Discussion

In this chapter we discussed about new distributed algorithms for the computation of the betweenness centrality on large-scale graphs.

We first proposed a fast, distributed algorithm for the computation of betweenness centrality on Multi-GPU systems on unweighted graphs. Our solution encapsulates three different levels of parallelism by combining a fine- and coarse-grained approach using GPU accelerators. We also proposed a technique to avoid exchanging predecessors during traversal steps. This solution reduces the exchange of data from $\mathcal{O}(m)$ to $\mathcal{O}(n)$ regardless of the partitioning strategy adopted. The proposed algorithm is able to scale up to multiple GPUs maintaining comparable performance to state-of-the-art mono GPU implementations. On Multi-GPU systems, MGBC enables the BC computation of large scale graphs, both real-world like Twitter or Friendster and R-MAT with scale up to 29.

We also investigated the impact of heuristics on betweenness centrality computation by providing comprehensive experiments. In particular, on the contrary to previous works, we extended the degree-1 reduction heuristics on distributed systems and evaluated the impact on both the computation and the communication. We presented a novel heuristics for degree-2 vertices based on an innovative algorithm (DMF) where the betweenness contributions are augmented from its two neighbors without performing the Brandes' algorithm explicitly. We also provided a theoretical result which allows build-

ing a single source shortest path from a vertex if the shortest path trees of its own adjacencies are known. Experimental results validated the effectiveness of our approach. The heuristics offers a speed-up that is, at least, proportional to the number of skipped vertices. A greater improvement can be obtained by combining degree-1 and degree-2 heuristics, since this allows deriving the BC score of particular degree-3 vertices as well.

On weighted graphs, our new maximal frontier algorithm for betweenness centrality achieves good parallel scaling due to its low theoretical communication complexity and a robust implementation of its primitive operations. The algebraic formalism we use for propagating information through graphs enables intuitive expression of frontiers and edge relaxations, making it extensible to other graph problems such as maximum flow. We expect that the approach of selecting frontiers to maximize overall progress also leads to good parallel algorithms for other graph computations. MFBC with CTF outperforms Combinatorial BLAS and is robust to different graphs and parallel scaling regimes. Automatic parallelism for sparse tensor contractions with arbitrary algebraic structures is useful in many other application contexts. The rigorous communication-efficiency achieved by CTF for these general primitives has a promising potential for changing the way massively-parallel code is developed.

Chapter 5

Clustering Coefficient

Clustering coefficients is a widely-used graph analytics for measuring the closeness in which vertices cluster together [WS98]. It is a fundamental tool in network analysis that offers insights on how tightly bound vertices are in a network. Computing the clustering coefficients has been applied in several networks including communication [SS05], social [Str01], biological [BO04] and spam detection [BBCG08]. There are two types of clustering coefficients: global and local. The global clustering coefficient (GCC) is a single value computed for the entire graph, whereas the local clustering coefficient (LCC) is computed for each vertex. Both can be computed in a similar way. The best known algorithm for sparse graphs requires $\mathcal{O}(|V| \cdot \hat{\rho}^2)$ time steps, where $\hat{\rho}$ is the size of the maximum degree among all the vertices in the graph [SW05]. LCC computation is easy to parallelize since there are relatively large number of independent operations that can be executed. Coarse- and fine-grained parallelism can be easily adopted to achieve a faster time-to-solution. However, effective and scalable parallelization of LCC requires load balancing among computing nodes (or cores). For the reasons detailed in Section 1.2 this goal is not easy to achieve since some vertices requiring more operations, in terms of computation and communication, than others. Remote Memory Access (RMA) primitives allow to design new asynchronous algorithms. RMA systems are often implemented using remote

direct memory access (RDMA) hardware which high-throughput and low-latency networking. Within this context, the contributions of this chapter are as follows:

- A new asynchronous algorithm for LCC computation.
- A new cache layer (CLaMPI) to improve data locality for irregular applications based on MPI-One-sided primitives.
- A communication-efficient implementation of LCC based on CLaMPI.

5.1 Background

5.1.1 Local Coefficient Cluster

Let $G = (V, E)$ be an undirected graph, the symbol $\rho(v)$ denotes the degree of a vertex v, whereas $\Gamma_G(v)$ represents the neighbours of v.

The Local Clustering Coefficient (LCC) [WS98] of a vertex $v \in G$ represents the fraction of possible links among the vertices within adjacency of v. Formally, the LCC of a vertex v on unweighted and undirected graphs is given by the following equation:

$$LCC(v) = \frac{2 \cdot |\{(u, w) : u, w \in \Gamma_G(v), (u, w) \in E\}|}{\rho(v) \cdot (\rho(v) - 1)} \tag{5.1.1}$$

A naive approach to the LCC computation consists in enumerating the triangles in the graph. There are several approaches for computing clustering coefficients:

- Enumerating over all node-triples. This has an $\mathcal{O}(V^3)$ upper bound complexity.
- Using matrix multiplication. This has an $\mathcal{O}(V^3)$ complexity where $\omega \leq 2.376$ [CW87]. On sparse graphs, the time complexity is func-

tion of the number of non-zero (nnz) elements of the adjacency matrix representing the graph G.

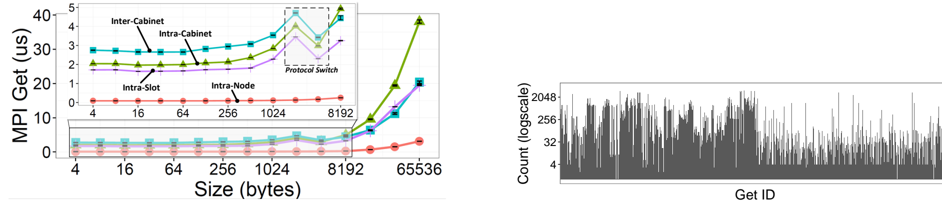
- Intersecting adjacency lists proposed by Shank *et al.* has $\mathcal{O}(|V| \cdot deg_{max}^2)$ time complexity, where deg_{max} is the size of the largest adjacency list in the graph.

5.1.2 Caching and MPI-3 One-Sided in a nutshell

Software and hardware caches are among the most important components of modern computer systems. CPU caches are an integral part of modern microprocessors and improve performance and reduce energy consumption by exploiting spatial and temporal locality in the application's memory access pattern. Similarly, block caches are a standard component in I/O systems such as magnetic hard drives or distributed network file systems [HKM+88]. In all these cases, the goal is to exploit locality between the application layer and the underlying memory or disk subsystem in the so called *vertical* communications.

Networked high-performance computing applications, like graphs analytics, usually run multiple processes that communicate not only with their respective main memories and I/O subsystems but also with each other in the so called *horizontal* communications. Horizontal communication is most often implemented with the Message Passing Interface (MPI) layer, a library interface for communicating processes. MPI offers two main modes for communications: message passing (MP, also known as two-sided) and remote memory access (RMA, also known as one-sided). RMA systems are often implemented using remote direct memory access (RDMA) hardware support [The04, AAC+10, KMF+12, FBR+12].

Caching is mostly motivated by the latency and bandwidth difference between a local typically small but fast layer (e.g., an on-chip SRAM memory) and a slower but large memory (e.g., off-chip DRAM) in the vertical configuration. Performance differences between the fast and slow memories



(a) Access latency as function of the message size and different node mapping of the involved processes

(b) NBody simulation example. Number of times the same get is issued by the a process. Number of processes: 4; Bodies: $4K$

Figure 5.1: Difference of performance of local vs. remote accesses in RDMA environments.

are often an order of magnitude and larger. We observe similar performance differences for local vs. remote accesses in RDMA environments. Figure 5.1a shows the access times for various "distances" in a Cray Cascade system based on the Aries interconnect. Here, latencies range from less than 100ns for a local DRAM access (less than ten nanoseconds if the access is cached on the CPU chip) up to 2-3 microseconds for remote accesses, spanning three orders of magnitude. The major difference is between process- or node-local accesses and off-node network accesses. Moreover, we observe that many applications characterized by irregular access patterns present exploitable data reuse. Figure 5.1b shows how in an N-Body simulation algorithm, the same *get* operation (i.e., targeting the same data) can be issued up to $3K$ times (averaged on all the processes). Motivated by these observations, we propose a transparent caching layer for RMA programming systems that caches data from horizontal remote accesses in local DRAM.

Our implementation targets the MPI-3 RMA specification. Our library, called CLaMPI (short for Cache Layer for MPI), can be linked to existing MPI-3 RMA programs and caches remote communication. CLaMPI also allows to add attributes in the form of standard MPI info arguments to windows to enable a more fine-granular control of the caching behavior. Overall, CLaMPI does not require any invasive extension to the MPI standard and it

is already compliant in the *transparent* and *always-cache* operational modes.

The MPI-3 standard [For12] defines One-Sided operations, enabling Remote Memory Access (RMA) semantic. We now briefly describe the RMA semantics needed in this paper, a full overview of the specification is provided by Hoefler et al. [HDT+15]. A set of processes belonging to a specific *communicator* (e.g., `MPI_COMM_WORLD`) can expose a memory region, defined as *window*, over the network by joining the **MPI_Win_create** or **MPI_Win_Allocate** collective. Once the *window* is created, processes can perform RMA operations on it: **MPI_Put**s and **MPI_Get**s are used to read and write data to/from remote processes directly. They are often supported by remote direct memory access (RDMA) hardware to provide the necessary performance.

RMA operations on a specific window can be issued only during access epochs. Synchronization calls are used to start/terminate an access epoch. Furthermore, the MPI standard defines two synchronization modes: active and passive. Passive synchronization does not require the participation of the target process: the initiator (i.e., the one that issues the RMA operation) can access the target's window during the section delimited by the **MPI_Win_lock** and **MPI_Win_unlock** calls. On the contrary, active synchronization requires the participation of all the processes that are sharing the same window. In this case epochs are delimited by **MPI_Win_fence** collective calls.

Both MPI *put* and *get* operations are non-blocking. The synchronization call that concludes an epoch ensures that all the RMA operations issued during this period are completed when the call returns. In the passive synchronization case, an epoch can be concluded also with the **MPI_Win_flush** call. We use the term *window flushing* to indicate the waiting for the termination of all (or a subset of) the RMA operations issued during an epoch.

There are many studies on software caching schemes for vertical communications [RD01,PDZ00] and their interaction with hardware caches [HIT05]. These systems represent a class of systems that are designed for blocking interfaces where the application waits for the completion of a request before starting the next request. Furthermore, these schemes are optimized for sys-

tems that request data in large blocks. We only exemplify some systems here for space reasons.

For horizontal communications, two systems exist for PGAS languages: a software caching system for UPC [CDS03] and Chapel [FB15]. Both follow traditional approaches from vertical caching and implement a standard block-based read/write caching scheme.

To the best of our knowledge, CLaMPI is the first caching scheme aiming at the asynchronous epoch-based MPI-3 RMA system. Here, we innovate on multiple fronts: we devise variable-block-size scheme which has constant overhead even in the worst-case. Furthermore, we focus on reads because MPI's semantics support write accesses well at the user-level. CLaMPI integrates with the MPI-3 RMA epoch consistency model and thus simplifies consistency management significantly. Our performance analysis shows that our fully-transparent software caching layer nearly reaches the performance of hand-crafted algorithm-specific implementations.

5.1.3 Caching RMA

Caching RMA operations is different from traditional cache designs. First, other caches (e.g., CPU or file-system) usually accelerate synchronous (blocking) accesses that need to be consumed immediately MPI enables asynchronous communication arranging accesses in epochs. Thus, while it is important for traditional caches to quickly consume writes, MPI does not benefit of write accesses caching because the MPI epoch models forbids a) overlapping writes and b) overlapping writes and reads in the same epoch [HDT+15]. As a consequence, each write must be issued to a different target location - i.e., local caching can thus not prevent network accesses - and writes cannot target same location of reads - i.e., read after write patterns cannot be exploited. Furthermore, the asynchronous nature of MPI epochs enables overlapping computation at communication at the programming level. Thus, **we focus on caching gets**. Another fundamental difference is that MPI/RDMA operates at byte granularity as opposed to CPU caches (operat-

ing at cache-line granularity) or disk caches (operating at block granularity). **This requires the caching system to handle variable-size cache entries**. Figure 5.2 shows the get sizes distribution presented by an instance of the Local Clustering Coefficient computation. The issued get sizes span over four orders of magnitude, hence motivating our choice of adopting variable-size cache entries.

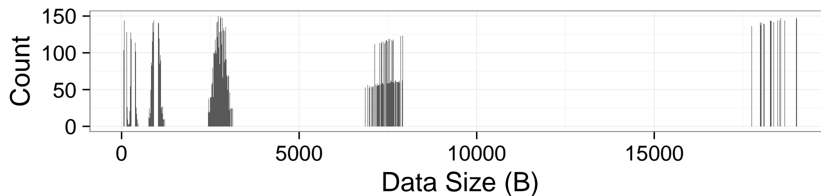


Figure 5.2: Data Size distribution of a Local Clustering Coefficient (LCC) instance, averaged on 32 processing elements. R-MAT graph with S=16 and EF=16.

According with the MPI epoch model, the destination buffer of a *get* issued in an epoch i can be considered ready - i.e., the requested data is available - only at the end of epoch i. After the *get* is completed, no assumptions on the destination buffer can be made by the MPI layer. This leads to the need of keeping a separate storage area for the cached *get*s. In addition, RDMA networks allow only one local destination buffer, so **data has to be explicitly copied into the cache at the end of the issuing epoch**.

RMA operations transfer data among the process from which the operation originates (i.e., the initiator) and a target process. We define a *get* as an operation transferring data from the target to the initiator. A *get* operation is uniquely identified by a tuple:

$$get = (win, eph, src, dst, off, size)$$

Where win is a MPI window, eph is the epoch in which x is issued, src is the rank of the initiator, trg is the rank of the target, off is the displacement in the target window and $size$ is the size of the data to transfer.

An epoch counter $w.eph$ is associated with each window w, counting the number of concluded epochs since the window creation. A *get* issued on w takes the current $w.eph$ as epoch identifier: $x.e = w.eph$. All the *get*s requests targeting a caching-enabled window w are processed by the caching layer associated with w. A *get* targeting caching-enabled window is referred as get_c. A caching-enabled window is associated with a caching layer $C_w = (I_w, S_w)$, where I_w and S_w are the data structures used for indexing and storing cache entries, respectively. The number of entries that can be indexed by C_w is $|I_w|$, while the total memory space that can be occupied by the cached entries is $|S_w|$.

5.2 Related Work

There are many studies on software caching schemes for vertical communications [RD01, PDZ00] and their interaction with hardware caches [HIT05]. These systems represent a class of systems that are designed for blocking interfaces where the application waits for the completion of a request before starting the next request. Furthermore, these schemes are optimized for systems that request data in large blocks. We only exemplify some systems here for space reasons. For horizontal communications, two systems exist for PGAS languages: a software caching system for UPC [CDS03] and Chapel [FB15]. Both follow traditional approaches from vertical caching and implement a standard block-based read/write caching scheme. To the best of our knowledge, CLaMPI is the first caching scheme aiming at the asynchronous epoch-based MPI-3 RMA system. Here, we innovate on multiple fronts: we devise variable-block-size scheme which has constant overhead even in the worst-case. Furthermore, we focus on reads because MPI's semantics support write accesses well at the user-level. CLaMPI integrates with the MPI-3 RMA epoch consistency model and thus simplifies consistency management significantly. Our performance analysis shows that our fully-transparent software caching layer nearly reaches the performance of

hand-crafted algorithm-specific implementations.

Concerning Local Coefficient Cluster problem, Green and Bader [GB13] proposed a novel clustering coefficients algorithm that employs vertex covers in order to reduce the number of list intersections and the number of actual comparisons needed to compute the triangle enumeration. Green *et al.* described two scalable approaches to improve the work balancing on shared-memory system [GMB14]. Azad *et al.* implemented triangle counting on distributed system using matrix multiplication [ABG15]. Their solution is based on MPI two-sided primitives. Finally, Wang *et al.* provided a comparative study on triangle enumeration on GPUs [WWYO16].

Contrary to existing works, we focus mainly on an asynchronous algorithm for LCC computation. Our communication framework can be employed on top of different parallelization strategies.

5.3 CLaMPI: a caching layer for MPI One-sided

In this section we propose CLaMPI, a caching system that follows the principles discussed in Section 5.1.3. In Section 5.3.1 we describe how applications can be interfaced with CLaMPI. In Section 5.3.2 we discuss the logic of the proposed caching layer. The aim of Section 5.3.3, 5.3.4 and 5.3.4.1 is to give a detailed description of the design/implementation choices.

One of the our main goals is to minimize the cost of the cache hit while introducing a minimal overhead w.r.t. the non-cached *get* operation in the cache-miss case. An optimal handling of the cache-hit case would consist of only the local data-copy from the cache to the user buffer specified in the get. Similarly, an optimal cache-miss would perform the get operation for retrieving the data plus one additional memory copy. However, additional overheads stemming from cache managing activities - i.e., renaming, lookup, allocation, and replacement - have to be taken into account.

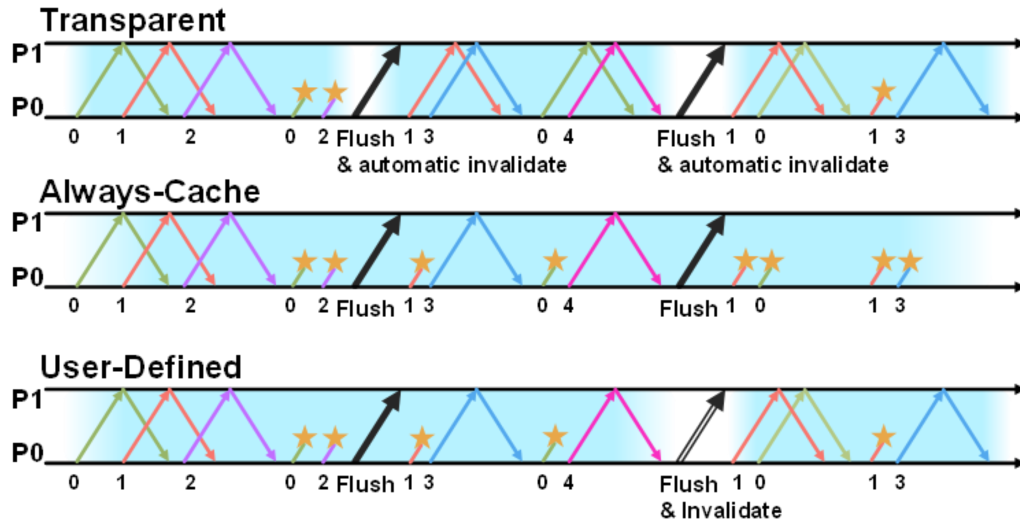


Figure 5.3: Operational Modes for Caching-Enabled Windows

5.3.1 Caching-Enabled Windows

CLaMPI offers three different strategies to enable RMA caching: *transparent*, *always-cache* and *user-defined*. If a window is defined as caching-enabled, then all the *get* operations issued on that window will be processed by the caching layer.

5.3.1.1 Transparent

The *transparent* mode allows applications to enable caching without requiring code changes. In this case, all the created MPI windows are caching-enabled. In this operational mode, no assumptions can be made on the data access pattern. Thus, in order to guarantee data consistency, the cache is invalidated at each epoch closure. This approach is effective only if the application presents reuse within epochs.

5.3.1.2 Always-Cache

If the memory area identified by the window is read-only for the entire window lifespan then there is no need to perform any cache invalidation. Examples of such application are the ones applying graph-processing algorithm (e.g., BFS): if the graph structure is not modified, then the window that represents it can be set in the always-cache mode. Such information can be communicated to CLaMPI as a MPI_INFO key passed at the window creation time - i.e., as argument of MPI_Win_create or MPI_Win_allocate.

5.3.1.3 User-Defined

This strategy let the user define epochs, or sets of consecutive epochs, in which the memory area identified by a window is in a read-only state. Use cases that can take advantage of this operational mode are, for example, BSP-like (Bulk Asynchronous Parallel) applications presenting steps where no write accesses are performed towards the specific window.

```
MPI_Win_lock(MPI_LOCK_SHARED, peer, 0, win);
while (!terminate){
  MPI_Get(lbuf1, ..., peer, offset1, ..., win);
  MPI_Get(lbuf2, ..., peer, offset2, ..., win);
  MPI_Win_flush(peer, win);
  terminate = computation(lbuf1, lbuf2);
}
MPI_Win_unlock(peer, MPI_CACHE_FLUSH, win);
```

Listing 5.1: Example of User-Defined Caching Strategy

In order to use this strategy, the user has to create the window with the *always-cache* option. Then the cache can be explicitly flushed using the MPI_CACHE_FLUSH flag as *assert* parameter of the epoch closure functions. It is worth noting that, in the current version of the MPI-3 standard, the MPI_Win_flush call does not accept an *assert* parameter. While we believe that this will be fixed in future versions of the standard, we

propose a work-around by providing an explicit cache invalidation call `CLAMPI_Invalidate(MPI_Win win)`.

An usage example for such operational mode is reported in Listing 5.1: a set of read-only epochs - i.e., where no write accesses are issued - is delimited by the `MPI_Win_lock` and `MPI_Win_unlock` functions. All the read accesses performed in such epochs are kept in cache. The cache is explicitly invalidated using `MPI_CACHE_FLUSH` flag in the `MPI_Win_unlock` function.

5.3.1.4 Discussion

The proposed caching strategies do not require any modification to the MPI standard, expect for the introduction of the additional macro `MPI_CACHE_FLUSH`. An alternative design could extend the MPI specification to offer special *get* calls to the user, allowing to use/bypass the caching on a per-operation basis. In the current MPI-compliant model, the user could achieve the same by creating two windows, with the same local memory, and enabling just one of the two for caching. At this point the user can issue operations on the two different window according with the need to cache such operations or not.

5.3.2 Processing Gets

When a get_c is issued on a window $\bar{w}$, the index $I_{\bar{w}}$ is queried in order to check if the entry is already in cache or not. The result of this query represent the state of the searched cache entry. In particular, a cache entry can be in one of the following states: `MISSING`, `PENDING`, or `CACHED`. Figure 5.4 sketches the possible state transitions.

MISSING

Since a `MISSING` entry is not stored in the cache, a get_c targeting an entry in such state leads to a cache miss. When a cache miss occurs on a window $\bar{w}$, a remote *get* is issued in order to acquire the requested data. The destination

buffer of the issued get is the user-provided one. This allows to overlap the remote *get* with the caching overheads. In particular, such overhead is required in order to check if $I_{\bar{w}}$ and $S_{\bar{w}}$ are able to respectively index and store the `MISSING` entry and to execute the eviction procedure in the negative case. Differently from the general model described in Section 5.1.3, *if a capacity access happens, CLaMPI will select exactly one victim to evict*. If on one side this does not guarantee that it will be possible to store the new entry after the eviction, on the other leads to a constant number of evictions per cache miss. In particular, we have a *capacity* access only if it is possible to store the data requested by the get_c after the eviction, otherwise we have a *failing access*. A get_c can fail only due to missing space in $S_{\bar{w}}$. Cache entries targeted by *direct*, *conflicting* or *capacity* accesses are moved to the `PENDING` state.

CACHED

A successful lookup in $I_{\bar{w}}$ returns an entry in the `CACHED` state. The available data is directly copied in the user provided buffer. If the hit is only partial, a remote *get* is issued in order to acquire the missing data. A cache entry in the `CACHED` state that is selected as victim from the eviction procedure is moved to the `MISSING` state.

PENDING

A cache entry is in the `PENDING` state when the associated data is “in flight”: a get_c targeting such entry has been issued in the current epoch finding it in the `MISSING` state. If a get_c finds the targeting entry in the `PENDING` state, a pending copy will be enqueued to the pending copies queue. When the current epoch terminates, all the entries in such state are moved to the `CACHED` state and the pending copies queue is empty.

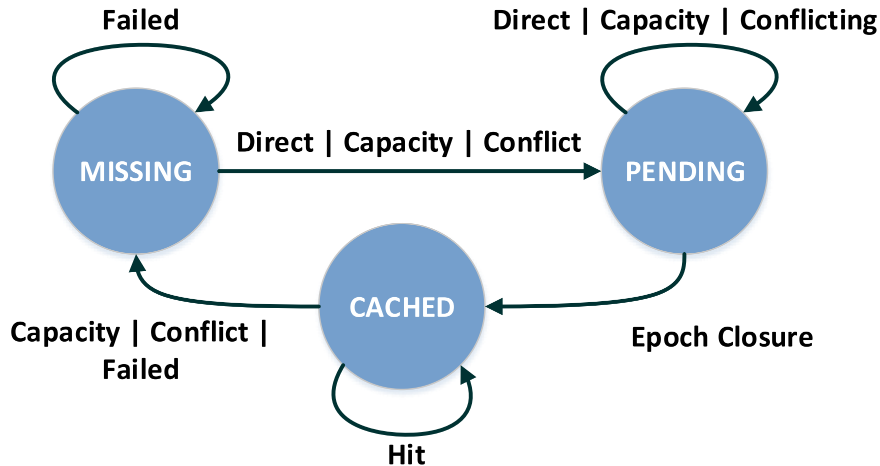


Figure 5.4: Cache entry state diagram. The initial state of any cache entry is `MISSING`.

5.3.2.1 Discussion

The missing guarantee to be always able to store a new cache entry make our proposed caching layer following a weak-caching approach. We motivate our design choice with the following points:

- Cache entries have variable size: multiple evictions could be required in order to make room in $S_{\bar{w}}$ to store the incoming data. This would lead to an overhead proportional to the current number of cached entries in the worst case.

- If a get_c is targeting highly-reused data, then it will be issued multiple times, leading to multiple evictions, hence increasing its own probability of being successfully cached.

5.3.3 Data Structures

This section describes how the data structures for the indexing $I_{\bar{w}}$ and the storage $S_{\bar{w}}$ of the cached entries are implemented, assuming a caching layer

$C_{\bar{w}}$ on a window $\bar{w}$. Figure 5.5 sketches the described data structures.

5.3.3.1 Naming - Indexing Entries

In CLaMPI, cache entries are indexed using a hash table. This leads to a different cache-hit definition respect to the one described in the general model (see Section 5.1.3). In the proposed caching layer, given a $get_c\ x$ operation on $\bar{w}$, the hash table is queried using the tuple $(x.t, x.d)$ as key. In particular, we have a cache-hit if there exists an entry (t', d', s') in $I_{\bar{w}}$ targeting a) the same remote node and b) a memory area sharing the same starting address of the issued get_c: $\exists(t', d', s') \in I_{\bar{w}} :\ \ x.t = t' \wedge x.d = d'$. Please note that the general approach proposed in Section 5.1.3 requires the adoption of data structures requiring an average $O(\log N)$ cost for the lookup, where N is the number of cached entries (e.g., interval trees [Wit84]). We employ the Cuckoo scheme [PR01] for resolving hash collision, enabling constant lookup cost in the worst case. This scheme uses p hash functions $h_0(e)...h_{p-1}(e)$ to identify the possible locations of an entry e in the hash table. Universal hashing [CW77] can be used in order to derive the p hash functions. The insertion procedure tries to insert a new element x in $h_i(x)$ where i is randomly chosen in $[0, p)$. If $h_i(x)$ already contains an element y - i.e., $h_i(x) = h_i(y)$ - then x is inserted in $h_i(x)$ and y is taken in exam. The procedure continues iterating trying to insert y in $h_{(i+1) \bmod p}(y)$. We call the sequence of hash table entries visited during the insertion procedure as "insertion path". The procedure stops when either an empty position is found or a maximum number of iteration is reached. This threshold helps to detect cycles in the Cuckoo graph. In general, the second case is handled as an *insertion failure*: a new set of hash functions is selected and all the entries are re-hashed. In our approach we do not perform the rehashing of all the entries in the *insertion failure* case in order to avoid a $O(N)$ time overhead. Instead, we handle this case as a *conflicting* access, triggering the eviction procedure in order to evict one of the cache entries in the insertion path. The victim selection scheme is described in Section 5.3.4.1.

5.3.3.2 Storage - Allocating Space

The storage data structure $S_{\bar{w}}$ is implemented as a memory buffer of size $M = k \cdot B$ bytes where B is a system parameter identifying the block size. We organize $S_{\bar{w}}$ in blocks of size B, which is a multiple of the CPU cache line size, in order to maintain the data aligned with CPU cache lines. In order to preserve such alignment, no meta-data information are stored along the actual data. A cache entry $c = (t, d, s)$ occupies $\lceil \frac{c.s}{B} \rceil$ contiguous blocks. We keep track of the free blocks using a bitmap of k bits. Storing cache entries in a set of contiguous blocks allows us to exploit hardware prefetching when accessing subsequent CPU cache lines during the memory copy. However, such layout may cause external fragmentation of the memory buffer. In order to tackle this problem, we associate a positional score with each cache entry, giving an estimation of how the entry contributes to the fragmentation of $S_{\bar{w}}$.

Given a caching layer $C_{\bar{w}}$, let $C_{\bar{w}}.G$ be the sequence of get_c issued in the program order and processed by $C_{\bar{w}}$. We define $C_{\bar{w}}.ags(i)$ as the Average Get Size of the entries in $C_{\bar{w}}$ after the i-th get_c in $C_{\bar{w}}.G$, indicated with x_i. This measure is computed as the exponentially weighted moving average of the cached entries:

$$C_{\bar{w}}.ags(i) = \begin{cases} x_i.s, & i = 0 \\ \alpha \cdot C_{\bar{w}}.ags(i-1) + (1-\alpha) \cdot x_i.s, & i > 0 \end{cases}$$

where $\alpha \in [0, 1]$ is the degree of weighting decrease: a lower α will make the average more sensible to new observations.

Given a cache entry $c = (t, d, s)$ in $C_{\bar{w}}$, we define d_c as the number of consecutive free blocks in $S_{\bar{w}}$ adjacent to c. The positional score $P^i_{\bar{w}}(c)$ of c after the $i - th$ processed get_c is computed as:

$$P^i_{\bar{w}}(c) = \min\left\{ \frac{|C_{\bar{w}}.ags(i) - d_c|}{C_{\bar{w}}.ags(i)}, 1 \right\}$$

After the $i - th$ get_c has been issued, if the number of consecutive free blocks adjacent to c (i.e., d_c) is close to the $C_{\bar{w}}.ags(i)$, then c gets a low positional score: it is more likely that its eviction will make possible to reuse enough space to store a get of size close to $C_{\bar{w}}.ags(i)$. Instead, a high positional score indicates that the number of free blocks adjacent to x is too low with respect to the current average get size of $C_{\bar{w}}$ - hence its eviction will not de-fragment enough space - or too high - hence its eviction will de-fragment more space than needed.

5.3.3.3 Cache Entries Descriptors

Every cache-entry c is associated with a descriptor $CLEntry(c)$, where meta-data information about such entry are stored.

In order to compute the positional score of a cache entry c, we have to update the associated d_c every time a cache entry is inserted in or evicted from a memory region adjacent to c. In order to efficiently perform this update we organize entry descriptors in doubly linked list. Given an entry $CLEntry(c)$, we indicate with $CLEntry(c).prev$ and $CLEntry(c).next$ the previous and the next element in the list, respectively. In particular such list reflects the order in which the entries are stored in $S_{\bar{w}}$:

$$CLEntry(a) = CLEntry(b).prev \implies a.o < b.o$$

Maintaining such ordered list could lead to an insertion cost linear in the number of cached entries. To avoid such overhead we enforce the following property:

Property 5.3.1 *When storing a new cache entry c in $S_{\bar{w}}$, the pointer to its previous element in the CLEntry list can be found in position c.o.*

The above property holds since: 1) when storing a cache entry x, if the block in position $x.o + \frac{(x.s+B)}{B} \cdot B$ is marked as free, a pointer to $CLEntry(x)$ is written in it. This imposes the constraint $B \geq ptr_size$, where ptr_size

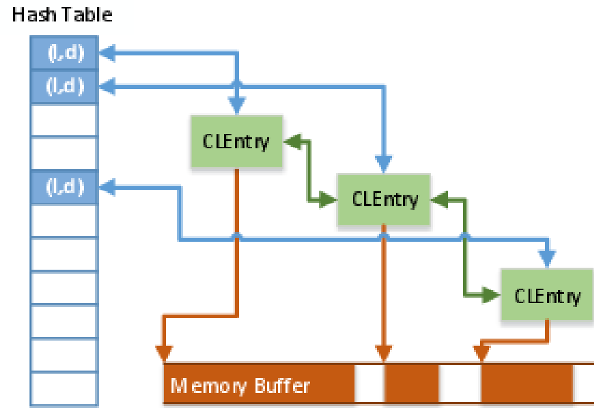


Figure 5.5: Employed data structures. The information related for the score computation of an entry x (i.e., l_x and d_x) are kept directly in the associated hash table entry.

is the size of a memory pointer on the target architecture; 2) during the eviction of a cache entry x, the pointer to $CLEntry(x).prev$ is written in the block starting at $x.o$. It is worth nothing that once such pointer is read during the storage of c, it can be overwritten by the copy of the incoming data, preserving the system cache line alignment.

5.3.4 Eviction Procedure

A cache miss is originated in a caching layer $C_{\overline{w}}$ as a consequence of one of the following access types: *direct*, *conflicting* or *capacity*. The *direct* access is the simplest one: enough resources to index and store the missing cache entry are available without requiring any eviction. In the following we describe the actions that are taken in the more complex cases of *conflicting* and *capacity* accesses.

If it is not possible to store a missing entry in $S_{\bar{w}}$, the eviction procedure has to be triggered in order to select and evict a cached entry. We model the hash table as an array of $M = |I_{\bar{w}}|$ entries, namely *entry_array*. The eviction procedure selects the victim among *sample_size* entries in the *entry_array*. Such entries are sequentially visited starting from a randomly chosen point in the array and in a circular manner. In particular, the eviction procedure keeps scanning entries until at least one non-empty entry is found. Hence, in order to select a victim, the actual number of entries that have to be visited starting from a position i of $I_{\bar{x}}$ is $v_i = \max(sample_size, k_i)$, where k_i is the number of consecutive empty entries starting from the i-th entry. The value of k_i depends on the sparsity of the *entry_array* and hence on the ratio $l = \frac{\min(H(M),N)}{M}$, where N is the number of distinct gets issued on $C_{\bar{w}}$ and $H(M)$ is the maximum number of entries that can be stored in $I_{\bar{w}}$. Please note that $H(M) \leq M$ since the Cuckoo scheme is not able fully utilize the hash table capacity [PR01]. Small values of the ratio l lead to an increased sparsity of the *entry_array*, hence to a lower number of non-empty entries visited by the eviction procedure. In particular, we define q as the ratio between the number of non-empty and total entries visited by the eviction procedure. Such indicator is used in Section 5.3.5.1 to adapt the $|I_{\bar{w}}|$ at runtime. Let us define $p(x) = CLEntry(x).prev$. The freed space due to the eviction of the entry x is identified by the interval $[p(x).o + p(x).s, x.o + x.s)$. The case in which issued get_c will fit in such interval is identified as *capacity* access. On the contrary, we identify the case in which the freed space is not enough to store the requested data as *failing* access.

5.3.4.1 Victim Selection

The eviction procedure can be triggered in two cases: 1) a conflict during the insertion in the hash table; 2) not enough free space in the memory buffer. For each cache entry we keep track of the index of the last get_c that was matched by. The index of a *get* is the index of such operation in the sequence of the issued operations since the creation or the last flush of the

targeted window. The temporal score of a cache entry x after the i-th *get* is the ratio between the last time such cache entry was hit l_x and i:

$$S_L^i(x) = \frac{l_x}{i}$$

We define the score of a cache entry as function of its temporal and positional score:

$$S^i(x) = S_P^i(x) \cdot S_L^i(x) \quad 0 \leq S^i(x) \leq 1$$

The aim of $S^i(x)$ is to provide an estimation of how x contributes to the fragmentation of the memory buffer and of what is the probability that such entry is going to be reused by the application When the eviction procedure is triggered, the victim is selected as the entry with the lowest score among the ones taken in exam.

5.3.5 Parameter Tuning

In Section 5.3.3 we discussed the data structures employed for indexing and storing the cache entries. The size of the index data structure $|I_{\bar{w}}|$ and of the memory buffer $|S_{\bar{w}}|$ are the most critical parameter of the proposed caching layer. The sequence of gets that are originated by a process and targeting a specific window $\bar{w}$ is identified by $r_{\bar{w}} = g_1 g_2 ... g_k$. We define the working set $W_{\bar{w}}(t, \tau)$ as the set of gets issued on a caching layer $C_{\bar{w}}$ over the interval $[t - \tau, t]$ [Den68], where t and τ are indices of $r_{\bar{w}}$. The set of gets belonging to the working set and that are stored in cache at time t is defined as $\gamma(t, \tau)$. The above discussed parameters limit the set $\gamma(t, \tau)$ in the following way:

$$|\gamma(t, \tau)| \leq |I_{\bar{w}}|$$

$$\sum_{g \in \gamma(t,\tau)} g.s \leq |S_{\bar{w}}|$$

In particular, the total number of indexable entries, $|I_{\bar{w}}|$, limits the total number of cache entries that can be indexed $|\gamma(t, \tau)|$, while the memory

buffer size $|S_{\bar{w}}|$ limits the total space that the entries in $\gamma(t, \tau)$ can occupy. Hence, $|I_{\bar{w}}|$ and $|S_{\bar{w}}|$ have a direct impact on the number of conflicting and capacity accesses, respectively.

5.3.5.1 Adaptive Parameter Selection

Tuning the above described parameters can be a challenging task for the user, especially if we consider their direct impact on the application performance. In this section we propose a strategy that allows the CLaMPI to adjust such parameters at runtime by itself. The idea is that the starting values of $|I_{\bar{w}}|$ and $|S_{\bar{w}}|$ are predefined. The caching layer will then increase or decrease such values keeping track of some statistics about the cache usage at runtime. It is worth noting that changing the value of such parameters requires the invalidation of the cache. A high number of conflicting accesses is a signal that the $I_{\bar{w}}$ has to be extended. When the ratio $\frac{conflict}{total_gets}$ is greater than a *conflict threshold* c_t^+, the hash table size $|I_{\bar{w}}|$ is linearly increased by a factor c_i. In Section 5.3.4 we define q as the ratio between the number of non-empty and total entries visited by the eviction procedure. We identify the event of this value getting lower of a certain threshold c_t^- as a signal of highly sparse hash table. In this case, in order to improve the quality of the victim selection (i.e., increasing q), we decrease the hash table size by a factor c_d. The number of capacity and failed accesses is monitored in order to increase the memory buffer size $|S_{\bar{w}}|$ when needed. In particular, when the ratio $\frac{capacity+failed}{total_gets}$ becomes greater than a *capacity threshold* s_t^+, the memory buffer is increased by a factor of s_d. In the rest of the paper we use the terms *fixed* and *adaptive* to refer the two strategies where the discussed parameters are fixed during the whole caching layer lifespan or dynamically adjusted, respectively.

5.4 Scalable LCC Algorithm

Assume there are $|P|$ processes $P = (p_1, \ldots, p_{|P|})$ in a distributed system. Let b be the the number of processes that may run on the same computing node cn_j with $1 \leq j \leq \left\lfloor \frac{|P|}{b} \right\rfloor$. P_j denoting the partition of P assigned to the computing node cn_j.

Without loss of generality, we assume that $G(V, E)$ is partitioned and distributed among P processes by using a 1-D schema [BM13]: V is divided into P subsets V_i and every $V_i \subseteq V$ is assigned to a different process p_i. The process p_i owns all the vertices $v \in V_i$ and all the edges (v, u) such that $v \in V_i$, $u \in V$.

We propose a new algorithm to compute the LCC of $G(V, E)$ based on MPI-One-sided. The pseudo-code is shown in Algorithm 15 . Each process p_i performs Algorithm 15 over its own partition. In order to compute the LCC of a local vertex $v_j \in V_i$, the algorithm has to retrieve the adjacency list of every incident vertex u (lines $5 - 12$). In case the owner of u is different from p_i, the retrieve operation can be performed as a *one-sided* communication (line 8). Concerning implementation aspects, the algorithm stores its own fraction of the graph into a Compressed Sparse Row (CSR) data structure, therefore the subroutine `Get`() performs two different *get* operations. The first one is used to determine the offset of the vertex in the remote CSR. The second one is performed in order to obtain the adjacency of the remote vertex. The number of *get*s issued by a single process depends on the size of its own partition and on the degree of the vertices in such partition. The size of the each issued *get* depends on the degree of each neighbour of the vertex v. Except for the subroutine `Get`(), all the other procedures exploit data locality. Real world graphs are characterized by high sparsity with a power-law distribution of the vertices degree. Synthetic graphs are usually generated respecting such properties (e.g., R-MAT [CZF04]). As a consequence, static graph partitioning strategies - such as one-dimensional schema - tend to introduce high load unbalancing among the involved processes [BM13]. Such

phenomena typically leads to a situation in which the probability of performing a remote access increases with the number of processes and with the size of the graph. On large-scale graphs, by increasing the number of processes used to analyze the graph, the probability to perform a *point-to-point* communication primitive increases in turn. The LCC computation exposes data reuse since the adjacency list of the same vertex u can be accessed several times by a single process: every time u appears in the adjacency list of an owned node.

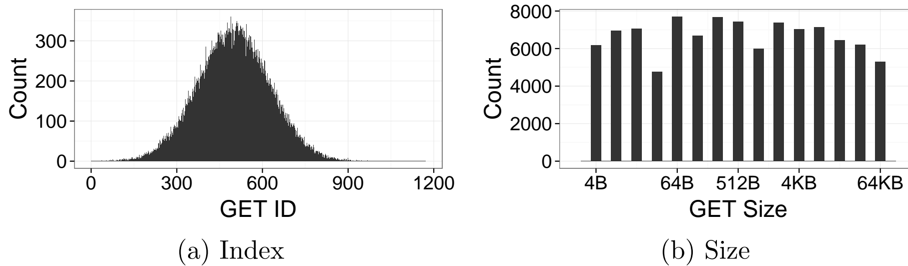
Algorithm 15 LCC

Input V_i, A_i ◁ V_i is the partition of V assigned to process p_i; A_i is the set of $\Gamma_G(v)$ for $v \in V_i$
Output $LCC(v) \quad \forall v \in V_i$
1: $LCC(v) = 0 \quad \forall v \in V_i$
2: **for** $j = 0$ to $|V_i| - 1$ **do**
3: $\quad v = V_i[j]$ ◁ v is the $j - th$ vertex in V_i
4: $\quad \Gamma_G(v) = \texttt{Copy}(A_i(v))$
5: $\quad$ **for** $j = 0$ to $\rho(v)$ **do**
6: $\quad\quad u = \Gamma_G(v)[j]$
7: $\quad\quad$ **if** $p(u) == p_i$ **then** ◁ $p(u)$ returns the process owner of u
8: $\quad\quad\quad adj(u) = \texttt{Get}(p(u), u)$ ◁ Get the $adj(u)$ from process $p(u)$
9: $\quad\quad$ **else** ◁ process i is the owner of vertex u
10: $\quad\quad\quad adj(u) = \texttt{Copy}(A_i(u))$
11: $\quad\quad$ **end if**
12: $\quad$ **end for**
13: $\quad LCC(v) = \texttt{ComputeLCC}(\Gamma_G(v), adj(u))$ ◁ Computing $LCC(v)$ according to Eq. 5.1.1
14: **end for**

5.5 Experimental Results

The benchmarks presented in this section are executed on a Cray XC30 system composed of 5,272 nodes. Each compute node is equipped with an eight-core Intel Xeon E5-2670 clocked at 2.60GHz. The system is interconnected with Cray's Aries network arranged in a Dragonfly topology. We use the optimized, open-source foMPI [GBH13] implementation of the MPI standard in order to enable a comparison with respect to the fastest available RMA implementation on the targeted system. All libraries and benchmarks

discussed in this section are compiled using the Cray Programming Environment 5.2.82. We use the Berkeley UPC compiler 2.22 for UPC benchmarks and GNU gcc 4.3.4 for applications requiring features not supported by the the Cray C compiler. We use the libLSB timing library [HB15] in order to have high-resolution measurements.



(a) Index (b) Size

Figure 5.6: *Gets* (a) and of the data sizes (b) distributions.

5.5.1 Micro-Benchmarks

In this section we compare the performance of the proposed caching layer with respect to the classic non-cached approach. Moreover, in Section 5.5.1.3 we show an evaluation of different strategies for the victim selection. The following discussions are based on a micro-benchmark consisting in two processes mapped on different physical nodes, namely *initatior* and *target*. The initiator creates a sequence of *get* operations to be executed on the area of memory exposed by the target. The sequence of *get*s is created in the following way: 1) a sequence G of $N = 10K$ non-overlapping (i.e., distinct) *get*s is created. Two *get*s are non-overlapping if they have different target or the same target but different window displacement. The size (in bytes) of each *get* in such sequence is randomly chosen in the set $S = \{2^i | i = 0..16\}$ according to an uniform distribution. 2) a sequence G_r of $Z \geq N$ *get*s is created sampling from G. The sampling is done according to a normal distribution $\mathcal{N}(\frac{N}{2}, \frac{N}{4})$. We decide to adopt a normal distribution in order to create a sequence in which a subset of *get*s is more frequent respect to the others.

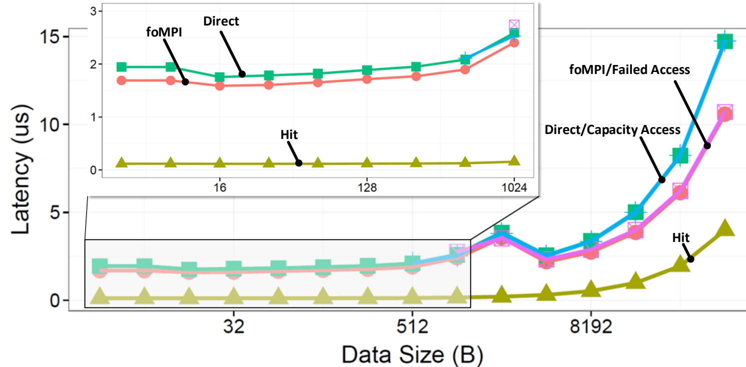


Figure 5.7: Cached and non-cached *get* costs comparison.

Figure 5.6 shows the details about how the *get*s are distributed in G_r. In particular, Figure 5.6a shows the distribution of the *get*s in G_r. The *GET ID* identifies the *get* in the G sequence. Figure 5.6b shows the distribution of the data sizes that is, as expected, still uniform.

5.5.1.1 Caching Costs Characterization

Figure 5.7 presents a characterization of the overheads introduced by CLaMPI. We compare the latencies of the different access types with the one presented by non-cached operations. In this benchmark we use a sequence G_r of size $Z = 20K$. The *hitting* access presents always a lower latency respect to the non-cached access, since it avoids to move data over the network. We can observe a difference in the latency ranging from a factor of 15x to a factor of 2.6x according with the requested data size, $4B$ and $64KB$ respectively. A performance breakdown showing the various overheads introduced by different access types is reported in Figure 5.8. It is worth noting how, increasing the data size, the data copy cost starts playing an important role, taking up to the 33% of the total latency in the $64KB$ case. This cost, in fact, cannot be overlapped with the *get* operation itself, since the copy in the cache memory buffer can start only at the end of the issuing epoch

(i.e., after the `MPI_Flush`). Despite this phenomena, the latency is always dominated by the `MPI_Flush` operation, that takes into account the network latency and finalizing overhead. Since this cost can be partially overlapped with computation, in Figure 5.9 we study the portion of the latency that can be overlapped according with different accesses.

As in the previous cases, our reference curve is *foMPI* (which provides non-cached accesses), that achieves the maximum overlap. The proposed caching layer, that is built on top of the MPI standard, can achieve at most the overlap presented by the former. In particular, from Figure 5.9 we can observe how the *direct* and *capacity* accesses provide the same overlap in most of the cases, since they are both dominated by the data copy phase. The *failing* access, instead, is able to provide a higher overlap for larger data sizes, since it does not require any additional data copy.

5.5.1.2 Adaptive Parameter Selection Evaluation

In Section 5.3.5.1 we discuss an adaptive strategy in order to adjust the sizes of $I_{\bar{w}}$ and $S_{\bar{w}}$ at runtime. This adjustment is made according to a set of indicators about the performance provided by the caching layer. In this section we evaluate such adaptive strategy showing the completion time of the micro-benchmark as function of different hash table sizes. In the adaptive case, the value of the hash table size is the starting value that could be dynamically adjusted. We can notice that, in the case in which the hash

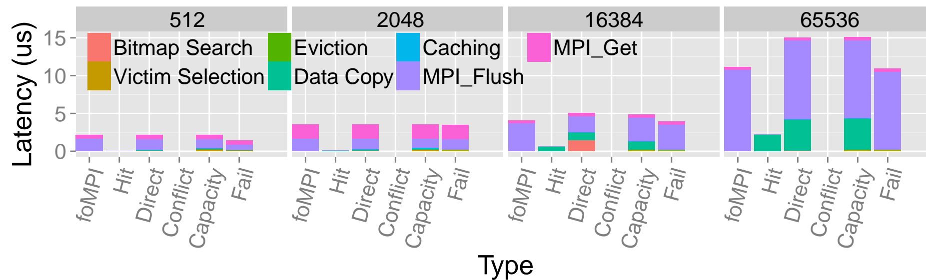


Figure 5.8: Performance breakdown per different access types.

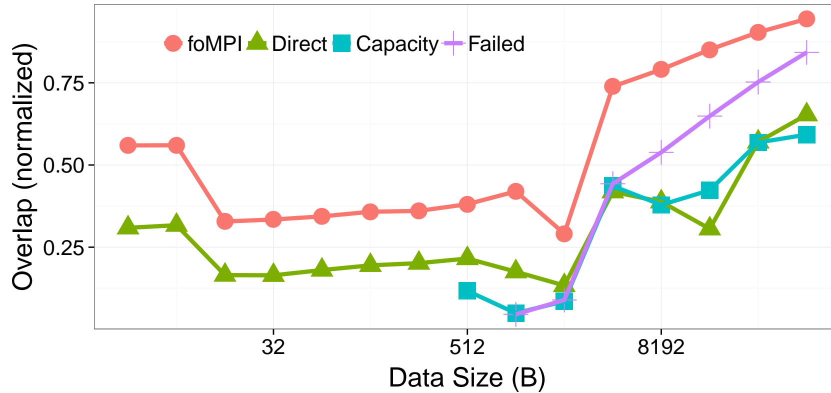


Figure 5.9: Cached and non-cached *get* costs comparison.

table size is set to 100 entries, the effects of the caching are null due to the high number of conflicting accesses. This does not happen with the adaptive strategy, which is able to adjust the hash table size at runtime.

5.5.1.3 Victim Selection Algorithm Evaluation

In Section 5.3.4.1 we propose a victim selection scheme that is based on the LRU algorithm with the addition that it also aims to reduce the external fragmentation in $S_{\bar{w}}$. In particular, every cache entry is associated with two scores: temporal and positional. The proposed victim selection scheme, referred as *Full* scheme, utilizes these two scores in order to select a victim. In this section we evaluate such scheme, comparing it to the cases in which the entry score is composed only from the temporal (i.e., LRU-like) or only from the positional score. We refer to these two selection schemes as *Temporal* and *Positional*, respectively.

In this experiment, the total number of issued gets is set to $Z = 100K$ in order to observe the effects of the external fragmentation on a longer term. Each get is associated with *Get Sequence ID*, that identifies the operation in the ordered issuing sequence. In this benchmark we are interested in studying different eviction schemes in the case of *capacity* or *failed* accesses. We choose to not include the *conflicting* accesses in this analysis since, in

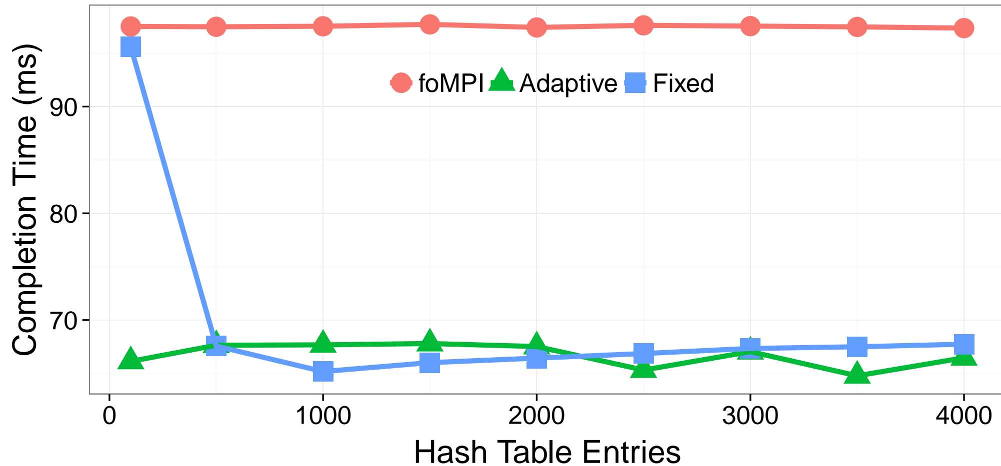


Figure 5.10: Synthetic benchmark completion time as function of hash table entries. The number of hash table entries represents the starting value for the *adaptive* strategy.

that case, the victim selection has to be performed among a limited number of entries (i.e., the conflicting ones), hence it can be considered a special case of the *capacity* access. Figure 5.11 presents a characterization of the accesses, showing that when the size of the hash table approaches to the number of distinct gets (i.e., $1K$) the number of *conflicting* accesses starts dropping. In the following analysis we consider only hash table sizes greater or equal of $1.5K$ entries.

Figure 5.12 reports the fraction of occupied space in the memory buffer as function of the *Get Sequence ID*. In particular we start reporting measurements once the buffer is completely filled for the first time (i.e., the first *capacity/failed* access takes place). Such measure gives us an estimation of the external fragmentation: fixed a *Get Sequence ID*, we have higher fragmentation if we observe a lower fraction of occupied space. As expected, the external fragmentation evolves with the issuing of subsequent get operations. It is possible to notice how the *Temporal* selection scheme leads to an

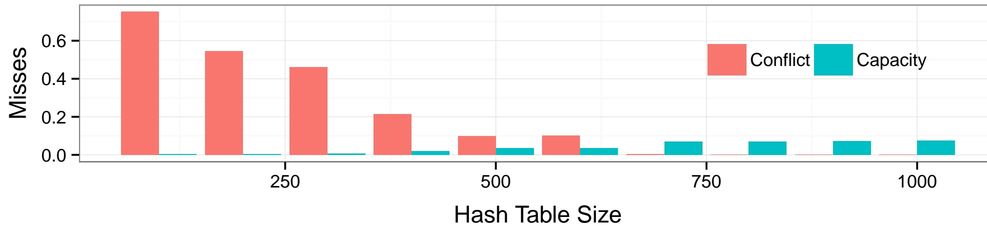


Figure 5.11: Conflict and capacity misses as function of the hash table size. The misses are normalized to the total number of issued gets.

increased fragmentation, while the *Full* and *Positional* schemes are able to keep the storage usage around the 90% of its capacity.

Figure 5.13 reports statistics as function of the hash table size. In particular, on the top we show the average number of visited entries per *capacity* access (that is the v_i described in Section 5.3.4 averaged on all the *capacity/-failed* accesses). The *sample_size* for this experiment is fixed to 16. However, it is possible to notice how the average number of visited entries grows due to the increased sparsity of the *entry_array* with larger hash table sizes. On the middle we show the hit ratio presented by different victim selection strategies. The *Full* scheme is able to provide the best hit ratio for all the considered hash table sizes. This is explained by the fact that such scheme is able to store an higher number of cache entries respect to the *Positional* one, due to the reduced external fragmentation. Moreover, it takes into account also the hotness of the entries while selecting the victim, hence it privileges the entries that are most likely reused by the application. On the bottom we show the average free space per different victim selection scheme. As expected, the *Temporal* scheme is the one presenting the higher free space (hence higher external fragmentation). We also show the portion of non-empty entries that are visited during the victim selection. The higher the hash table size, the higher the sparsity in *entry_array*, the lower the number of entries among which the victim is selected.

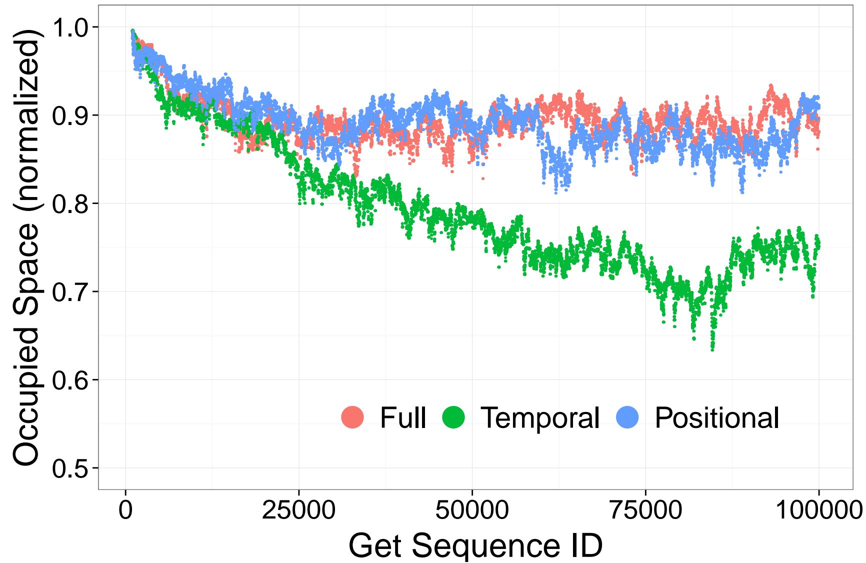


Figure 5.12: Storage space occupation as function of the *Get Sequence ID* and the employed victim selection scheme. Hash table size: $1.5K$ entries.

5.5.2 LCC

In the following section, we evaluate the performance of the implementation of the Algorithm 15, comparing configurations of CLaMPI with respect to the non-caching enabled implementation. Input instance are created with the R-MAT random graph generation algorithm [CZF04]. We generate scale-free graphs which are used to model real-world networks. The edge factor parameter (EF) defines the number of edges as $2^{\mathsf{S}} \times EF$. Our goal is to understand how well our solution exploits the data locality and, as a consequence, the communication performances of the Algorithm 15.

As discussed in Section 5.3.5, ClaMPI exposes two performance critical parameters that are $|I_{\bar{w}}|$ and $|S_{\bar{w}}|$. In order to evaluate the effects of such parameters on the LCC computation we use a R-MAT graph with $S = 20$ and $EF = 16$ distributed over $P = 32$ processes. Figure 5.14 reports the speedup of different configurations with respect to the non-caching enabled

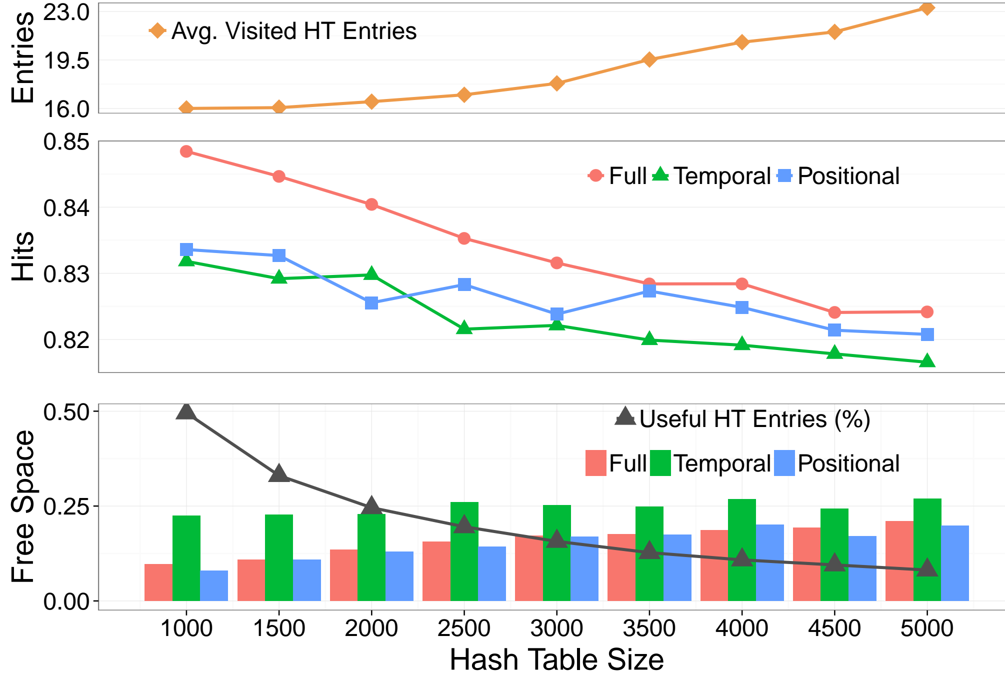


Figure 5.13: Completion time of *gets* sequence.

MPI implementation (i.e., foMPI). In particular, we test both the adaptive and fixed strategies for different values of $|S_{\bar{w}}|$ and $|I_{\bar{w}}|$. Such parameters are used as starting value by the adaptive strategy. We can notice the ClaMPI with fixed strategy and $|S_{\bar{w}}| = 64MB$ shows the worst performance among the analyzed cases. The statistics presented in Figure 5.15 show how such configuration leads to a significant number of *capacity*/*failed* accesses (i.e., $\sim$ 60% of the issued *gets*). Increasing the memory buffer size to $128MB$ the number of *capacity*/*failed* accesses decreases to less the 5% of the total issued *gets*. In both cases the *conflicting* accesses becomes lower than the 1% when the $|I_{\bar{w}}|$ is set to $256K$ entries. This explains the up to a factor of 3x speedup presented by this last configuration.

The adaptive strategy achieves a similar speedup presented by the best fixed configuration. From Figure 5.15b it is possible to notice how the adap-

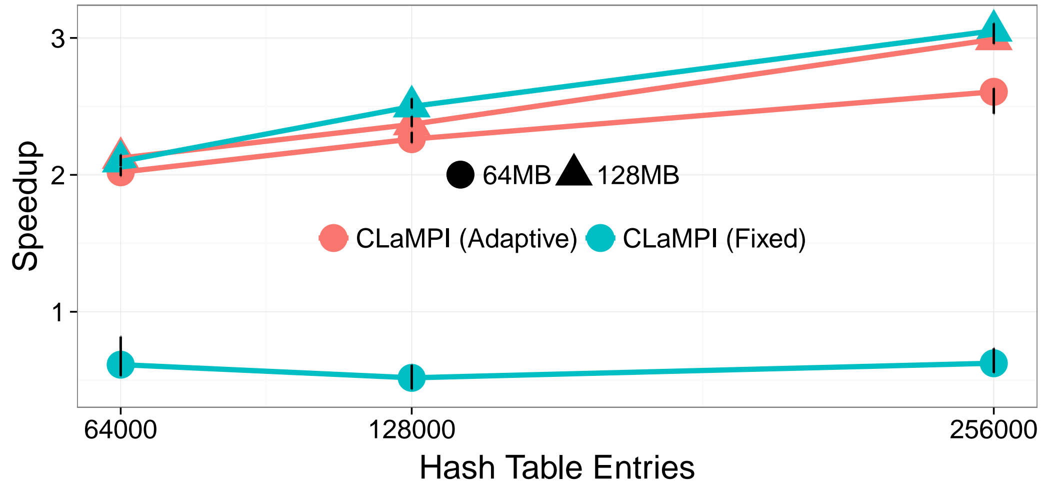
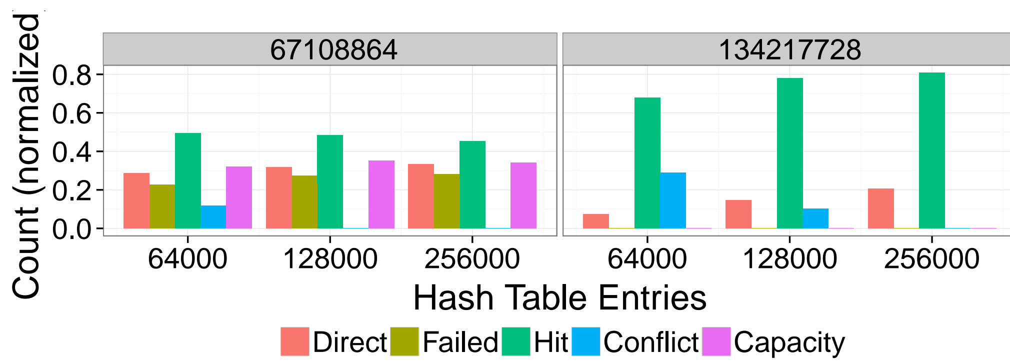


Figure 5.14: LCC communication time for a R-MAT graph $S = 20$ and $EF = 16$. Number of processes: 32. Measures for the fixed and adaptive configurations with varying $|I_{\bar{w}}|$ and $|S_{\bar{w}}|$ are reported. The vertical black bar denotes the confidence interval.

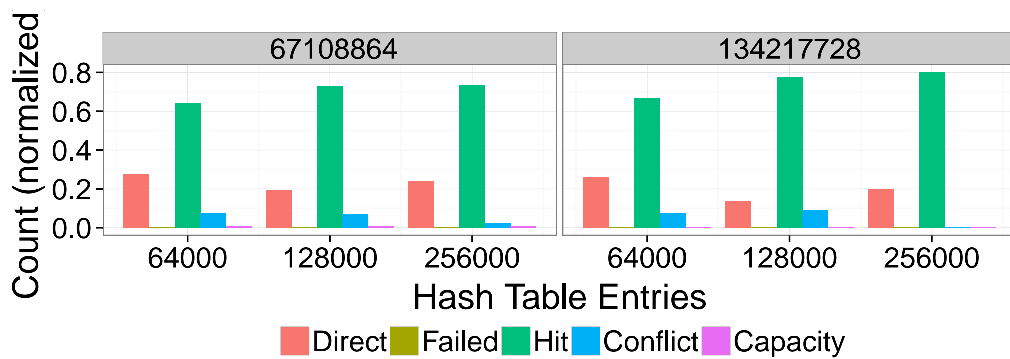
tive strategy is able to keep the number of *hitting* access always above the 60% of the issued *get*s. The differences in the speedup achieved starting from different values of $I_{\bar{w}}$ and $S_{\bar{w}}$ are explained by the number of adjustments (with consequent cache invalidation) needed to keep the *capacity/failing* and *conflicting* accesses under the specified threshold. In all the cases the adaptive strategy converges to the values of $144K$ entries and $128MB$ for $I_{\bar{w}}$ and $S_{\bar{w}}$, respectively.

5.5.2.1 Weak Scaling

In the weak scaling experiment, the problem size per processing elements (i.e., process) stays constant. We set $|I_{\bar{w}}| = 128K$ entries and $|S_{\bar{w}}| = 128MB$. The input graph is an R-MAT with $EF = 16$. The scale varies from $S = 19$ with 16 process to $S = 22$ with 128 processes. The experiment results are reported in Figure 5.16. Increasing the graph scale with the number



(a) Fixed.



(b) Adaptive.

Figure 5.15: LCC cache statistic for a R-MAT graph $S = 20$ and $EF = 16$. Number of processes: 32. Measures for the fixed (a) and adaptive (b) configurations are reported.

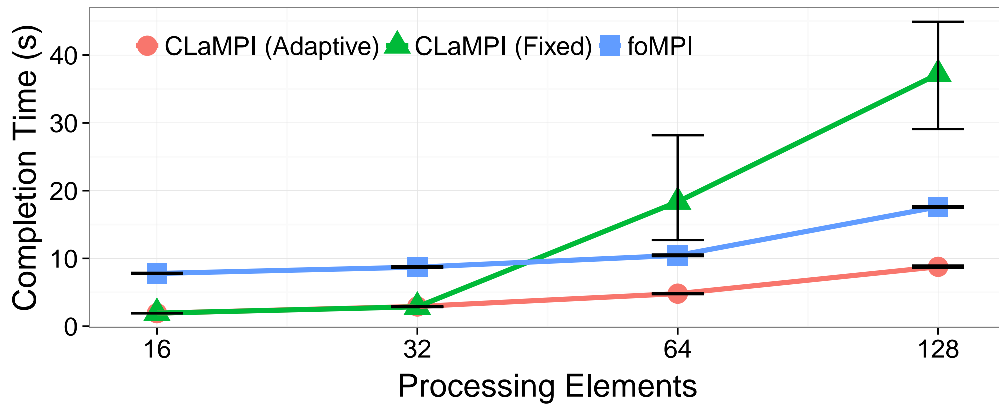
of processing elements leads to a situation in which the number of *get*s per process stays constant but the average *get* size increases. As a consequence, the fixed strategy suffers of an higher number of *capacity*/*failed* accesses when increasing the number of processing elements. This explains the poor scalability presented by this configuration when $P > 32$. On the contrary, the adaptive strategy is able to resize $S_{\bar{w}}$ in order to accommodate the larger *get*s. This allows this configuration to achieve the same scalability of the foMPI one, keeping a speedup of ~ 1.8x with respect to it. Therefore in this case we expect at beginning to observe the cases in which the cache size is enough to guarantee a limited number of evictions. By increasing the size of the graph, at some point, the size of the gets quickly fill the memory up.

5.5.2.2 Strong Scaling

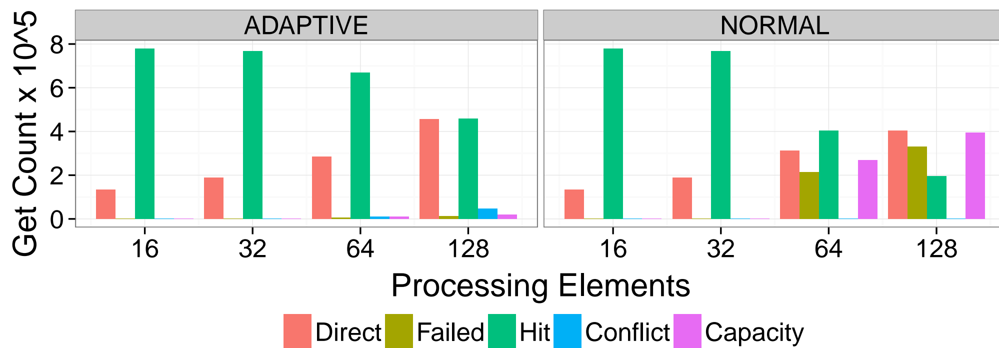
Finally, a different scenario is figured out by strong scaling experiment (i.e., scaling with a fixed-size input). In the strong scaling experiment the global problem size is kept constant, while the number of processing elements varies. By increasing the number of processes, the number of vertices assigned to them decreases, as well as the probability to get the adjacency of a local vertex. Let Tr be the total number of requests of vertex adjacencies (remote or local). This value depends on the degree of the vertices assigned to each process according to the adopted one-dimensional partitioning scheme. The study of the strong scaling allows evaluating the communication cost by increasing the ratio between the number of remote requests (Rr) and Tr.

We expect that increasing the number of processes, the value of Rr decreases. This is confirmed by the results showed by Figure 5.17b: the total number of remote *get*s Rr is the sum of all reported accesses and it decreases from $\sim 1.7M$ with 16 processes to $\sim 400K$ when $P = 128$.

In Figure 5.17a, we report the communication time (in seconds) for the both the discussed versions of the LCC algorithm. In particular we set $|I_{\bar{w}}| = 128K$ entries and $|S_{\bar{w}}| = 128MB$ as parameters for both the fixed and the adaptive strategies. We can observe that both the ClaMPI configurations are



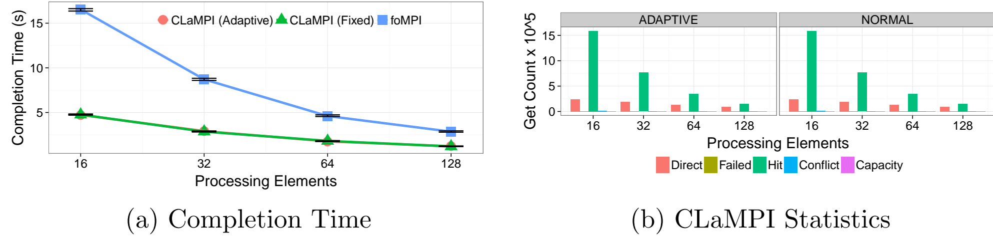
(a) Completion Time



(b) CLaMPI Statistics

Figure 5.16: (a) LCC weak scaling experiment starting with R-MAT graph ranging from $S = 19$ to $S = 22$ and $EF = 16$. (b) CLaMPI statistics for both *fixed* and *adaptive* strategies.

generally faster than foMPI. However the gap decreases due to the decreasing number of issued *get*s and hence the decreasing benefit of the caching.



(a) Completion Time (b) CLaMPI Statistics

Figure 5.17: (a) LCC strong scaling experiment with R-MAT graph $S = 20$, $EF = 16$. (b) CLaMPI statistics for both *fixed* and *adaptive* strategies.

5.6 Summary and Discussion

In this chapter we presented a new asynchronous distributed algorithm for Local Clustering Coefficient computation. We implemented the algorithm in two different ways exploiting MPI One-sided primitives. The second one we proposed is based on a new caching system of RMA get operations and is 2.5× faster than the baseline implementation. The idea behind is to exploit the data reuse that is hard to predict on irregular applications (e.g., graph processing), introducing a transparent caching layer between the application and the network. The proposed designed can be easily integrated in the MPI standard requiring minimal modifications. In particular, the proposed design enables a fully associative caching system where cache entries are indexed with a hash table. Collisions are resolved adopting the Cuckoo scheme. In order to tackle the external fragmentation induced by the variable-size cache entries, the eviction procedure takes also in account the so called *positional* score of cached entries during the victim selection. The *positional* score gives an estimation of how a cache entry contributes to the current fragmentation status. To evaluate the performance of the proposed caching system, we present a set of micro-benchmarks to show the overheads introduced by the

various access types.

Chapter 6

Conclusions

The present thesis investigates the problem of improving graph analytics performances with a particular emphasis on large-scale instances. In detail, we aim at designing new algorithms that allow to increase the parallel scalability and reduce the time to solution. To this aim, we focus on a) avoiding computations, b) achieving load-balancing among computational resources, c) improving memory usage, and d) exploiting massively parallel systems. In particular, the thesis provides novel solutions for executing fast and scalable graph analytics on parallel and distributed systems, especially composed by GPUs. Following hardware/software co-design, the proposed solutions tackle different aspects, ranging from architecture –i.e, GPU and data-thread mapping strategies– and runtime systems –i.e., caching layer– to novel algorithm design –i.e., Maximal Frontier Betweenness Centrality Algorithm– passing through heuristics. By encapsulating such solutions we provide efficient parallel implementations that in most of the cases overcome state-of-the-art solutions. To summarize the results concerning *st*-connectivity, we show how to solve efficiently memory contention on GPUs in order to speed-up bi-directional based algorithms. Performances depend on the efficiency of atomic operations provided by the GPU architecture. Our solutions are able to solve 340 ST-CON problems per second on a graph having about two million vertices and 32 million edges. Furthermore this study poses the foun-

dation for the design of new Breadth-first search (BFS) algorithms based on multi-source starts. More complex problems, such as Betweenness Centrality, require more sophisticated techniques. The combination of an efficient and scalable algorithm, multi-level parallelism and heuristics allow to compute the exact betweenness score of unweighted graphs composed by hundreds millions edges in few hours, outperforming existing solutions. On weighted graphs, we provide our new maximal frontier algorithm for betweenness centrality. It achieves good parallel scaling due to its low theoretical communication complexity and a robust implementation of its primitive operations. For graphs with n vertices and average degree k, we show that, on p processors, MFBC performs a factor of $p^{1/3}$ less communication than known alternatives when $k = n/p^{2/3}$. MFBC with CTF also outperforms existing algebraic-based solutions by factors of up to 8. The automatic parallelism we provide for sparse tensor contractions with arbitrary algebraic structures may be used for the solution of many other graph problems. The rigorous communication-efficiency achieved by CTF for these general primitives has a promising potential for changing the way in which massively-parallel code is developed. Furthermore, as for the betweenness centrality, we also provide a graph-centric formulations as well as linear-algebra-centric ones on distributed system. Finally, we introduce a new asynchronous algorithm for the computation of the Local Clustering Coefficient. The algorithm exhibits more data-reusing than traditional existing algorithms. We demonstrate how our caching system improves the performance of the algorithm achieving a speed-up factor of 1.8× and 5×.

Although we have acquired a good understanding of the landscape of parallel graph analytics, the increasing size of the data requires further efforts to build efficient frameworks, particularly on non-traditional and emerging architectures. Within this context, in the future we aim at studying novel techniques to exploit new architectures and their capabilities (i.e, 3D memories). Concerning distributed memory systems, it is worth to investigate if RMA technology is mature enough to design efficient distributed (and possi-

bly asynchronous) algorithms. At the same time, from an algorithmic point of view, we have seen how preprocessing techniques and graph topology manipulation impact on algorithm performance. Along this way, we are also investigating more sophisticated solutions for compressing high-connected vertices (i.e, hub vertices). Finally, we are also going to reduce the gap between graph-centric and linear-algebra-centric formulations (see for example GraphBLAS effort [grab]). Indeed, linear algebra frameworks for graph analytics do not yet provide primitives which allow to exploit topology characteristics of the graphs. Towards exascale high-performance computation, we finally believe that the hardware-software codesing is the way to overcome the challenges that graph analysis poses every day.

6.1 My Publications

The work described in the present dissertation has been published in the following papers[a,b]:

- Flavio Vella, Giancarlo Carbone, and Massimo Bernaschi. Algorithms and heuristics for scalable betweenness centrality computation on Multi-GPU systems. **Submitted** to IEEE Transactions on Parallel and Distributed Systems (TPDS), arXiv preprint arXiv:1602.00963, 2016 [VCB16][a].

- S. Di Girolamo, F. Vella, T.Hoefler, "Transparent Caching for RMA Systems", **Accepted** to the 31st IEEE International Parallel & Distributed Processing Symposium (IPDPS)[a].

- E. Solomonik, M. Besta, F. Vella, T.Hoefler, "Betweenness Centrality is more Parallelizable than Dense Matrix Multiplication", **Submitted** to the 22nd ACM SIGPLAN Symposium on Principles and Practice

[a]The PhD candidate has mainly or equally contributed.
[b]The PhD candidate has partially contributed.

of Parallel Programming, arXiv preprint arXiv:1609.07008 [SBVH16], 2016[b].

- M. Bernaschi, G. Carbone, F. Vella, "Scalable betweenness centrality on multi-GPU systems". In Proceedings of the ACM International Conference on Computing Frontiers, 2016, [BCV16][a].

- M. Bernaschi, G. Carbone, F. Vella, "Betweenness centrality on Multi-GPU systems." In Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms. ACM, 2015 [BCV15][a].

- M. Bernaschi, G. Carbone, E. Mastrostefano, F. Vella, "Solutions to the st-connectivity problem using a GPU-based distributed BFS". Journal of Parallel and Distributed Computing (JPDC), 2014, [BCMV15][a].

Summary

The present dissertation focuses on the parallel computation of several important properties on large-scale graphs, including reachability, betweenness centrality, and clustering coefficients by exploiting modern parallel architectures (i.e, GPUs) and low-latency networks. This approach requires a re-design of the underlying algorithms in order to take full advantage of the specific capabilities of the platforms deployed. It is well known that graph algorithms are extremely hard to be parallelized, since most of them are memory intensive and exhibit irregular and fine-grained memory access patterns that strongly depend on the structure of the graph.

More in detail, concerning reachability problem, we focus on ST-Connectivity (ST-CON) which consists in deciding if a vertex t is reachable from a vertex s of the graph. Such a problem can be naturally solved by traversing the graph starting from s and checking if the target vertex t is reached. A more efficient solution can exploit bi-directional search. However, that approach requires to solve memory contentions on the data structures. We address the problem by providing two scalable implementations which efficiently solve ST-CON on large-scale graphs.

Betweenness Centrality (BC) is steadily growing in popularity as a metrics useful for finding the most important nodes in a network. We propose two different distributed algorithms for the computation of the Betweenness Centrality on both unweighted and weighted graphs in order to analyze graphs that cannot be stored in a shared-memory system. As for unweighted graphs, our novel approach combines bi-dimensional (2-D) decomposition

and multi-level parallelism. Contrary to previous works, we provide a suitable data-thread mapping that overcomes most of the difficulties caused by the irregularity of the computation on GPUs regardless of specific characteristics of the graph. Then, we propose novel algorithms which exploit the topology information of the graph in order to reduce time and space requirements of BC computation. Experimental results on synthetic and large-scale real-world graphs show that the proposed techniques allow a significant reduction of the computing time. General infrastructure and scalable algorithms for sparse matrix multiplication also enable compact high-performance implementation of numerical methods and graph algorithms especially for weighted graphs. We showcase the theoretical and practical quality of novel sparse matrix multiplication routines in Cyclops Tensor Framework (CTF) via MFBC: a Maximal Frontier Betweenness Centrality algorithm. Our sparse matrix multiplication algorithms and consequently MFBC require asymptotically less communication than previous approaches. For graphs with n vertices and average degree k, we show that on p processors, MFBC requires a factor of $p^{1/3}$ less communication than known alternatives when $k = n/p^{2/3}$. We formulate and implement MFBC by leveraging specially-designed monoids and functions. We prove the correctness of the new formulation. CTF allows a parallelism-oblivious C++ implementation of MFBC to achieve good scalability for both extremely sparse and relatively dense graphs. The resulting code outperforms the well-known CombBLAS library by factors of up to 8 and shows more robust performance.

The constantly increasing gap between communication and computation performance emphasizes the importance of communication-avoidance techniques for graph analytics. Caching is a well-known concept used to reduce accesses to slow local memories. We extend the caching idea to MPI-3 Remote Memory Access (RMA) operations. Here, caching can avoid inter-node communications and achieve similar benefits for irregular applications as communication-avoiding algorithms. We propose CLaMPI, a caching library layered on top of MPI-3 RMA, to automatically optimize code with

minimum user's intervention. We demonstrate how cached RMA improves the performance of a new distributed algorithm for the computation of Clustering Coefficient. Our implementations based on MPI One-sided exhibit a high data-reusing and are well suitable for CLaMPI. Experiments show that CLaMPI-based implementations achieve a speed-up factor of $1.8\times$ and $5\times$ with respect to our baseline code. Due to the low overheads in the cache miss case and the potential benefits, we expect that our ideas around transparent RMA caching will soon be an integral part of many MPI libraries to design new generation of graph analytics.

List of Tables

1.1 Examples of algorithms used in graph analytics. 4

2.1 Summary of the symbols used in the dissertation. 13

3.1 Mean and variance of the length of shortest paths for three different instances of an R-MAT graph with different values of SCALE S. NP is the number of GPUs and d the diameter of the graph. For each instance, NE is the number of extracted vertices. 28

3.2 As in Table 3.1, we report the mean and variance of the length of the shortest paths for two real graphs. 28

3.3 Columns 2, 3, and 4 show, for different graphs, the percentage of vertices in the graph visited by the *naive*, *atomic*, and *no-atomic* implementations, respectively. Columns 4 and 5 specify the size of each graph in terms of SCALE and EF (for real datasets, SCALE and EF are approximated from the number of vertices and edges). Column 6 shows at which BFS level (MV-lvl) the matching vertex is found. The level is computed as the average with respect to 10000 random instances of the ST-CON problem for the same graph. 32

4.1 Features of the real-world graphs used for the tests. **d** represents the diameter whereas dg-1 and dg-2 represent the percentage of vertices having one and two neighbors respectively . 63

4.2 Comparison with other single GPU implementations on real-world graphs. The time reported is expressed in seconds. The acronym `nt` stands for "execution does not terminate". 64

4.3 Total time to compute exact BC for the Orkut graph with $fd = 2$. 68

4.4 Performance comparison between the BC implementation provided by CombBLAS and different MGBC configurations on an RMAT graph $S = 29$ and $EF = 16$. We report the time required to complete the BC computation from 128 source vertices. In MGBC, the number of concurrent vertices corresponds to fr. 68

4.5 Impact on BC processing time due to degree-1 reduction. The value reported in parenthesis are referred to MGBC with degree-1 off. 71

4.6 Impact of the heuristics on the exact Betweenness Computation on RoadNet-PA. The numbers in parenthesis represent the total number of vertices that may be computed by heuristics. 76

4.7 Symbols used in the section; $v, s, t \in V$ are vertices of a graph G 77

List of Figures

1.1 Visualization of the routing paths of the Internet. Figure made by the Opte Project (www.opte.org). CC Attribution-Noncommercial 4.0 International License [opt]. 2

1.2 Whasapp monthly active users [biw]. 4

3.1 Weak scaling plot of the number of ST-CON problems solved within a second (*NSTPS*) for the two implementations described in the text: *atomic-stcon* and *no-atomic-stcon*. For comparison, we also show the performance of the naive (single BFS) implementation. The SCALE of the R-MAT graph ranges from $21 - 27$ for $1 - 64$ GPUs, respectively, with EF equal to 16. 29

3.2 Weak scaling plot of *NSTPS* using the *atomic-stcon* implementation for three different values of EF. The SCALE of the R-MAT graph ranges from $19-26$ for $1-128$ GPUs, respectively. 30

3.3 Strong scaling plot of *NSTPS* using the *atomic-stcon* implementation. 31

3.4 Time breakdown for *no-atomic-stcon* and *atomic-stcon* on 16 GPUs. The scale of the R-MAT graph is equal to 25, and EF is equal to 16. 33

3.5 Computation and communication time breakdown for *no-atomic-stcon* and *atomic-stcon* for three graphs. The scale of the R-MAT and RANDOM graphs is equal to 25, and EF is equal to 16. The scale of the com-Orkut graph is approximately 22, and EF is approximately 38. 34

4.1 Example of Active-Edge Parallelism. 45

4.2 Overlapping of GPU - CPU data transfer with MPI communication. 50

4.3 Sub-clustering. On the left side the configuration ($p = 16$, $fd = 1$ and $fr = 1$) enables a pure fine-grained strategy. On the right side, a sub-cluster configuration with $p = 16$, $fd = 4$ and $fr = 4$. 52

4.4 Strong scaling experiments for R-MAT graphs with $S = 23$ and $EF = 16, 32$. 65

4.5 Strong scaling experiments for R-MAT graphs with $S = 23$ and $EF = 32$. 66

4.6 Strong scaling experiments for Twitter and Friendster graphs. 66

4.7 Weak Scaling Experiments 67

4.8 Impact of the prefix-sum optimization on single GPU system. 70

4.9 Impact of the overlapping on synthetic and real-world graphs. 70

4.10 Strong scaling experiment of the preprocessing algorithm for a R-MAT graph with $S = 23$ and $EF = 32$. 72

4.11 Impact of degree-1 reduction on a R-MAT graph with $S = 20$. The bars show the time in seconds of the computation (top), communication (middle) and overlap (bottom) respectively. . . 73

4.12 Heuristics comparison on RoadNet-PA. 75

4.13 Strong scaling of MFBC and CombBLAS for R-MAT graphs, which have roughly 2^S vertices and average degree E (weights are selected randomly between 1 and 100 for weighted graphs in Figure 4.13c) 99

4.14 Weak scaling of MFBC and CombBLAS for uniform random graphs 102

5.1 Difference of performance of local vs. remote accesses in RDMA environments. 110

5.2 Data Size distribution of a Local Clustering Coefficient (LCC) instance, averaged on 32 processing elements. R-MAT graph with S=16 and EF=16. 113

5.3 Operational Modes for Caching-Enabled Windows 116

5.4 Cache entry state diagram. The initial state of any cache entry is `MISSING`. 120

5.5 Employed data structures. The information related for the score computation of an entry x (i.e., l_x and d_x) are kept directly in the associated hash table entry. 124

5.6 *Gets* (a) and of the data sizes (b) distributions. 130

5.7 Cached and non-cached *get* costs comparison. 131

5.8 Performance breakdown per different access types. 132

5.9 Cached and non-cached *get* costs comparison. 133

5.10 Synthetic benchmark completion time as function of hash table entries. The number of hash table entries represents the starting value for the *adaptive* strategy. 134

5.11 Conflict and capacity misses as function of the hash table size. The misses are normalized to the total number of issued gets. 135

5.12 Storage space occupation as function of the *Get Sequence ID* and the employed victim selection scheme. Hash table size: $1.5K$ entries. 136

5.13 Completion time of *get*s sequence. 137

5.14 LCC communication time for a R-MAT graph $S = 20$ and $EF = 16$. Number of processes: 32. Measures for the fixed and adaptive configurations with varying $|I_{\bar{w}}|$ and $|S_{\bar{w}}|$ are reported. The vertical black bar denotes the confidence interval. 138

5.15 LCC cache statistic for a R-MAT graph $S = 20$ and $EF = 16$. Number of processes: 32. Measures for the fixed (a) and adaptive (b) configurations are reported. 139

5.16 (a) LCC weak scaling experiment starting with R-MAT graph ranging from $S = 19$ to $S = 22$ and $EF = 16$. (b) CLaMPI statistics for both *fixed* and *adaptive* strategies. 141

5.17 (a) LCC strong scaling experiment with R-MAT graph $S = 20$, $EF = 16$. (b) CLaMPI statistics for both *fixed* and *adaptive* strategies. 142

Bibliography

[AAC+10] B. Arimilli, R. Arimilli, V. Chung, S. Clark, W. Denzel, B. Drerup, T. Hoefler, J. Joyner, J. Lewis, J. Li, N. Ni, and R. Rajamony. The PERCS high-performance interconnect. In *Proceedings of the IEEE Symposium on High Performance Interconnects (HOTI'10)*, pages 75–82. IEEE Computer Society, 2010.

[ABB+15] Ariful Azad, Grey Ballard, Aydin Buluc, James Demmel, Laura Grigori, Oded Schwartz, Sivan Toledo, and Samuel Williams. Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication. *arXiv preprint arXiv:1510.00844*, 2015.

[ABC+06] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. The landscape of parallel computing research: A view from berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[ABG+95] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar. A three-dimensional approach to parallel matrix multiplication. *IBM J. Res. Dev.*, 39:575–582, September 1995.

[ABG15] Ariful Azad, Aydin Buluç, and John Gilbert. Parallel triangle counting and enumeration using matrix algebra. In *Parallel and*

Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International, pages 804–811. IEEE, 2015.

[ACS90] Alok Aggarwal, Ashok K. Chandra, and Marc Snir. Communication complexity of PRAMs. *Theoretical Computer Science*, 71(1):3 – 28, 1990.

[APPB10] V. Agarwal, F. Petrini, D. Pasetto, and D.A. Bader. Scalable Graph Exploration on Multicore Processors. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1 –11, nov. 2010.

[ASA16] Seher Acer, Oguz Selvitopi, and Cevdet Aykanat. Improving performance of sparse matrix dense matrix multiplication on large-scale parallel systems. *Parallel Computing*, 2016.

[BA99] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[BAP12] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing Breadth-first Search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 12:1–12:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[BAP13] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. *Scientific Programming*, 21(3-4):137–148, 2013.

[BBCG08] Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides Gionis. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 16–24. ACM, 2008.

[BBD+13] Grey Ballard, Aydin Buluc, James Demmel, Laura Grigori, Benjamin Lipshitz, Oded Schwartz, and Sivan Toledo. Communication Optimal Parallel Multiplication of Sparse Random Matrices. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, pages 222–231, New York, NY, USA, 2013. ACM.

[BBM+15] Aydın Buluç, Scott Beamer, Kamesh Madduri, K Asanovic, and David Patterson. Distributed-memory breadth-first search on massive graphs. *CRC Press, Taylor-Francis*, 2015.

[BBM16] M. Bisson, M. Bernaschi, and E. Mastrostefano. Parallel distributed breadth first search on the kepler architecture. *IEEE Transactions on Parallel and Distributed Systems*, 27(7):2091–2102, July 2016.

[BCM+15] Massimo Bernaschi, Giancarlo Carbone, Enrico Mastrostefano, Mauro Bisson, and Massimiliano Fatica. Enhanced GPU-based Distributed Breadth First Search. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, CF '15, pages 10:1–10:8, New York, NY, USA, 2015. ACM.

[BCMV15] Massimo Bernaschi, Giancarlo Carbone, Enrico Mastrostefano, and Flavio Vella. Solutions to the st-connectivity problem using a GPU-based distributed BFS. *Journal of Parallel and Distributed Computing*, 76:145–153, 2015.

[BCV15] Massimo Bernaschi, Giancarlo Carbone, and Flavio Vella. Betweenness centrality on multi-GPU systems. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, IA3 '15, pages 12:1–12:4, New York, NY, USA, 2015. ACM.

[BCV16] Massimo Bernaschi, Giancarlo Carbone, and Flavio Vella. Scalable betweenness centrality on multi-gpu systems. In *Proceed-*

ings of the ACM International Conference on Computing Frontiers, pages 29–36. ACM, 2016.

[BE05] Ulrik Brandes and Thomas Erlebach. *Network analysis: methodological foundations*, volume 3418. Springer Science & Business Media, 2005.

[Ber89] Jarle Berntsen. Communication efficient matrix multiplication on hypercubes. *Parallel Computing*, 12(3):335–342, 1989.

[BG11] Aydın Buluç and John R Gilbert. The combinatorial blas: Design, implementation, and applications. *International Journal of High Performance Computing Applications*, page 1094342011403516, 2011.

[BG12] Aydin Buluç and John R. Gilbert. Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments. *SIAM Journal on Scientific Computing*, 34(4):C170–C191, 2012.

[BGPL12] Miriam Baglioni, Filippo Geraci, Marco Pellegrini, and Ernesto Lastres. Fast exact computation of betweenness centrality in social networks. In *Proceedings of the 2012 International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2012)*, pages 450–456. IEEE Computer Society, 2012.

[BH15] Maciej Besta and Torsten Hoefler. Accelerating irregular computations with hardware transactional memory and active messages. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, pages 161–172, New York, NY, USA, 2015. ACM.

[BHM11] D Bader, C Heitsch, and Kamesh Madduri. Large-scale network analysis. *Graph Algorithms in the Language of Linear Algebra,*

J. Kepner and J. Gilbert, Eds. Philadelphia, PA: SIAM, pages 253–285, 2011.

[biw] Whatsapp is targeting snapchat's potential market with its latest update. http://uk.businessinsider.com/whatsapp-is-targeting-snapchats-potential-market-with-its-latest-update-2016-10. Accessed: 2016-10-06.

[BKMM07] David A Bader, Shiva Kintali, Kamesh Madduri, and Milena Mihail. Approximating betweenness centrality. In *International Workshop on Algorithms and Models for the Web-Graph*, pages 124–137. Springer, 2007.

[BM06] David A Bader and Kamesh Madduri. Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray mta-2. In *2006 International Conference on Parallel Processing (ICPP'06)*, pages 523–530. IEEE, 2006.

[BM11] Aydin Buluc and Kamesh Madduri. Parallel breadth-first search on distributed memory systems. *SC '11 Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[BM13] Aydın Buluç and Kamesh Madduri. Graph partitioning for scalable distributed graph computations. *Cont. Math*, 588, 2013.

[BMS+13] Aydin Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent advances in graph partitioning. *CoRR*, abs/1311.3144, 2013.

[BN16] Michele Borassi and Emanuele Natale. KADABRA is an adaptive algorithm for betweenness via random approximation. In *24th Annual European Symposium on Algorithms, ESA 2016, August 22-24, 2016, Aarhus, Denmark*, pages 20:1–20:18, 2016.

[BNP12] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. A quantitative study of irregular programs on GPUs. In *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, pages 141–151. IEEE, 2012.

[BO04] Albert-Laszlo Barabasi and Zoltan N Oltvai. Network biology: understanding the cell's functional organization. *Nature reviews genetics*, 5(2):101–113, 2004.

[BR91] Greg Barnes and Walter L Ruzzo. Deterministic algorithms for undirected s-t connectivity using polynomial time and sublinear space. In *Proceedings of the twenty-third annual ACM symposium on Theory of computing*, pages 43–53. ACM, 1991.

[Bra01] Ulrik Brandes. A faster algorithm for betweenness centrality*. *Journal of Mathematical Sociology*, 25(2):163–177, 2001.

[BS94] Stephen T Barnard and Horst D Simon. Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. *Concurrency: Practice and experience*, 6(2):101–117, 1994.

[BS09] Ed Bullmore and Olaf Sporns. Complex brain networks: graph theoretical analysis of structural and functional systems. *Nature Reviews Neuroscience*, 10(3):186–198, 2009.

[Can69] Lynn Elliot Cannon. *A cellular computer to implement the Kalman filter algorithm*. PhD thesis, Bozeman, MT, USA, 1969.

[CC11] Shumo Chu and James Cheng. Triangle listing in massive networks and its applications. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 672–680. ACM, 2011.

[CDS03] Wei Chen, Jason Duell, and Jimmy Su. A software caching system for upc. *Memory*, 1:P2, 2003.

[CF12] Deepayan Chakrabarti and Christos Faloutsos. Graph mining: laws, tools, and case studies. *Synthesis Lectures on Data Mining and Knowledge Discovery*, 7(1):1–207, 2012.

[Cha10] Barbara Chapman. *Parallel Computing: From Multicores and GPU's to Petascale*, volume 19. IOS Press, 2010.

[CLR90] Thomas H Cormen, Charles E Leiserson, and Ronald L Rivest. *Introduction to algorithms*. MIT press, 1990.

[CP14] Fabio Checconi and Fabrizio Petrini. Traversing Trillions of Edges in Real Time: Graph Exploration on Large-Scale Parallel Machines. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 425–434, Washington, DC, USA, 2014. IEEE Computer Society.

[CPW+12] Fabio Checconi, Fabrizio Petrini, Jeremiah Willcock, Andrew Lumsdaine, Anamitra Roy Choudhury, and Yogish Sabharwal. Breaking the speed and scalability barriers for graph exploration on distributed-memory machines. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–12. IEEE, 2012.

[CW77] J Lawrence Carter and Mark N Wegman. Universal classes of hash functions. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 106–112. ACM, 1977.

[CW87] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 1–6. ACM, 1987.

[CWZ09] Luonan Chen, Rui-Sheng Wang, and Xiang-Sun Zhang. *Biomolecular networks: methods and applications in systems biology*, volume 10. John Wiley & Sons, 2009.

[CZF04] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *SDM*, volume 4, pages 442–446. SIAM, 2004.

[Den68] Peter J. Denning. The working set model for program behavior. *Commun. ACM*, 11(5):323–333, May 1968.

[DNM14] Mayank Daga, Mark Nutter, and Mitesh Meswani. Efficient breadth-first search on a heterogeneous processor. In *Big Data (Big Data), 2014 IEEE International Conference on*, pages 373–382. IEEE, 2014.

[DNS81] Eliezer Dekel, David Nassimi, and Sartaj Sahni. Parallel matrix and graph algorithms. *SIAM Journal on Computing*, 10(4):657–675, 1981.

[DV14] Niels Doekemeijer and Ana Lucia Varbanescu. A survey of parallel graph processing frameworks. *Delft University of Technology*, 2014.

[DWM09] Yangdong Steve Deng, Bo David Wang, and Shuai Mu. Taming irregular EDA applications on GPUs. In *Proceedings of the 2009 International Conference on Computer-Aided Design*, pages 539–546. ACM, 2009.

[EHL10] Nick Edmonds, Torsten Hoefler, and Andrew Lumsdaine. A space-efficient parallel algorithm for computing betweenness centrality in distributed memory. In *High Performance Computing (HiPC), 2010 International Conference on*, pages 1–10. IEEE, 2010.

[ER59] P ERDdS and A R&WI. On random graphs i. *Publl. Math. Debrecen*, 6:290–297, 1959.

[EWHL10] Nicholas Edmonds, Jeremiah Willcock, T Hoefler, and A Lumsdaine. Design of a large-scale hybrid-parallel graph library. In *International Conference on High Performance Computing, Student Research Symposium, Goa, India*, 2010.

[FB15] Michael P Ferguson and Daniel Buettner. Caching puts and gets in a pgas language runtime. In *Partitioned Global Address Space Programming Models (PGAS), 2015 9th International Conference on*, pages 13–24. IEEE, 2015.

[FBR+12] Greg Faanes, Abdulla Bataineh, Duncan Roweth, Tom Court, Edwin Froese, Bob Alverson, Tim Johnson, Joe Kopnick, Mike Higgins, and James Reinhard. Cray Cascade: A scalable HPC system based on a Dragonfly network. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'12)*, pages 103:1–103:9. IEEE Computer Society, 2012.

[FDB+14] Zhisong Fu, Harish Kumar Dasari, Bradley Bebee, Martin Berzins, and Bryan Thompson. Parallel breadth first search on gpu clusters. In *Big Data (Big Data), 2014 IEEE International Conference on*, pages 110–118. IEEE, 2014.

[FFF99] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. In *ACM SIGCOMM computer communication review*, volume 29, pages 251–262. ACM, 1999.

[FJF15] Lester Randolph Ford Jr and Delbert Ray Fulkerson. *Flows in networks*. Princeton university press, 2015.

[FM82] Charles M Fiduccia and Robert M Mattheyses. A linear-time heuristic for improving network partitions. In *Design Automation, 1982. 19th Conference on*, pages 175–181. IEEE, 1982.

[For12] MPI Forum. MPI. A Message-Passing Interface standard. Version 3.0. 2012.

[FR07] Daniel Fernholz and Vijaya Ramachandran. The diameter of sparse random graphs. *Random Structures & Algorithms*, 31(4):482–516, 2007.

[Fre77] Linton C Freeman. A set of measures of centrality based on betweenness. *Sociometry*, pages 35–41, 1977.

[GB13] Oded Green and David A Bader. Faster betweenness centrality based on data structure experimentation. *Procedia Computer Science*, 18:399–408, 2013.

[GBH13] R. Gerstenberger, M. Besta, and T. Hoefler. Enabling Highly-Scalable Remote Memory Access Programming with MPI-3 One Sided. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 53:1–53:12. ACM, 11 2013.

[GHTL14] William Gropp, Torsten Hoefler, Rajeev Thakur, and Ewing Lusk. *Using advanced MPI: Modern features of the message-passing interface*. MIT Press, 2014.

[GJS76] Michael R Garey, David S. Johnson, and Larry Stockmeyer. Some simplified np-complete graph problems. *Theoretical computer science*, 1(3):237–267, 1976.

[GLS99] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[GM88] Hillel Gazit and Gary L Miller. An improved parallel algorithm that computes the bfs numbering of a directed graph. *Information Processing Letters*, 28(2):61–65, 1988.

[GMB14] Oded Green, Lluís-Miquel Munguía, and David A Bader. Load balanced clustering coefficients. In *Proceedings of the first workshop on Parallel programming for analytics applications*, pages 3–10. ACM, 2014.

[graa] Graph500 benchmark.

[grab] GraphBLAS.

[gre] Green Graph500 benchmark.

[HB] Jared Hoberock and Nathan Bell. Thrust CUDA library (http://thrust.github.com/).

[HB15] T. Hoefler and R. Belli. Scientific Benchmarking of Parallel Computing Systems. In *Proceedings of the IEEE/ACM International Conference on High Performance Computing, Networking, Storage and Analysis*, 11 2015.

[HDT+15] Torsten Hoefler, James Dinan, Rajeev Thakur, Brian Barrett, Pavan Balaji, William Gropp, and Keith Underwood. Remote Memory Access Programming in MPI-3. *ACM Trans. Parallel Comput.*, 2(2):9:1–9:26, June 2015.

[HIT05] Ram Huggahalli, Ravi Iyer, and Scott Tetrick. Direct cache access for high bandwidth network i/o. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 50–59, Washington, DC, USA, 2005. IEEE Computer Society.

[HKM+88] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and

Michael J. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1):51–81, February 1988.

[HKOO11] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating cuda graph algorithms at maximum warp. In *ACM SIGPLAN Notices*, volume 46, pages 267–276. ACM, 2011.

[HN07] Pawan Harish and PJ Narayanan. Accelerating large graph algorithms on the GPU using cuda. In *High performance computing–HiPC 2007*, pages 197–208. Springer, 2007.

[HOO11] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. Efficient parallel graph exploration on multi-core cpu and gpu. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 78–88. IEEE, 2011.

[HR73] Laurent Hyafil and Ronald L Rivest. *Graph partitioning and constructing optimal decision trees are polynomial complete problems.* IRIA. Laboratoire de Recherche en Informatique et Automatique, 1973.

[HT13] Takaaki Hiragushi and Daisuke Takahashi. Efficient hybrid breadth-first search on gpus. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 40–50. Springer, 2013.

[JáJ92] Joseph JáJá. *An introduction to parallel algorithms*, volume 17. Addison-Wesley Reading, 1992.

[JLH+11] Yuntao Jia, Victor Lu, Jared Hoberock, Michael Garland, and John C Hart. Edge vs. node parallelism for graph centrality metrics. *GPU Computing Gems: Jade Edition*, pages 15–28, 2011.

[KAB+16] Jeremy Kepner, Peter Aaltonen, David A. Bader, Aydin Buluç, Franz Franchetti, John R. Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, Scott McMillan, José E. Moreira, John D. Owens, Carl Yang, Marcin Zalewski, and Timothy G. Mattson. Mathematical foundations of the graphblas. *CoRR*, abs/1606.05790, 2016.

[KDSA08] John Kim, Wiliam J. Dally, Steve Scott, and Dennis Abts. Technology-Driven, Highly-Scalable Dragonfly Topology. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 77–88, Washington, DC, USA, 2008. IEEE Computer Society.

[KG11] Jeremy Kepner and John Gilbert. *Graph algorithms in the language of linear algebra*, volume 22. SIAM, 2011.

[KGGK94] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to parallel computing: design and analysis of algorithms*, volume 400. Benjamin/Cummings Redwood City, CA, 1994.

[KK98] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[KKJ08] Gary J Katz and Joseph T Kider Jr. All-pairs shortest-paths for large graphs on the gpu. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 47–55. Eurographics Association, 2008.

[KL70] Brian W Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *Bell system technical journal*, 49(2):291–307, 1970.

[KLPM10] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *WWW '10: Proceedings of the 19th international conference on World wide web*, pages 591–600, New York, NY, USA, 2010. ACM.

[KMF+12] S. Kumar, A.R. Mamidala, D. A. Faraj, B. Smith, M. Blocksome, B. Cernohous, D. Miller, J. Parker, J. Ratterman, P. Heidelberger, Dong Chen, and B. D. Steinmacher-Burrow. PAMI: A parallel active message interface for the Blue Gene/Q supercomputer. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'12)*, pages 763–773. IEEE Computer Society, 2012.

[KRR+00] Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, Dandapani Sivakumar, Andrew Tompkins, and Eli Upfal. The web as a graph. In *Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–10. ACM, 2000.

[LCK+10] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. Kronecker graphs: An approach to modeling networks. *Journal of Machine Learning Research*, 11(Feb):985–1042, 2010.

[LGHB07] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(01):5–20, 2007.

[LHKK79] Chuck L. Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.

[LK15] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, Nov 2015.

[LNP16] Andrew Lenharth, Donald Nguyen, and Keshav Pingali. Parallel graph analytics. *Communications of the ACM*, 59(5):78–87, 2016.

[Lu01] Linyuan Lu. The diameter of random massive graphs. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 912–921. Society for Industrial and Applied Mathematics, 2001.

[MAB+10] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[Mad08] Kamesh Madduri. *A high-performance framework for analyzing massive complex networks*. ProQuest, 2008.

[MB13] Enrico Mastrostefano and Massimo Bernaschi. Efficient breadth first search on multi-GPU systems. *J. Parallel Distrib. Comput.*, 73(9):1292–1305, September 2013.

[MB14a] Adam McLaughlin and David A Bader. Revisiting edge and node parallelism for dynamic gpu graph analytics. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 1396–1406. IEEE, 2014.

[MB14b] Adam McLaughlin and David A Bader. Scalable and high performance betweenness centrality on the GPU. In *Proceedings of the International Conference for High Performance Computing,*

Networking, Storage and Analysis, pages 572–583. IEEE Press, 2014.

[MBBC07] Kamesh Madduri, David A Bader, Jonathan W Berry, and Joseph R Crobak. An experimental study of a parallel shortest path algorithm for solving large-scale graph instances. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, pages 23–35. Society for Industrial and Applied Mathematics, 2007.

[MEJ+09] Kamesh Madduri, David Ediger, Karl Jiang, David A Bader, and Daniel Chavarria-Miranda. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE, 2009.

[Mer15] Bruce Merry. A performance comparison of sort and scan libraries for GPUs. *Parallel Processing Letters*, 25(04):1550007, 2015.

[MGG12] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable GPU graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 117–128, New York, NY, USA, 2012. ACM.

[Mil67] Stanley Milgram. The small world problem. *Psychology today*, 2(1):60–67, 1967.

[MK07] Richard C Murphy and Peter M Kogge. On the memory access patterns of supercomputer applications: Benchmark selection and its implications. *IEEE Transactions on Computers*, 56(7):937–945, 2007.

[MT99] W. F. McColl and A. Tiskin. Memory-efficient matrix multiplication in the BSP model. *Algorithmica*, 24:287–297, 1999.

[NBGS08] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008.

[NBP13] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Data-driven versus topology-driven irregular computations on GPUs. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 463–474. IEEE, 2013.

[New01] Mark EJ Newman. Scientific collaboration networks. ii. shortest paths, weighted networks, and centrality. *Physical review E*, 64(1):016132, 2001.

[New03] Mark EJ Newman. The structure and function of complex networks. *SIAM review*, 45(2):167–256, 2003.

[New10] Mark Newman. *Networks: an introduction*. Oxford university press, 2010.

[NWS02] Mark EJ Newman, Duncan J Watts, and Steven H Strogatz. Random graph models of social networks. *Proceedings of the National Academy of Sciences*, 99(suppl 1):2566–2572, 2002.

[opt] Opte project: The internet 2015. http://www.opte.org/the-internet/. Accessed: 2016-09-30.

[PDZ00] Vivek Pai, Peter Druschel, and Willy Zwaenepoel. IO-lite: A unified I/O buffering and caching system. *ACM Trans. Comput. Syst.*, 18(1):37–66, February 2000.

[PEZ+15] Rami Puzis, Yuval Elovici, Polina Zilberman, Shlomi Dolev, and Ulrik Brandes. Topology manipulations for speeding between-

ness centrality computation. *Journal of Complex Networks*, 3(1):84–112, 2015.

[Pfi01] Gregory F Pfister. An introduction to the infiniband architecture. *High Performance Mass Storage and Parallel I/O*, 42:617–632, 2001.

[PHV+13] Sreeram Potluri, Khaled Hamidouche, Akshay Venkatesh, Devendar Bureddy, and Dhabaleswar K Panda. Efficient inter-node mpi communication using gpudirect rdma for infiniband clusters with nvidia gpus. In *2013 42nd International Conference on Parallel Processing*, pages 80–89. IEEE, 2013.

[Poh69] Ira Pohl. *Bi-directional and heuristic search in path problems*. PhD thesis, Dept. of Computer Science, Stanford University., 1969.

[Pot97] Alex Pothen. Graph partitioning algorithms with applications to scientific computing. In *Parallel Numerical Algorithms*, pages 323–368. Springer, 1997.

[PP13] Dimitrios Prountzos and Keshav Pingali. Betweenness centrality: algorithms and implementations. In *ACM SIGPLAN Notices*, volume 48, pages 35–46. ACM, 2013.

[PR01] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *Proceedings of the 9th Annual European Symposium on Algorithms*, pages 121–133. Springer, 2001.

[PSL90] Alex Pothen, Horst D Simon, and Kang-Pu Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM journal on matrix analysis and applications*, 11(3):430–452, 1990.

[PWW+15] Yuechao Pan, Yangzihao Wang, Yuduo Wu, Carl Yang, and John D. Owens. Multi-GPU graph analytics. *CoRR*, abs/1504.04804, 2015.

[RD01] Antony Rowstron and Peter Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. *SIGOPS Oper. Syst. Rev.*, 35(5):188–201, October 2001.

[Rei05] Omer Reingold. Undirected st-connectivity in log-space. In *Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing*, STOC '05, pages 376–385, New York, NY, USA, 2005. ACM.

[Sar15] Sariy˙ Regularizing graph centrality computations. *Journal of Parallel and Distributed Computing*, 76(0):106 – 119, 2015. Special Issue on Architecture and Algorithms for Irregular Applications.

[SBVH16] Edgar Solomonik, Maciej Besta, Flavio Vella, and Torsten Hoefler. Betweenness centrality is more parallelizable than dense matrix multiplication. *arXiv preprint arXiv:1609.07008*, 2016.

[SCKD14] Edgar Solomonik, Erin Carson, Nicholas Knight, and James Demmel. Tradeoffs between synchronization, communication, and computation in parallel linear algebra computations. In *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*, pages 307–318. ACM, 2014.

[SD11] Edgar Solomonik and James Demmel. Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par 2011 Parallel Processing*, volume 6853 of *Lecture Notes in Computer Science*, pages 90–109. Springer Berlin Heidelberg, 2011.

[SH15] E. Solomonik and T. Hoefler. Sparse Tensor Algebra as a Parallel Programming Model. *ArXiv e-prints*, November 2015.

[SKCD12] Nadathur Satish, Changkyu Kim, Jatin Chhugani, and Pradeep Dubey. Large-scale energy-efficient graph traversal: A path to efficient data-intensive supercomputing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 14:1–14:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[SKSc13] Ahmet Erdem Sariyüce, Kamer Kaya, Erik Saule, and Ümit V. Çatalyürek. Betweenness centrality on GPUs and heterogeneous architectures. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, pages 76–85, New York, NY, USA, 2013. ACM.

[SS05] Yuval Shavitt and Eran Shir. Dimes: Let the internet measure itself. *ACM SIGCOMM Computer Communication Review*, 35(5):71–74, 2005.

[SSKC13] Ahmet Erdem Sarıyüce, Erik Saule, Kamer Kaya, and Umit V Catalyürek. Shattering and compressing networks for betweenness centrality. In *SIAM Data Mining Conference (SDM)*. SIAM, 2013.

[SSP+14] Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M. Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 979–990, New York, NY, USA, 2014. ACM.

[Str01] Steven H Strogatz. Exploring complex networks. *Nature*, 410(6825):268–276, 2001.

[SW05] Thomas Schank and Dorothea Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In

International Workshop on Experimental and Efficient Algorithms, pages 606–609. Springer, 2005.

[SW13] Semih Salihoglu and Jennifer Widom. Gps: a graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, page 22. ACM, 2013.

[The04] The InfiniBand Trade Association. *Infiniband Architecture Specification Volume 1, Release 1.2*. InfiniBand Trade Association, 2004.

[Tis01] Alexander Tiskin. All-pairs shortest paths computation in the BSP model. In Fernando Orejas, Paul Spirakis, and Jan van Leeuwen, editors, *Automata, Languages and Programming*, volume 2076 of *Lecture Notes in Computer Science*, pages 178–189. Springer Berlin / Heidelberg, 2001.

[top] Top 500. https://www.top500.org/. Accessed: 2016-10-12.

[TSG11] Guangming Tan, Vugranam C Sreedhar, and Guang R Gao. Analysis and performance results of computing betweenness centrality on IBM Cyclops64. *The Journal of Supercomputing*, 56(1):1–24, 2011.

[TTS09] Guangming Tan, Dengbiao Tu, and Ninghui Sun. A parallel algorithm for computing betweenness centrality. In *2009 International Conference on Parallel Processing*, pages 340–347. IEEE, 2009.

[UKBM11] Johan Ugander, Brian Karrer, Lars Backstrom, and Cameron Marlow. The anatomy of the facebook social graph. *arXiv preprint arXiv:1111.4503*, 2011.

[US13] Koji Ueno and Toyotaro Suzumura. Parallel distributed breadth first search on GPU. In *High Performance Computing (HiPC), 2013 20th International Conference on*, pages 314–323. IEEE, 2013.

[UY91] Jeffrey D Ullman and Mihalis Yannakakis. High-probability parallel transitive-closure algorithms. *SIAM Journal on Computing*, 20(1):100–125, 1991.

[Val90] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[VCB16] Flavio Vella, Giancarlo Carbone, and Massimo Bernaschi. Algorithms and heuristics for scalable betweenness centrality computation on multi-gpu systems. *arXiv preprint arXiv:1602.00963*, 2016.

[VDGW97] R. A. Van De Geijn and J. Watts. SUMMA: Scalable Universal Matrix Multiplication Algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997.

[Was94] Stanley Wasserman. *Social network analysis: Methods and applications*, volume 8. Cambridge university press, 1994.

[WDP+15] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 265–266, New York, NY, USA, 2015. ACM.

[WDP+16] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. Gunrock: A high-performance graph processing library on the gpu. In *Proceedings of the 21st*

ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, page 11. ACM, 2016.

[Wit84] Andrew P Witkin. Scale-space filtering: A new approach to multi-scale description. In *Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP'84.*, volume 9, pages 150–153. IEEE, 1984.

[WMWJ11] Jiaoe Wang, Huihui Mo, Fahui Wang, and Fengjun Jin. Exploring the network structure and nodal centrality of china's air transport network: A complex network approach. *Journal of Transport Geography*, 19(4):712–721, 2011.

[WS98] Duncan J Watts and Steven H Strogatz. Collective dynamics of 'small-world'networks. *nature*, 393(6684):440–442, 1998.

[WWYO16] Leyuan Wang, Yangzihao Wang, Carl Yang, and John D Owens. A comparative study on exact triangle counting algorithms on the gpu. In *Proceedings of the ACM Workshop on High Performance Graph Processing*, pages 1–8. ACM, 2016.

[xeo] Introducing the intel xeon phi processor. http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html. Accessed: 2016-10-12.

[YBD14] Yang You, David Bader, and Maryam Mehri Dehnavi. Designing a heuristic cross-architecture combination for breadth-first search. In *2014 43rd International Conference on Parallel Processing*, pages 70–79. IEEE, 2014.

[YCH+05a] Andy Yoo, Edmond Chow, Keith Henderson, William McLendon, Bruce Hendrickson, and Ümit Çatalyürek. A scalable distributed parallel breadth-first search algorithm on bluegene/l. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pages 25–25. IEEE, 2005.

[YCH+05b] Andy Yoo, Edmond Chow, Keith Henderson, William McLendon, Bruce Hendrickson, and Umit Catalyurek. A scalable distributed parallel breadth-first search algorithm on bluegene/l. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, pages 25–, Washington, DC, USA, 2005. IEEE Computer Society.