Communication Optimization by Data Replication for Distributed Graph Algorithms

Master's Thesis of

Tobias Ribizel

at the Department of Informatics
Institute for Theoretical Informatics, Algorithmics II

at the Department of Mathematics
Institute for Applied and Numerical Mathematics

Reviewer:	Prof. Dr. Peter Sanders
Second reviewer:	Prof. Dr. Christian Wieners
Advisor:	Dr. Christian Schulz
Second advisor:	Sebastian Schlag, M.Sc.

April 1st 2019 – September 30th 2019

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Abstract

This thesis investigates the performance of distributed graph algorithms on network graphs. More precisely, it looks at ways the communication structure can be optimized by replicating the data from a few vertices to all parallel tasks. We propose multiple simple heuristics that can be used to choose replicated vertices and evaluate their performance both in a communication model as well as based on a practical distributed implementation using sparse matrix-vector multiplication as a model problem. Additionally, we used the distributed implementation as the basis for the *PageRank* algorithm.

Experimental results seem to indicate that data replication indeed has the potential to speed up distributed graph algorithms quite significantly. On a few network graphs, we observed runtime reductions of more than 25%, but only when the data assigned to each vertex was sufficiently large, i.e., when communication latencies play only a small role. On other graphs however, we were not able to observe any speedups. This is consistent with results from the communication model, where we observed very mixed impacts of data replication.

Zusammenfassung

Diese Arbeit untersucht die Performance von verteilten Graphalgorithmen auf Graphen mit Netzwerkstruktur. Im Speziellen betrachtet sie die in verteilten Graphalgorithmen auftretenden Kommunikationsstrukturen und wie diese optimiert werden können, indem die zu wenigen "wichtigen" Knoten gehörenden Daten repliziert werden. Wir stellen mehrere einfache Heuristiken zur Auswahl dieser replizierten Knoten vor und untersuchen ihre Effektivität und Effizienz basierend auf einem Kommunikationsmodell und in einer echten verteilten Implementierung. Als Modellproblem verwenden wir hierzu die Multiplikation einer dünn-besetzten Matrix mit einem Vektor. Aufbauend auf diesem Baustein wurde zudem der bekannte *PageRank*-Algorithmus implementiert.

Die experimentellen Ergebnisse deuten darauf hin, dass Replikation das Potential hat, die Kommunikation in verteilten Graphalgorithmen deutlich zu beschleunigen: Für einige Eingabegraphen konnten wir um mehr als 25% reduzierte Laufzeiten beobachten. Diese Ergebnisse lassen sich allerdings nur beobachten, wenn wir das Produkt einer Matrix mit mehreren rechten Seiten berechnen, also die Kommunikationslatenzen eine vernachlässigbare Rolle im Vergleich zur Bandbreite spielen. Auf vielen Netzwerkgraphen konnten wir keinerlei positiven Effekt der Replikation beobachten. Diese Beobachtungen decken sich mit den sehr unterschiedlichen Ergebnissen basierend auf dem Kommunikationsmodell.

Acknowledgments

This thesis would not have been possible without the support of many people. First of all I want to thank my supervisors Dr. Christian Schulz and Sebastian Schlag for many great discussions, suggestions, feedback and encouragement that helped finish this thesis. Secondly I want to thank Prof. Sanders for strongly shaping my interest in Algorithm Engineering and Prof. Wieners for helping me extend the topics of this thesis to the mathematical world. I also want to thank Dr. Hartwig Anzt for introducing me to the fascinating High Performance Computing ecosystem and Dennis Keck, Dominik Kiefer and Xenia Volk for their helpful proof-reading. Finally I'd like to thank Florian Grötschla for providing his generator code for random hyperbolic graphs.

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, 30.09.2019

. . . .

(Tobias Ribizel)

Contents

1. Introduction **1**

2. Fundamentals **3**
2.1. Graph Theory 3
2.2. Graph Partitioning 4
2.3. Parallel Programming Models 7
2.4. Sparse Matrices 9
2.5. Graph and Matrix Reordering 11
2.6. Iterative Solvers for Linear and Eigenvalue Problems 12

3. Related Work **19**
3.1. Distributed SpMV 19
3.2. Communication Models 21
3.3. (Hyper-)Graph Partitioning Algorithms 23
3.4. Network Graphs 24
3.5. Vertex Centrality 27

4. Data Replication **29**
4.1. Basic Idea 29
4.2. Communication Model 31
4.3. Replication Heuristics 32
4.4. Two-Level Replication 35

5. Evaluation **39**
5.1. Experimental Setup 39
5.2. Experimental Results 42

6. Conclusion **49**

A. Bibliography **51**

B. Additional Experimental Results **57**
B.1. Theoretical Communication Model 57
B.2. Runtime Measurements for SpMV 61

List of Figures

2.1. Example of a hypergraph and its bipartite representation 4
2.2. Graph partition and its quotient graph 5
2.3. Parallel programming models: shared and distributed memory 7
2.4. Runtime of a bulk synchronous parallel process 8
2.5. Tree broadcast and reduction 9
2.6. Compressed row and column storage format for a sparse matrix 11

3.1. Two possible implementations of distributed SpMV with 1D partitioning 21
3.2. Random hyperbolic graph 27

4.1. Effect of replication on the bottleneck communication volume 32
4.2. Degree and cut-degree distribution in a network graph 33
4.3. Counterexample for greedy replication 35
4.4. Four phases of two-level All-Reduce 36

5.1. Communication graphs for unreplicated and replicated SpMV 43
5.2. Runtime breakdown for `eu-2005` and `in-2004` 44
5.3. Runtime breakdown by node 46
5.4. Convergence plot of PageRank 48
5.5. Stagnation of GMRES 48

List of Algorithms

2.1. SpMV for different sparse matrix formats 10
2.2. Arnoldi process with modified Gram-Schmidt orthonormalization 14
2.3. QR decomposition of Hessenberg matrix H_m 15
2.4. Power iteration 16
2.5. Rayleigh quotient iteration 17
3.1. Distributed SpMV 20
3.2. Preferential attachment graph generation 26
4.1. Distributed SpMV with replication 30
4.2. Greedy replication 34
4.3. Distributed SpMV with two-level replication 37

List of Tables

5.1. Input graphs with basic properties 40
5.2. Minimal communication volume by replication heuristic for $k = 64$ . . . 44
5.3. Total bottleneck runtime for distributed SpMV with $k = 64$ 45

B.1. Minimal communication volume by replication heuristic for $k = 16, 256$. 57
B.2. Theoretical communication volume for different replication heuristics . . 58
B.3. Total bottleneck runtime for distributed SpMV with $k = 16$ 61
B.4. Distributed SpMV runtime for $k = 64$ 62
B.6. Distributed SpMV runtime for $k = 16$ 65

1. Introduction

Since the early ages of computing, graphs have been the objects at the center of many algorithms. With the digital age entering more and more of our lives, the amount of real-world data collected grows faster and faster. Much of this data contains relational information, whose analysis requires sophisticated algorithms. This growing amount of data is however not mirrored by a corresponding growth in single-core performance of modern computers. As even the performance of multicore processor starts to hit a limit, we need to move to larger, distributed systems to still be able to handle these input sizes. Another important development is the growing divergence between communication and memory performance on the one hand and computational power on the other hand in parallel and distributed systems. On modern systems, avoiding or minimizing communication and synchronization becomes more and more important.

This thesis evaluates a speedup technique for communication in distributed graph algorithms on a class of irregularly structured graphs. On these graphs, we can trade-off communication for computation in order to speed up the overall algorithm, which is especially interesting in terms of the aforementioned trends.

Structure of the thesis

Chapter 2 introduces the basic concepts of (hyper-)graph partitioning, parallel programming models and sparse matrices. Additionally, it describes iterative algorithms that can be used to compute solutions to linear systems and Eigenvalue problems, which we will later use as an application example for distributed SpMV computation. Chapter 3 lists previous efforts in the theoretical treatment and implementation of distributed graph algorithms. It also introduces distributed sparse matrix-vector multiplication (SpMV) as a model problem and common models for its parallel communication. Finally, it introduces the structural properties and generation models for our central objects of study - network graphs as well as the PageRank metric as an example for an Eigenvalue problem that is encountered based on network graphs. Chapter 4 introduces data replication as an extension of normal distributed implementations of SpMV, extends previous communication models to incorporate replicated data and heuristics for the choice of replicated vertices. Finally, it gives an outlook to extensions of the data replication model for multilevel applications. Chapter 5 presents a practical evaluation of data replication as a communication optimization technique, partly based on a communication model and partly implemented on real hardware. Chapter 6 discusses the implications of the experimental results and conclusions which lead to potential further areas of exploration.

2. Fundamentals

In this chapter, we introduce the basic notions of graphs and hypergraphs and their connection to sparse matrices, graph and hypergraph partitions and their application in parallel algorithms as well as graph and matrix reordering.

2.1. Graph Theory

The central object of study for this thesis are graphs $G = (V, E)$, both undirected in the context of graph partitioning and directed when representing data flow or computations, where we also allow self-loops. We use the following notation throughout the thesis:

Definition 2.1: Graph notation

For a graph $G = (V, E)$, we write $V = V(G)$ for the vertex set and $E = E(G)$ for its edge set.
For a vertex $u \in V$, we denote by $\mathrm{N}(u) = \{v \mid (u, v) \in E\}$ the (forward-) neighborhood and by $\mathrm{N}^{\leftarrow}(u) = \{v \mid (v, u) \in E\}$ the backward-neighborhood of u.
Similarly, $\deg(u) = |\mathrm{N}(u)|$ denotes the (forward-)degree and $\deg^{\leftarrow}(u) = |\mathrm{N}^{\leftarrow}(u)|$ the backward-degree of u.
Finally, we denote by $G^{\leftarrow} = (V, E^{\leftarrow})$ the backwards graph we get by reversing all edges, i.e., $E^{\leftarrow} = \{vu \mid uv \in E\}$, and by $G^{bi} = (V, E \cup E^{\leftarrow})$ the corresponding bidirected graph.

We will often use the natural relationship between graphs and square matrices:

Definition 2.2: Relationship between graphs and matrices

To every graph $G = (V, E)$, we can define an adjacency matrix $A(G)$ by ordering the vertices $V = \{v_1, \ldots, v_n\}$:

$$a_{ij} = \begin{cases} 1, & v_j v_i \in E \\ 0, & v_j v_i \notin E \end{cases}$$

Conversely, to every square matrix $A \in \mathbb{R}^{n \times n}$ there exists a corresponding graph with vertices $V = \{v_1, \ldots, v_n\}$ and edges $E = \{v_j v_i \mid a_{ij} \neq 0\}$ describing its sparsity pattern.

Remark: Our definition of an adjacency matrix is transposed compared to its usual definition, because then, the edge directions in the graph correspond to the direction of data flow when computing the matrix-vector product $A(G)x$

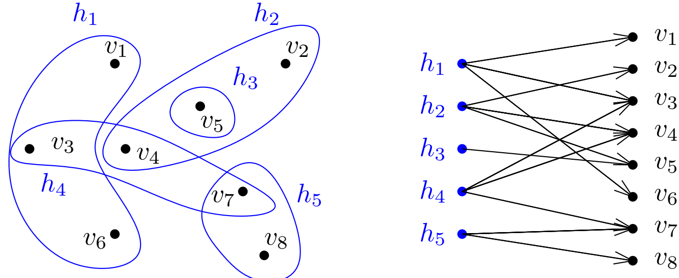


Figure 2.1.: Example of a hypergraph with vertices $v_1, \ldots, v_7$ and nets $h_1 = \{v1, v3, v6\}$, $h_2 = \{v_2, v_4, v_5\}$, $h_3 = \{v_5\}$, $h_4 = \{v_3, v_4, v_7\}$, $h_5 = \{v_7, v_8\}$ and its bipartite representation.

If we relax the requirement that every edge connects exactly two vertices, thus allowing an arbitrary number of vertices per edge, we arrive at the concept of hypergraphs:

Definition 2.3: Hypergraph

A hypergraph $\mathcal{H}$ consists of a set of vertices $\mathcal{V}$ and nets $\mathcal{N} \subseteq \mathcal{P}(\mathcal{V})$ (also sometimes called *hyperedges*). For a given net $n \in \mathcal{N}$, the vertices $v \in n$ are called *pins*.
Each Hypergraph also has a natural correspondence to a bipartite graph, which we shall call the bipartite representation:
For a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$, we define a bipartite graph with the vertex set $\mathcal{V} \cup \mathcal{N}$, where a vertex $v \in \mathcal{V}$ and a net $n \in \mathcal{N}$ are connected by an edge if v is a pin of n, i.e., $v \in n$.

Hypergraphs are able to directly model more complex relationships involving multiple vertices, and are thus sometimes the preferred model for communication between parallel processes, as we will see later. Figure 2.1 shows an example of such a hypergraph with its bipartite representation. Note that while empty nets are not explicitly forbidden, they usually don't have a practical application and can thus be ignored.

2.2. Graph Partitioning

Graph partitioning is a fundamental tool for work distribution and load balancing in a large number of parallel programming applications. It is based on the observation that graphs can encode a large variety of computational tasks described by operations and their dependencies. For a given graph $G = (V, E)$, a *k-way partition* is a partition of its vertex set into k disjoint parts: $V = V_1 \dot\cup \ \ldots \ \dot\cup V_k$. To simplify the notation, we introduce the partition function p giving the unique part index for each vertex: $p(v) = i \Leftrightarrow v \in V_i$. Based on such a partition, the edges and vertices can be classified as follows:

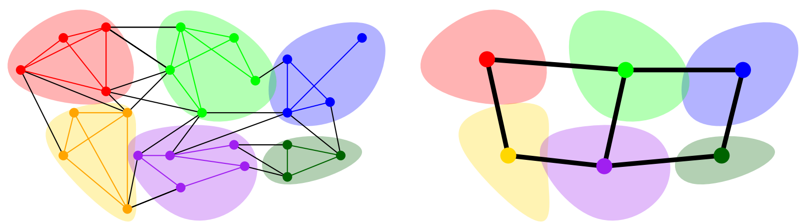


Figure 2.2.: Graph partition (l) and its quotient graph (r). Cut edges are drawn in black, inner edges in the respective part colors.

Definition 2.4: Boundary and interior of parts

cut-edge uv:	u and v are in different parts, i.e., $p(u) \neq p(v)$
inner edge uv:	u and v are in the same part, i.e., $p(u) = p(v)$
boundary vertex v:	v has neighbors in other parts, i.e., it is incident to a cut edge.
inner vertex v:	v only has neighbors in the same part

We denote the set of all cut-edges by E^{cut} and the set all of cut-edges incident to the part V_i by E_i^{cut}.

The global structure of a graph partition can also be described using the so-called quotient graph. The vertices of the quotient graph are all parts of the partition, where two partitions are connected by a partition if there is any cut edge between these parts. Figure 2.2 shows an example of a graph partition with its corresponding quotient graph.

If we additionally equip the graph with node weights $w : V \longrightarrow \mathbb{R}_{\geq 0}$ and edge weights[1] $c : E \longrightarrow \mathbb{R}_{\geq 0}$, we can describe the quality of a graph partition based on the following quantities:

Definition 2.5: Key quantities of a graph partition

Balance

avg. part weight:	$\overline{w}(V) = w(V)/k$
part weight:	$w(V_i) = \sum_{v \in V_i} w(v_i)$
imbalance:	$I(G) = \max_{i=1}^{k} w(V_i)/\overline{w}(V)$

Cut-edges

part cut-size:	$c(V_i) = \sum_{e \in E_i^{cut}} c(e)$
total cut-size:	$c(G) = \sum_{i=1}^{k} c(V_i)$
bottleneck cut-size:	$c_{max}(G) = \max_{i=1}^{k} c(V_i)$

A graph partition is called ε-balanced if its imbalance is at most $1 + \varepsilon$, i.e., its largest part is at most $\varepsilon\overline{w}(V)$ larger than the average part.

[1]If unspecified, we assume unit node and edge weights $w \equiv 1, c \equiv 1$.

The goal of a graph partitioning algorithm is then to find a k-way partition of an input graph that is ε-balanced and minimizes the total (or bottleneck) cut-size.

Remark 2.1: Existence of a feasible solution

For arbitrary vertex weights, even deciding if a feasible solution fulfilling the balance constraint exists is NP-hard, as it can be used to encode the BinPacking problem. For unit vertex weights $w \equiv 1$, the average part weight is usually modified to force the existence of a feasible solution for every imbalance $\varepsilon > 0$:

$$\overline{w}(V) = \lceil w(V)/k \rceil$$

With a few modifications, the general approach of graph partitioning can also be extended to hypergraphs, which are able to more exactly model the communication encountered in distributed graph algorithms. A k-way partition of a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ is a partition of the vertices into k parts $\mathcal{V} = \mathcal{V}_1 \;\dot\cup\; \dots \;\dot\cup\; \mathcal{V}_k$. Again, we introduce the partition function $p : \mathcal{V} \to \{1, \dots, k\}$ to assign the part index to every vertex. Based on this, we can give the following description of the resulting cut:

Definition 2.6: Connectivity

For each net $n \in \mathcal{N}$, the partition function $p(n)$ gives us the *connectivity set*, i.e., the set of parts which n intersects. With this, we can again distinguish between cut- and inner nets:

cut net n: n contains pins from more than one part, i.e., $|p(n)| > 1$
inner net n: n contains only pins from a single part, i.e., $|p(n)| = 1$

We denote the set of all cut nets by $\mathcal{N}^{cut}$ and the set all of cut nets connected to the part $\mathcal{V}_i$ by $\mathcal{N}_i^{cut}$.

As with regular graph partitioning, we want to compute a hypergraph partition with limited imbalance ε, while optimizing a cut size metric. For hypergraphs, there are two such metrics which are commonly used (for simplicity, we assume unit hyperedge weights):

Definition 2.7: Cut size metrics for hypergraphs

The simplest definition of a cut size metric is the number of cut nets:

$$c(\mathcal{V}_i) = \sum_{n \in \mathcal{N}_i^{cut}} 1,$$

however, the communication volume of distributed algorithms is often better modeled by the cut connectivity:

$$c(\mathcal{V}_i) = \sum_{n \in \mathcal{N}_i^{cut}} (|p(n)| - 1).$$

Based on these per-part cuts, we again define total and bottleneck cut metrics.

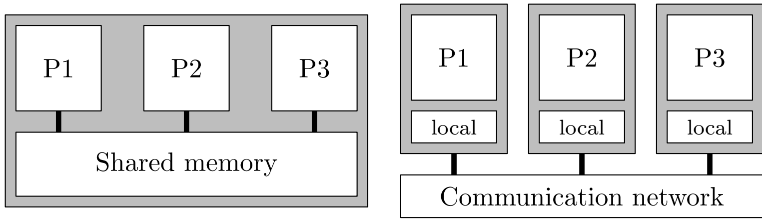


Figure 2.3.: Parallel programming models: shared and distributed memory

Hypergraph partitioning provides a generalization of graph partitioning in that every graph partition is also a hypergraph partition constructed by converting every edge from the graph into a two-element net in a hypergraph. The cut of the graph partition is then equal to both the number of cut nets and the cut connectivity of the corresponding hypergraph partition. The increased flexibility of hypergraphs proves important in the exact modeling of the communication volume for distributed algorithms, as we will see later.

Both graph partitioning and hypergraph partitioning as its generalization are NP-complete problems, and can thus in practice only be optimized approximately using various heuristics. These heuristics mostly operate by first computing a suitably good initial partition, which is then refined using local searches that move individual vertices or groups of vertices between parts to improve the objective.

2.3. Parallel Programming Models

For the theoretical treatment of parallel algorithms, many programming models were developed over the years, mostly based on different granularities of parallelism as well as different hardware capabilities and limitations.

The most fundamental distinction in these models lies in the way in which different parallel tasks are able to communicate and synchronize (See Figure 2.3):

Shared-memory programming models assume that all parallel tasks have read and write access to a shared area of memory, which can be used to synchronize and (implicitly) communicate between these tasks. Arguably the most important model is the CREW-PRAM[2] model. It allows multiple parallel processes read access to the same memory word, but only exclusive write operations, which means that for all write accesses, the algorithm must either make sure that they can never intersect, or use synchronization primitives to ensure exclusive access to this memory location. The runtime of an algorithm in the PRAM model is the time until the last parallel process finishes.

While such applications can theoretically be implemented directly using the tools provided by most modern programming languages, frameworks like OpenMP[3] are often used to take care of work distribution, synchronization and low-level thread management, which would otherwise need to be implemented explicitly.

[2]Concurrent Read, Exclusive Write Parallel Random Access Machine

[3]Open Multi-Processing

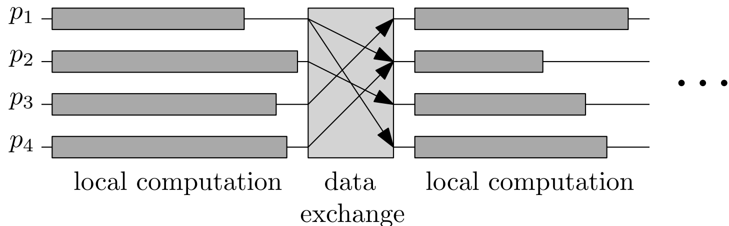


Figure 2.4.: Runtime of a bulk synchronous parallel process

Distributed-memory programming models require the parallel tasks to explicitly exchange data by passing messages. Most distributed-memory applications are implemented using the *Message Passing Interface* (MPI), which provides low-level communication operations as well as more complicated collective operations. There are many different models approximating the runtime characteristics of distributed algorithms, especially depending on the communication structure. However, the applications described in this thesis exclusively use collective communication operations, so their runtime is best described using the *bulk synchronous parallel* (BSP) model. As Figure 2.4 shows, an algorithm in the BSP model consists of alternating steps of local computation and data exchange/global communication. The runtime of such an algorithm is then the maximum time necessary for local computations and communication over all distributed nodes, as the communication step entails an implicit synchronization between all nodes. The runtime is thus limited by both the slowest local computation step as well as the slowest communication step of all parallel processes.

Remark 2.2: Nodes and vertices

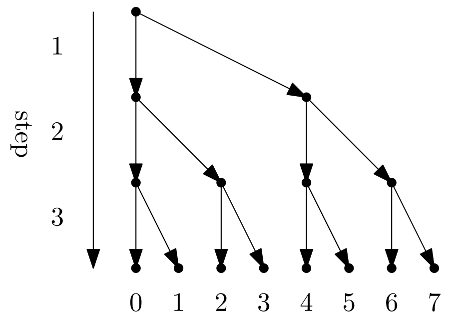
To avoid name confusion with the elements of a graph, we will exclusively use *nodes* to describe distributed parallel processes and *vertices* for the graph elements.

In the development of algorithms with increasing degrees of parallelism, many problems exhibit properties where the actual computations play a diminishing role compared to unavoidable communication between different parallel tasks. This need for communication can arise both on a fine-grained parallelism level (processor cores) and on very coarse levels (compute nodes or even groups of nodes that are physically or logically “close”).

In distributed computing, one usually distinguishes between two communication modes: In *peer-to-peer communication* (p2p), individual tasks communicate with a small number of other tasks. This communication mode is usually used if the parallel tasks only need to exchange small amounts of data at irregular intervals.

During collective operations, all tasks or a large number of tasks participate in a data exchange operation. Such operations are usually implemented using p2p communication. There are many different types of collective operations, some of which are described in the following:[4]

[4]Note that these are only descriptions of the logical operations, not their real implementation



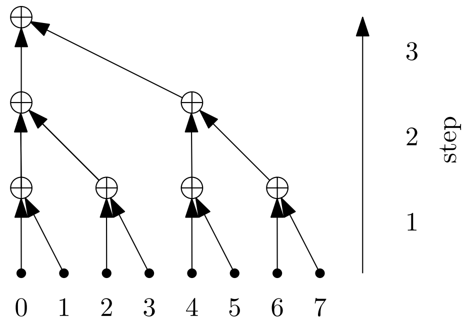


Figure 2.5.: Tree broadcast (left) and reduction (right) with 8 parallel processes

Definition 2.8: Types of collective operations

All-to-all	Each task i sends a message m_{ij} to each other task j. These exchanged messages may all be of the same size (*regular*), of varying sizes (*irregular*) or even empty (*sparse*).
(All-)Gather	Each task i sends a message m_i. One task (or all tasks) receives the combined messages $(m_1, \dots)$
(All-)Reduce	Each task i sends a message m_i. The messages are accumulated using an associative binary operation $\oplus$, and one task (or all tasks) receives the result $\bigoplus_i m_i$
Broadcast	A single task sends a message m to all other tasks

All mentioned collective operations except for all-to-all communication can all be implemented based on tree-shaped reduction and broadcast as shown in Figure 2.5. Both operations require $\lceil \log_2 p \rceil$ steps for p parallel processes, and each process sends or receives at most two messages per step. A parallel reduction or broadcast of n bytes thus, assuming a full duplex interconnection, takes time $2(T_{start} + nT_{byte}) \log p$, where T_{byte} is the time to transmit a single byte and T_{start} is the transmission latency. For irregular all-to-all communication, it is difficult to give a general runtime bound, as the runtime can strongly depend on hardware characteristics, among others the bandwidth and latency for a single message, how many independent pairs of nodes can communicate in parallel, and also the number and sizes of messages to be exchanged. We will simply use the total or bottleneck communication volume as a runtime approximation, i.e., the combined size of all messages sent by all nodes or the "slowest" node.

2.4. Sparse Matrices

In practical applications, for example as a spatial discretization of differential operators often occurring in partial differential equations, the resulting matrices closely resemble the geometry of the underlying space, especially its sparse neighborhood structure: Each row or column only contains a few non-zero entries belonging to the "neighboring" rows or columns in a geometric sense. Storing all the zero entries in-between would be a waste

of memory (and also computational power), so the sparse structure of these matrices is usually leveraged by only storing and computing with non-zero entries.

Due to their often irregular structure, sparse matrices need specialized formats to efficiently store them. Depending on the use case, several formats can provide the best runtime-memory tradeoff:

- The *coordinate format* (COO) stores the entries of a sparse matrix directly as a list of $(row, column, value)$ tuples, similar to the edge list representation of its corresponding graph. While this format is probably the most simple, it is usually difficult to use in practical algorithms.
- The *compressed sparse row format* (CSR) stores the concatenated column indices of all non-zeros together with the corresponding index range for each row and is thus equivalent to the (transposed! See Definition 2.2) adjacency array structure of the graph whose adjacency matrix has the same sparsity structure as the sparse matrix. In practice, its usage and advantages can be compared to the row-major storage order for dense matrices.
- The *compressed sparse column format* (CSC) stores the row indices of all non-zeros together with the corresponding index ranges for all columns, and is thus equivalent to the CSR representation of the untransposed matrix. It is thus similar to the column-major storage order for dense matrices.

Figure 2.6 shows an example of the CSR and CSC format for a small sparse matrix. The sorted order of the column and row indices is not strictly necessary, but usually improves the performance of corresponding kernels by improving the spatial locality of the memory accesses.

When we want to use a sparse matrix $A \in \mathbb{R}^{n \times m}$ to compute the matrix-vector product $y = Ax$, the sequential implementations are straightforward (assuming y is zero-initialized):

Algorithm 2.1: SpMV for different sparse matrix formats

Function *SpMV_CSR(ranges, col, value, x, y)*
  for $row = 0, \ldots, n$ **do**
    for $i = ranges[row], \ldots, ranges[row + 1] - 1$ **do**
      $y[row] \leftarrow y[row] + values[i] \cdot x[col[i]]$

Function *SpMV_CSC(ranges, row, value, x, y)*
  for $col = 0, \ldots, m$ **do**
    for $i = ranges[col], \ldots, ranges[col + 1] - 1$ **do**
      $y[row[i]] \leftarrow y[row[i]] + value[i] \cdot x[col]$

Function *SpMV_COO(row, col, value, x, y)*
  for $i = 0, \ldots, nnz$ **do**
    $y[row[i]] \leftarrow y[row[i]] + value[i] \cdot x[col[i]]$

In these sequential implementations, there are no significant difference in terms of performance – except for the larger memory footprint of the COO for matrices with at

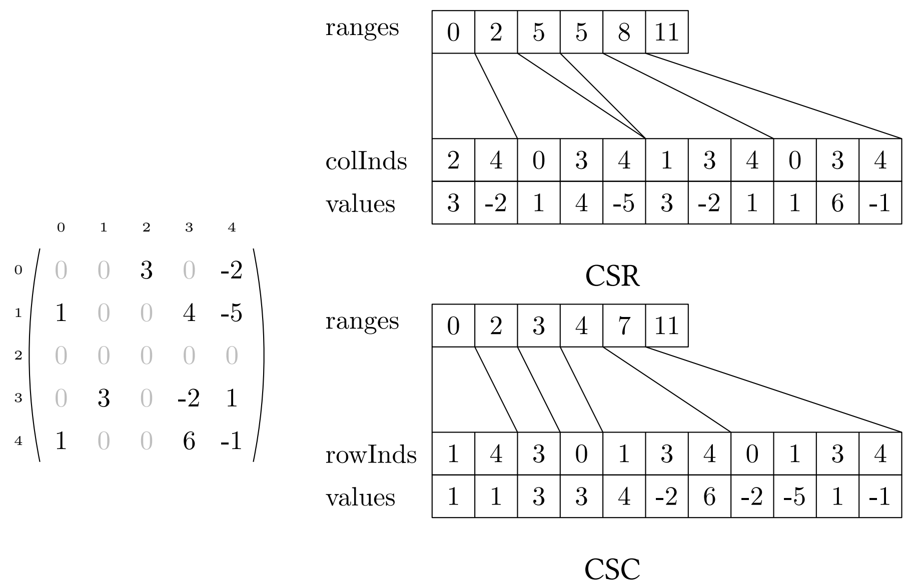


Figure 2.6.: Compressed row and column storage format for a sparse matrix

least a few entries per row/column. When parallelizing them on a shared-memory system however, the results are significantly different: The CSR kernel can easily be parallelized by distributing the different rows among different parallel processes - all write accesses operate on distinct entries of y and require no synchronization. The CSC and COO kernels don't share this property: There, two different parallel processes can try to read from or write to the same entry of y, which is why these implementations require atomic read and write operations to avoid race conditions. In practice, it should be expected that shared-memory parallel CSR kernels achieve much better performance than their CSC or COO counterparts.

2.5. Graph and Matrix Reordering

When using them in any algorithm, both matrices and graphs are usually stored and accessed by assigning indices to the rows/columns or vertices. While in many situations, there is a natural indexing order available either by construction or due to structural properties of the graph or matrix, it may sometimes prove useful to use another indexing order.

For (sparse) matrices, such reordering is frequently used in numerical linear algebra, for example to reduce fill-in when computing matrix decompositions. Popular examples of this are the Cuthill–McKee algorithm [1] and the nested dissection order [2]. Reordering can also be used to improve the memory locality and thus the cache utilization of a variety of algorithms, as has been demonstrated for geometric data using space-filling curves [3]. In the context of graphs, reordering can be used to compress adjacency-array representations by making sure that the neighborhoods of most graphs are within a small index range, thus making them easily compressible using delta-encoding [4].

Finally, reordering gives a natural representation of (hyper-)graph partitions: By reordering the vertices such that each part forms a consecutive index interval, we no longer need to store the partition function p, but can simply check by comparing its index to the boundaries of the part whether a vertex belongs to this part. Additionally, the conversion between part-local indices and global indices can be greatly simplified: If the pth part consists of vertices from the index range $[a_p, b_p)$, we can simply translate between a local vertex numbering in V_p and the global numbering in V by adding or subtracting a_p.

From the view of the underlying matrix, such a partition-based reordering brings the matrix into a blocked form - the computation of a sparse matrix-vector product with coherently k-way partitioned vectors can thus be understood in its reordered form

$$A = \begin{pmatrix} A_{11} & \cdots & A_{1k} \\ \vdots & \ddots & \vdots \\ A_{k1} & \cdots & A_{kk} \end{pmatrix}, \; x = \begin{pmatrix} x_1 \\ \vdots \\ x_k \end{pmatrix}, \; y = \begin{pmatrix} y_1 \\ \vdots \\ y_k \end{pmatrix} \tag{2.1}$$

$$y = Ax \Leftrightarrow y_i = \sum_{j=1}^{k} A_{ij} x_j$$

where the blocks A_{ij} describe the contributions of x_j to y_i.

2.6. Iterative Solvers for Linear and Eigenvalue Problems

2.6.1. Linear Systems

Large, sparse linear systems of equations $Ax = b$ occur in a large number of scientific computing applications. Common direct solvers for such systems usually break down on the large scale, as the sparse structure of A usually cannot be maintained: They rely on operations like the LU-, QR- or Cholesky decomposition, which often can produce significant fill-in even for very sparse matrices.

The solution for this problem lies in iterative solvers, which respect the sparsity structure of A and can still produce very good approximations to the exact solution x in a small number of iterations. A large variety of iterative solvers have been developed for different applications. They can be roughly classified as:

- Krylov Subspace Methods, which can be categorized into two groups: The first group are methods based on the Arnoldi process computing an orthonormal basis for the Krylov spaces

  $$\mathcal{K}_n(A, b) = \operatorname{span}\{b, Ab, A^2b, \ldots, A^{n-1}b\} \tag{2.2}$$

  like the GMRES method for general matrices. The second group are methods based on the Non-Symmetric Lanczos Process computing biorthogonal bases for two Krylov spaces $\mathcal{K}_n(A, b)$ and $\mathcal{K}_n(A^*, \hat{b})$ like the BiCG family. The conjugate gradient (cg) method for symmetric positive-definite matrices can be considered part of both groups, as the Lanczos and Arnoldi process are equivalent in this case.

- Multigrid Methods that combine a hierarchy increasingly coarse representations of the linear system connected by restriction and prolongation operators with iterative solvers on the different levels, like geometric and algebraic multigrid methods.

- A large variety of Fixed-Point Iteration Methods that can either be used directly as iterative solver, or as preconditioners in other methods, like the Jacobi and, Gauß-Seidel iteration and incomplete factorization methods (ILU, incomplete Cholesky)

Preconditioners are easy-to-evaluate operators $P \approx A^{-1}$ that usually increase the convergence rate of iterative solvers and are implicitly applied by solving the linear system

$$PAx = Pb \tag{2.3}$$

instead of $Ax = b$

At their core, all of these methods use sparse matrix-vector products combined with other operations like scalar products, linear combinations of vectors and operations on smaller dense matrices that usually have a lower complexity than the SpMV operation, which is why we mostly concentrate on the SpMV operations. Most operations necessary for these solvers can easily be parallelized, which is why they are of special interest for HPC applications.

We want to focus on the General minimal residual (GMRES) method introduced by Saad and Schultz [5] as it works with the most general set of matrices and can easily be adapted for Eigensolvers. Due to its simplicity and embarrassingly parallel implementation, we will additionally combine it with a Jacobi preconditioner. The GMRES algorithm consists of two parts: The computation of an orthonormal basis for the Krylov spaces $\mathcal{K}_m(A, b)$ using the modified Gram-Schmidt method (MGS) and the minimum-residual approximation

$$x_m = \underset{x \in \mathcal{K}_m(A,b)}{\operatorname{argmin}} \|Ax - b\| \tag{2.4}$$

of the solution of our linear system $Ax = b$.

The Arnoldi process described in Algorithm 2.2 computes a basis $V_m = (v_1 \dots v_m)$ of $K_m(A, b)$ and the coefficients $H_m = (h_{ij})_{ij}$ of the Arnoldi relation

$$AV_m = V_{m+1}H_m, \quad H_m \in \mathbb{R}^{(m+1)\times m} \text{ with } h_{ij} = 0 \text{ for } i > j + 1 \tag{2.5}$$

The minimum residual-approximation $x \in K_m(A, b)$ can then be computed using the orthonormality of V_m: With $\rho = \|b\|$ and using $z = V_m^T x$ we have

$$\|b - Ax\| = \|\rho v_1 - AV_m z\| = \|\rho v_1 - V_m H_m z\| = \|\rho e_1 - H_m z\|,$$

so it suffices to solve the simpler least-squares problem

$$\hat{z} = \underset{z \in \mathbb{R}^m}{\operatorname{argmin}} \|\rho e_1 - H_m z\| \tag{2.6}$$

and get the corresponding solution $\hat{x} = V_m z$.

Algorithm 2.2: Arnoldi process with modified Gram-Schmidt orthonormalization

$v_1 \leftarrow b/\|b\|$
for $j = 1, 2, \ldots$ **do**
  $\tilde{v}_{j+1} \leftarrow Av_j$
  for $i = 1, \ldots, j$ **do**
    $h_{ij} \leftarrow \langle v_i, \tilde{v}_{j+1} \rangle$
    $\tilde{v}_{j+1} \leftarrow \tilde{v}_{j+1} - h_{ij} v_i$
  end
  $h_{j+1,j} \leftarrow \|\tilde{v}_{j+1}\|$
  $v_{j+1} \leftarrow \tilde{v}_{j+1}/h_{j+1,j}$
end

This smaller problem (2.6) is most easily solved using the QR decomposition of H_m. Here we can use the Hessenberg form of the matrix and the fact that H_m grows by adding a row and column per iteration. Thus for $H_{m-1} = Q_{m-1}R_{m-1}$ where $Q_{m-1} \in \mathbb{R}^{m\times m}$ is unitary and $R_{m-1} \in \mathbb{R}^{m\times(m-1)}$ is an upper triangular matrix, we can derive

$$\tilde{R}_m = \overbrace{\begin{pmatrix} Q_{m-1}^T & 0 \\ 0 & 1 \end{pmatrix}}^{=:\, \tilde{Q}_m^T} H_m = \begin{pmatrix} R_{m-1} & \overbrace{\begin{matrix} Q_{m-1}^T h_{\bullet,m} \\ h_{m+1,m} \end{matrix}}^{=:\, \tilde{h}_{\bullet,m}} \\ 0 & \end{pmatrix}$$

Thus we only need to find a unitary matrix G_m such that $R_m = G_m \tilde{R}_m$ is upper-triangular. Then with $Q_m = \tilde{Q}_m G_m^T$, we get the QR decomposition $H_m = Q_m R_m$ of the next iteration step. A simple choice of G_m is the Givens rotation

$$G_m = \begin{pmatrix} I_{n-1} & & \\ & c & s \\ & -s & c \end{pmatrix} \text{ with } c^2 + s^2 = 1$$

To zero the last row of $\tilde{R}_m$, the parameters c, s only need to satisfy $ah_{m+1,m} = b\tilde{h}_{mm}$. For a practical implementation, it does not make sense to store the matrix Q_m directly. Instead, we store only the parameters c_i, s_i of the givens rotations and use them to evaluate every multiplication by $Q_m = G_1 G_2 \cdots G_m$. The complete QR decomposition based on computing the Givens parameters c_i, s_i and upper triangular matrix $R_m = (r_{ij})_{ij}$ is described in Algorithm 2.3.

While the decomposition algorithm could also be reordered to apply each Givens rotation to whole rows of R_m, the formulation in Algorithm 2.3 has the advantage that R_m can grow in the same way as H_m during the Arnoldi process. This approach has another helpful side-effect: With the QR decomposition, our optimization problem (2.6) can be rewritten based on the isometry properties of Q_m:

$$\|\rho e_1 - H_m z\| = \|\rho Q_m^T e_1 - R_m z\|$$

The first m entries of the residual $\rho Q_m^T e_1 - R_m z$ can be set to zero by solving the triangular system

$$Q_m^T e_1 = R_m z \tag{2.7}$$

for z, the last entry of $Q_m^T e_1$ then gives the norm of the residuum of (2.4) due to the orthonormality of Q_m and V_m. Thus, using algorithm 2.3, we can continuously update the right-hand $Q_m^T e_1$ to compute the residuum norm $\|b - Ax\|$ of our minimum-residuum approximation, which can be useful as a stopping criterion. This implicitly covers the case that the solution of $Ax = b$ already lies in $K_m(A, b)$, which leads to $h_{m+1,m} = 0$.

Algorithm 2.3: QR decomposition of Hessenberg matrix H_m

$R_m \leftarrow$ upper triangle of H_m
for $j = 1, \dots, m$ **do**
  for $i = 1, \dots, j-1$ **do**
    $$\begin{pmatrix} r_{ij} \\ r_{i+1,j} \end{pmatrix} = \begin{pmatrix} c_i & s_i \\ -s_i & c_i \end{pmatrix} \begin{pmatrix} r_{ij} \\ r_{i+1,j} \end{pmatrix}$$
  end
  Determine c_j, s_j such that $c_j h_{j+1,j} = s_j r_{jj}$ and $c_j^2 + s_j^2 = 1$
  $r_{jj} \leftarrow c_j r_{jj} + s_j h_{j+1,j}$
end

The convergence speed of GMRES cannot be bounded in general based on the Eigenvalues of A, as Greenbaum et al. [6] showed for non-normal matrices. However, for normal matrices, Liesen and Tichý [7] proved tight worst-case bounds based on the spectrum of A. Still, without a-priori knowledge about the properties of the matrix A under study, we cannot give suitable bounds for the minimum-residual approximation (2.4) before actually computing the Krylov basis and Arnoldi relations. In exact arithmetic, GMRES would compute an exact solution after at most n steps for $A \in \mathbb{R}^{n \times n}$, as the Krylov spaces $K_m(A, b)$ become A-invariant when they contains the solution x to $Ax = b$.

The storage required for the basis vectors V_m grows linearly with the number of iterations m and can overwhelm the available memory even large parallel computers. Thus in practice one often uses GMRES with restarts after a limited number k of iteration, also called GMRES(k). To this aim, the intermediate solution x computed before the restart will be used to compute the residual $r = b - Ax$. Instead of the original system, the following run of GMRES then computes the solution Δx of the modified system $r = A\Delta x$, which we can combine to the solution $A(x + \Delta x) = b$ of the original system. This approach leads to a slower convergence of the algorithm in terms of the number of iterations, but can still be faster in practice.

2.6.2. Eigenvalue Problems

The simplest way to compute Eigenvalues and Eigenvectors of a square matrix $A \in \mathbb{R}^{n \times n}$ is based on the power iteration: Assuming A is diagonalizable with Eigenpairs

$$(\lambda_1, b_1), \dots (\lambda_n, b_n), |\lambda_1| \geq \dots \geq |\lambda_n|,$$

we can write any vector $x \in \mathbb{R}^n$ using the Eigenbasis $x = c_1 b_1 + \cdots + c_n b_n$. Repeated multiplication with A thus gives us

$$A^k x = \lambda_1^k c_1 b_1 + \cdots + \lambda_n^k c_n b_n$$

based on the properties of the Eigenvectors. This vector can give a very good approximation to the Eigenvector corresponding to the largest Eigenvalue: Assuming $c_1 \neq 0$ and $|\lambda_1| \gg |\lambda_2|$, we can assume w.l.o.g.[5] that $c_1 = 1$ and $\lambda_1 = 1$, so $A^k x$ is an approximation to the Eigenvector b_1 with error $\left\|A^k x - b_1\right\| = O(\|\lambda_2^k x\|)$. The convergence speed of the power iteration thus only depends on the quotient $\eta := |\lambda_2|/|\lambda_1|$ between the two largest Eigenvalues.

Repeated multiplication with A can thus be used to compute an Eigenvector corresponding to the largest Eigenvalue in the so-called power iteration first introduced by von Mises and Pollaczek-Geiringer [8]:

Algorithm 2.4: Power iteration

$x_0 \leftarrow$ initial guess for b_1
for $i = 1, 2, \ldots$ **do**
  $\tilde{x}_i \leftarrow A x_{i-1}$
  $x_i \leftarrow \tilde{x}_i / \|\tilde{x}_i\|$
end

The normalization step is necessary as the norm of the iterates x_i grows like λ_1^i. To get an approximation to the Eigenvalue λ_1, we can use the Rayleigh quotient

$$\rho_i = \frac{\langle x_i, A x_i \rangle}{\langle x_i, x_i \rangle} \overset{\|x_i\|=1}{=} \langle x_i, A x_i \rangle$$

which is equal to the Eigenvalue if x_i is an Eigenvector.

The power law iteration allows only the computation of the largest Eigenpair (λ_1, b_1), but we can use two observations to compute other Eigenpairs:

$$Ax = \lambda x \Leftrightarrow (A - sI)x = (\lambda - s)x \tag{2.8}$$

$$Ax = \lambda x \Leftrightarrow A^{-1} x = \lambda^{-1} x \text{ for invertible } A \tag{2.9}$$

If we have a good initial guess $\rho \approx \lambda_i$ for an Eigenvalue with Eigenvector x, i.e., $|\rho - \lambda_i| \ll |\rho - \lambda_j|$ for $i \neq j$, we can use a power iteration using $B := (A - \rho I)^{-1}$ instead of A, because $(\lambda - \rho)^{-1}$ is then the largest Eigenvalue of B with corresponding Eigenvector x. The exact computation of B would be numerically unstable due to the matrix inversion, so in practice we solve the linear equation $B\tilde{x}_i = x_{i-1}$ for $\tilde{x}_i$ instead.

We can even continuously update the guess ρ using the Rayleigh quotient to increase the convergence speed η, which gives us the so-called Rayleigh quotient iteration

[5] By replacing A with $\lambda_1^{-1} A$ and x with $c_1^{-1} x$

Algorithm 2.5: Rayleigh quotient iteration

$x_0 \leftarrow$ initial guess for b_1
$\rho_0 \leftarrow$ initial guess for λ_i
for $i = 1, 2, \ldots$ **do**
  Solve $(A - \rho_{i-1} I)\tilde{x}_i = x_{i-1}$ for $\tilde{x}_i$
  $x_i \leftarrow \tilde{x}_i / \|\tilde{x}_i\|$
  $\rho_i \leftarrow \langle x_i, A x_i \rangle$
end

For normal matrices, i.e. matrices satisfying $A^* A = AA^*$, the Rayleigh quotient iteration leads to a locally cubic convergence of the sequence (ρ_i) of Eigenvalue approximations.

Due to the similarities in the repeated multiplication to the power iteration, the Hessenberg matrix H_m computed during the Arnoldi process has similar Eigenvalues as A, so its entries can be used to compute an approximation to the spectrum of A, if we are interested in computing multiple Eigenpairs at the same time.

There is a second group of Eigenvalue solvers that are able to compute multiple Eigenpairs simultaneously without computing an intermediate matrix like H_m. An example for these so-called matrix-free solvers is the Locally Optimal Block Preconditioned Conjugate Gradient method (LOBPCG) introduced by Knyazev [9] that solves the generalized Eigenvalue problem

$$Ax = B\lambda x$$

with positive-definite B. This equation has only solutions with real Eigenvalues and B-orthogonal Eigenvectors. We know that this Eigenbasis X satisfies

$$AX = BX\Lambda, X^T BX = I \text{ with } \Lambda = \operatorname{diag}(\lambda_1, \ldots, \lambda_n)$$

Due to these properties, the B-orthogonal basis X of eigenvectors with $X^T BX = I$ minimizes the trace $\operatorname{tr}(X^T AX)$, which is exactly when we know that $X^T AX = X^T BX\Lambda = \Lambda$ holds. A possible approach to compute Eigenvectors corresponding to larger Eigenvalues of A is thus the application of an optimization algorithm to the optimization problem

$$\min_{X^T BX = I} \operatorname{tr}(X^T AX).$$

Naïve LOBPCG can have some issues with stability, but a recent work by Duersch et al. [10] showed how the orthogonalization can be improved in terms of robustness and efficiency to make LOBPCG more usable in practice.

3. Related Work

Many distributed graph algorithms share a similar, so-called *vertex-centric* structure based on alternating steps of local computation and data exchange with neighboring vertices. This lead to the development of several distributed graph frameworks like Pregel [11] and GraphLab [12]. As the generality provided by these frameworks would far exceed the scope of this work, we choose a model problem that somewhat realistically represents the communication and computation structure of many graph algorithms: Sparse matrix-vector multiplication (SpMV), i.e. the computation of the product

$$y = Ax$$

where A is a sparse matrix and x is one or multiple column vectors. Here, the graph corresponding to the sparsity pattern of A describes the data flow from entries of x to entries of y, where each edge amounts to a multiplication $a_{ij}x_j$ and every vertex to the accumulation of these contributions. Examples for such computation and neighborhood data exchange patterns are clustering algorithms like Label Propagation proposed by Raghavan et al. [13] or algorithms for the computation of closeness measures like the Algebraic Distance introduced by Chen and Safro [14].

3.1. Distributed SpMV

The simplest distributed-memory parallelizations of an SpMV computations $y := Ax$ with a square matrix A are based on a *symmetric* or *one-dimensional partitioning* of the matrix and vector – the elements of both the input and output vector are partitioned the same way. While this approach provides less flexibility in terms of load balancing than more sophisticated types of partitioning, it has an important advantage for a large class of applications: If the SpMV computation has to be iterated multiple times – for example in an iterative solver for linear systems – there is no need to exchange data between iterations, as the input and output vector partitions coincide.

There are two natural ways to extend such a partition of x and y to a partition of the matrix A – by partitioning A *row-wise* or *column-wise*. These two partitioning approaches lead to the following distributed SpMV implementation variants:

We consider the graph $G = (V, E)$ corresponding to A and a partition $V = V_1 \dot\cup \ldots \dot\cup V_k$ of its vertices (and thus x and y). This means that we compute the SpMV product on k distributed nodes, where the ith node stores only the entries of x and y for vertices from V_i, as well as the corresponding rows/columns from A. Based on this partition and for a fixed node i, we call all vertices $v \in V_i$ *local* and all other vertices $v \notin V_i$ *non-local*. We can

compute a single entry of y corresponding to a vertex u as

$$y_u = \overbrace{\underbrace{\sum_{\substack{uv \in E \\ v \in V_i}} w_{uv} x_v}_{\substack{\textit{compute-first:} \\ \text{store locally}}}}^{\substack{\textit{send-first:} \\ \text{store locally}}} + \underbrace{\sum_{\substack{uv \in E \\ v \notin V_i}} w_{uv} \overbrace{x_v}^{\text{receive}}}_{\text{receive from each part}}$$

Here we see the impact of the partitioning of A: if we partition row-wise, we only need to consider contributions from all vertices to local vertices, thus we need all relevant entries of x, but only the local entries from y (*send-first*). If we partition A column-wise, we only need to store the local entries of x, but instead have to compute and exchange the contributions from the local vertices to all entries of y (*compute-first*). Figure 3.1 and Algorithm 3.1 shows these two partitioning and implementation variants from the view of a single boundary vertex and a parallel process, respectively. They are based on the distributed SpMV implementation described by Uçar and Aykanat [15]. The implementation can either send the entries that are required by other tasks before executing the computations (*send-first*) or compute the local contributions to all non-local vertices and send them instead (*compute-first*) – the computation simply gets shifted from the sender to the receiver, which causes no change in the communication volume and computation balance for a matrix with symmetric sparsity pattern, but can have a large impact in non-symmetric cases.

Algorithm 3.1: Distributed SpMV

Parallel process storing part p with its local vectors x_l, y_l and corresponding rows/columns $A_{\bullet l}/A_{l\bullet}$ of the whole matrix A.

Send-first

for $q = 1, \ldots, k,\ q \neq p$ **do**
- $x_{snd} \leftarrow \text{gather}(x_l, N^{\leftarrow}(V_q) \cap V_p)$
- Send x_{snd} to Part q
- Receive x_{rcv}^q from Part q

end

$y_l \leftarrow A_{l\bullet} \cdot (x_l | x_{rcv}^1 | \cdots | x_{rcv}^k)^T$

Compute-first

$(y_l | y_{snd}^1 | \cdots | y_{snd}^k)^T \leftarrow A_{\bullet l} \cdot x_l$

for $q = 1, \ldots, k,\ q \neq p$ **do**
- Send y_{snd}^q to Part q
- Receive y_{rcv} from Part q
- $y_l \leftarrow y_l + \text{scatter}(y_{rcv}, N(V_q) \cap V_p)$

end

It is easy to see that the send-first and compute-first implementation are in a sense dual to each other: aside from the reversed order of communication and computation, send-first stores all columns of A and uses a sparse gather operation to distribute the local data to other parts, while compute-first stores all rows of A and uses a sparse scatter operation to combine the contributions of other parts to the local data.

$a_{ij} \cdot x_j$ $\sum$

x_j y_i

$$\tilde{y}_k \leftarrow \sum_{j \text{ local}} a_{kj} \cdot x_j$$
send $\tilde{y}_k$
receive $\tilde{y}_i^k$
$$y_i \leftarrow \sum_{j \text{ local}} a_{ij} \cdot x_j + \sum_k \tilde{y}_i^k$$

compute-first/column-wise

x_j $\sum a_{ij} \cdot$

x_j y_i

send x_i
receive x_k
$y_i \leftarrow \sum_j a_{ij} \cdot x_j$

send-first/row-wise

Figure 3.1.: Two possible implementations of distributed SpMV with 1D partitioning, where i represents the vertex currently being processed and k represents a neighboring vertex in another part.

3.2. Communication Models

Central to every distributed graph algorithm is the assignment of vertices and edges to the different parallel tasks. This load balancing task strongly influences the runtime and scalability of a distributed algorithm. Usually, two objectives need to be optimized for such a load balancing approach to achieve a good result: 1. the amount of messages exchanged between processes should be minimal – both in terms of message size and message count – and 2. each process should have roughly the same amount of work to perform locally.

These two objectives nicely translate into the objectives and constraints used in (hyper-) graph partitioning: The (hyper-)graph cut size gives a good (or even exact) model for the communication volume, while the balance constraint can be used to ensure a balanced distribution of work when assigning to each vertex a weight corresponding to the amount of computation it generates.

We want to look at three partitioning approaches that can be used in distributed SpMV implementations: Symmetric graph partitioning, column-wise and row-wise hypergraph partitioning.

Definition 3.1: Symmetric graph partitioning

For the square sparse matrix A we build the corresponding graph G. To each vertex v we assign the weight $\deg(v) + 1$ (compute-first) or $\deg^{\leftarrow}(v) + 1$ (send-first). We partition the undirected graph G^{bi} corresponding to G with these vertex weights.

Graph partitioning is generally only considered on undirected graphs, which is why we need to use G^{bi}. The weight of a vertex v corresponds to the number of multiply-and-add operations that need to be executed to compute y_v (send-first) or the number of multiply-and-add operations that involve x_v (compute-first). The $+1$ is necessary to avoid vertices of weight 0 during the partitioning, as there is at least some overhead involved with each vertex. In case of the compute-first implementation, the number of operations is not strictly correct, as the partial results $\tilde{y}_i^k$ still need to be accumulated, but for larger graphs and not too many parts, this overhead is usually negligible. A balanced graph partition thus means

that every part has roughly the same number of floating point operations to compute, which should lead to a balanced computation runtime (ignoring the parallelization potential and memory locality etc.). The graph partitioning approach is however not able to exactly model the communication volume: In the send-first model, when we have three vertices u, v, w, with v and w in the same part and cut edges $u \longrightarrow v, u \longrightarrow w \in E$, the data from x_u only needs to be transmitted to the part storing v and w once, but the graph partition nevertheless counts both cut edges as communication. The dual observation holds for the compute-first model, where two edges from one part to the same vertex only cause a single unit of communication.

To describe the improved hypergraph models, we introduce some additional notation. For the sparse square matrix A, we write

$$nz(A, i, \cdot) = \{j \mid A_{ij} \neq 0\}, \quad nz(A, \cdot, j) = \{i \mid A_{ij} \neq 0\}$$

for the set of non-zero column/row indices in row/column i/j and $nnz(\cdots) = |nz(\cdots)|$ for the corresponding size. Using this notation, we can give exact hypergraph models for the communication volume in both implementations. They are described in more detail in the work by Çatalyürek and Aykanat [16].

Definition 3.2: Column-wise hypergraph partitioning

We build a hypergraph with vertices $\mathcal{V} = \{c_1, \ldots, c_n\}$ corresponding to the columns with weights $nnz(A, \cdot, j)$ for c_j and nets $\mathcal{N} = \{r_1, \ldots, r_n\}$ corresponding to the rows, where $r_i = \{c_j \mid j \in nz(A, i, \cdot)\}$ is the set of columns with non-zero entries in row i.

When we use the hypergraph partition from the column-wise partitioning approach in a distributed SpMV implementation using the compute-first approach, the (total or bottleneck) connectivity cut size of the partition is equal to the communication volume of the all-to-all operation.

Definition 3.3: Row-wise hypergraph partitioning

We build a hypergraph with vertices $\mathcal{V} = \{r_1, \ldots, r_n\}$ corresponding to the rows with weights $nnz(A, i, \cdot)$ for r_i and nets $\mathcal{N} = \{c_1, \ldots, c_n\}$ corresponding to the columns, where $c_j = \{r_i \mid i \in nz(A, \cdot, j)\}$ is the set of rows with non-zero entries in column j.

Like with the column-wise approach, we can use the hypergraph partition to partition the vectors x and y during a distributed SpMV computation. The (total or bottleneck) connectivity objective of the hypergraph partition is then again equal to the communication volume of the send-first implementation.

In case not only the total communication volume needs to be optimized, but the bottleneck volume and number of messages is also important, Acer et al. [17] show how the recursive bisection approach to hypergraph partitioning can be used in hypergraph communication models to simultaneously balance total volume V, bottleneck volume B, computation volume C and the number of messages M by minimizing an objective of the

form

$$V + \beta M, \text{ while keeping } C + \alpha B \text{ balanced between parts.}$$

They achieve this by, after each recursive bisection step, increasing the weight of all nodes by the $\alpha-$weighted communication volume they caused to their neighboring part, and introducing a $\beta-$weighted net for each message that needs to be passed between these parts.

Finally, there are also two-dimensional partitioning approaches that partition the entries of x and y and even the individual computations $a_{ij}x_j$ independently. These can be divided into three groups according to Çatalyürek et al. [18]: Firstly, there is the fine-grained two-dimensional partitioning approach that models each multiplication $a_{ij}x_j$ corresponding to a non-zero entry as a vertex and introduces one net per row and column that connects all non-zeros from this row/column. This approach allows for the greatest flexibility in load balancing, but significantly complicates the data exchange. Secondly, jagged partitioning first partitions the rows or column of the sparse matrix and then partitions the columns or rows of the resulting sub-matrices independently. Finally, checkerboard partitioning approaches start like the jagged approach, but partition the columns or rows of all sub-matrices coherently, which leads to a checkerboard-type pattern in the reordered matrix.

Aside from hypergraph partitioning, the fine-grained two-dimensional partitioning approach shares a few similarities with edge partitioning methods introduces in the PowerGraph framework by Gonzalez et al. [19]. Recent work by Schlag et al. [20] showed how such edge partitions can be computed efficiently in parallel without the need to fall back to – usually comparably slow – hypergraph partitioning methods.

3.3. (Hyper-)Graph Partitioning Algorithms

Even though graph and hypergraph partitioning are NP-complete problems, in practice, there are many heuristics that are able to efficiently and effectively optimize many partitioning instances. These heuristics can be roughly classified as follows:

- *Initial partitioning* heuristics can be used to compute a bi- or multi-way partition of an input (hyper-)graph without a preexisting partition. Among the most well-known initial partitioners are *spectral partitioning methods* based on the Eigenvalues and -vectors of matrices associated with the input graph, like the Fiedler Eigenvector [21] of the Laplacian matrix. Other approaches include traversal-based approaches, random partitions or even exact solutions based on Integer Linear Program formulations [22]. Initial partitioning algorithms are usually only used for small input graphs, but there it may be sensible to even employ an ensemble of many different initial partitions, from which we can choose the partition with the best objective. When an initial partitioner is only able to compute bisections, but a multi-way partition is needed, the *recursive bisection* paradigm is used to recursively split the input until we have enough parts.

- *Local search* algorithms take an existing partition and iteratively improve its objective function or balance by moving vertices between parts. Many of these algorithms are

based on the Kernighan-Lin heuristic [23]. The heuristic was subsequently improved in terms of runtime by Fiduccia and Mattheyses [24] as a linear-time algorithm. Other approaches are based on flow networks [25] [26] that minimize the objective in the form of a min-cut problem. While most of these local search algorithms are formulated for two-way partitions, they can be adapted for multi-way partitions either by executing them for every pair of parts, or by combining them by iteratively choosing the "best" pair of parts that promises the largest objective improvements.

- *Coarsening* algorithms recursively compute a coarser representation of a large input (hyper-)graph until it is small enough to be partitioned using an initial partitioner. Typical coarsening approaches are either clustering- or matching-based [27]. While graphs with a regular structure often profit from matching-based coarsening, clustering or community detection approaches can work very well on network graphs, as Meyerhenke et al. [28], and Heuer and Schlag [29] recently showed for graphs and hypergraphs, respectively.

Coarsening, initial partitioning and local search (or refinement) algorithms can be combined in the *multilevel* paradigm to compute partitions even for large (hyper-)graphs. In this approach, the input is first coarsened until initial partitioning becomes feasible. The initial partition can then be unpacked recursively, with local searches improving the intermediate partitions until we reach the finest hierarchy level.

This multilevel approach is employed by all popular graph and hypergraph partitioning frameworks. Among them are `KaHIP` [30] and `METIS` [31] for graph partitioning and `KaHyPar` [32], `PaToH` [33], `hMETIS` [34] and `Zoltan` [35] for hypergraphs.

3.4. Network Graphs

The main motivation for studying the implementation of distributed graph algorithms with replication lies in the structure of a large class of graphs, the so-called network graphs. Their irregular structure compared to other scientific computing applications makes it sensible to introduce a special treatment for small parts of the graph which speed up the whole distributed algorithm.

Graphs with a network structure occur everywhere, in natural contexts like biological networks, in sociological concepts like social networks as well as in technological contexts in the form of communication networks and web graphs. Despite these numerous backgrounds, they share a number of common properties that have been observed in all of these areas. Networks often show heavily skewed degree distributions, meaning they have only few vertices of high degree, while most vertices have a very low degree. One common assumption is that their vertex degrees are approximately distributed according to a power-law distribution (which is why they are also called scale-free graphs):

Definition 3.4: Power-law distribution

A random variable X follows a (discrete) power-law distribution if

$$\mathbb{P}(X = d) = c \cdot d^{-k}$$

for suitable constants $c, k > 0$.

Empirically, power-law distributions can be identified using histograms or a log-log plot of the (decreasingly) sorted sequence of values against their rank in the sequence, which we use throughout this thesis to demonstrate the scale-free property of graphs. However, statistical testing for power-law distributions is more complex, and Clauset et al. [36] [37] even showed that exact power-law distributions are surprisingly rare in real-world networks. Still, we don't actually need exact power-law degree distributions for our ideas to work, a heavily tailed degree distribution with few vertices of large degree can be sufficient.

Network graphs show very interesting connectivity structures: They usually have a small diameter, i.e., most pairs of vertices are connected by a short path through the graph, which is also known as the *Small-World Effect*. This has been observed in E-Mail communication [38] as well as online social networks [39]. Networks usually have a large connected component that contains most of the vertices. This component can only be disconnected by removing a certain threshold number of vertices, after which the component decomposes into many smaller components. If this situation occurs, the resulting graph usually gives a much larger degree of freedom to partitioning algorithms, as independent components can be moved without changing the cut metrics, especially for improving the imbalance. This resistance against disconnection of a graph is called the percolation threshold [40] and can be rather high compared to other, more regular graphs like grids. Another notable measure that can be used to identify network-like properties is the clustering coefficient, which measures the probability that adjacency relations are transitive, i.e., the probability that a path $u - v - w$ implies the existence of the edge uw.

Definition 3.5: Clustering coefficient

The clustering coefficient of a graph $G = (V, E)$ is given by

$$C(G) = \frac{\text{Number of closed triplets in } G}{\text{Number of triplets in } G} = \frac{|\{(u, v, w) \in V^3 \mid uv, vw, uw \in E\}|}{|\{(u, v, w) \in V^3 \mid uv, vw \in E\}|}$$

Holland and Leinhardt [41] showed that this transitivity often occurs in social networks, which was again corroborated by Watts and Strogatz [38].

There are several popular models for the generation of random network graphs. Here we want to focus on two well-researched models that generate graphs with some of the aforementioned properties, either by construction or inherently.

The Barabási-Albert random graph model [42] is based on a preferential attachment process. Starting from a small seed graph, we iteratively add one vertex at a time and connect it to δ vertices from the previous graph chosen at random with probabilities proportional to their degree. This models a purely random attachment of new members

of a network to the existing structure with a preference to attach to high-degree vertices and produces a network with a power-law degree distribution of the form d^{-3}. However, Barabási-Albert graphs don't reproduce the high clustering coefficient observed in real-world networks, as their clustering coefficient decays with increasing network size [43]. The preferential attachment process with parameter δ can be implemented as follows, based on Batagelj and Brandes [44]:

Algorithm 3.2: Preferential attachment graph generation

Seed graph $G_0 = (V_0, E_0)$ with n_0 vertices growing to n vertices:
- $V \leftarrow V_0,\ E \leftarrow E_0$
- **for** $u = n_0, \ldots, n-1$ **do**
  - $V \leftarrow V \cup \{u\}$
  - **for** $i = 0, \ldots, \delta - 1$ **do**
    - Choose $e \in E$ uniformly at random
    - Choose vertex v of e uniformly at random
    - $E \leftarrow E \cup \{uv\}$
  - **end**
- **end**

Krioukov et al. [45] observed that many network graphs can be embedded into the hyperbolic plane $\mathbb{H}^2_\zeta$ with constant curvature $-\zeta^2 < 0$ such that vertices that are placed close to each other on $\mathbb{H}^2_\zeta$ are connected in the graph with high probability. This gave rise to the random hyperbolic graph model for networks, also called the hyperbolic geometric graph model: We choose n points at random in the Poincaré disk representation of $\mathbb{H}^2_\zeta$ with uniform angle θ and radius r according to the probability density

$$p(r) = \alpha \frac{\sinh \alpha r}{\cosh \alpha R - 1}$$

for parameters $\alpha, R > 0$. The probability that two vertices at coordinates (θ, r) and (θ', r') are then connected is based on their distance

$$d = \cosh^{-1}(\cosh \zeta r \cosh \zeta r' - \sinh \zeta r \sinh \zeta r' \cos |\theta - \theta'|)/\zeta$$

While one could use a naïve implementation of the above description as an $O(n^2)$ algorithm to build a random hyperbolic graph, there exist improved implementations that work in subquadratic time by exploiting the geometric structure of the point set, like the approach by von Looz et al. [46]. Figure 3.2 shows an example for such a hyperbolic geometric graph embedded into $\mathbb{H}^2_\zeta$.

Krioukov et al. [45] also showed that random hyperbolic graphs follow a power-law degree sequence, and Gugelmann et al. [47] proved that they additionally have a clustering coefficient close to 1 with high probability for $\alpha > \frac{1}{2}$.

Figure 3.2.: Random hyperbolic graph with 200 vertices, $\alpha = 0.8$ and $R = 5$ in the Klein disk model (left) and Poincaré disk model (right).

3.5. Vertex Centrality

Much work has gone into the structural analysis of network graphs from many different viewpoints. An important aspect are metrics that measure the *centrality* of a vertex. A simple centrality measure is the degree of a vertex, but over time, many different measures were developed: Bavelas [48] introduced the closeness metric, i.e., the reciprocal sum of all shortest-path distances to other vertices, and Freeman [49] proposed to use a vertex's betweenness, i.e., the fraction of all shortest paths passing through it. A number of centrality measures are based on the Eigenvectors of the adjacency matrix or other related matrices of the network graph. The last important group of centrality measures is based on random walks through the graph, where the probability to arrive at a certain vertex is used to measure its centrality. Arguably the most popular random walk approach is the *PageRank* model by Brin et al. [50][51] which is used in part to rank results in the Google search engine.

Initially, PageRank was not formulated as a random walk, but instead as the steady-state of an iterative process that distributes the PageRank of every web-page evenly among the pages pointed to by outgoing hyperlinks, so the updated PageRank of a page is the sum of the contribution along all incoming hyperlinks:

$$\text{PageRank}(v) \leftarrow \sum_{uv \in E} \frac{1}{\deg(u)} \text{PageRank}(u)$$

The equivalent random-walk formulation of PageRank models the behavior of a "random surfer" visiting a series of web pages by randomly following links, and frequently restarting at a random page. If the graph $G = (V, E)$ represents a set of web pages as vertices and a hyperlink from web page u to web page v by the directed edge uv, we can formalize the behavior of the random surfer as follows: The surfer moves from vertex u to vertex v with

probability

$$p_{uv} = \begin{cases} (1-\omega)\frac{1}{\deg(u)} + \omega\frac{1}{|V|}, & \text{if } uv \in E \\ \omega\frac{1}{|V|} & \text{otherwise} \end{cases},$$

where $\omega \in [0, 1]$ is the restart probability, i.e., the probability that the surfer restarts from a random vertex of G.

This random walk can naturally be interpreted as a discrete, finite Markov chain, which for $\omega > 0$ is even irreducible. Thus we can use its transition probability matrix $P = (p_{uv})$ to describe its unique stationary distribution π as its stochastic left Eigenvector

$$P^T\pi = \pi \text{ such that } \pi \geq 0 \text{ and } \|\pi\|_1 = 1$$

This stationary distribution gives the probability to find the random surfer at any given vertex, and can thus be used as a centrality measure.
In practice, this dense matrix P is separated into $P = (1-\omega)L + \omega E$ with

$$L_{uv} = \begin{cases} 1/\deg(u), & \text{if } uv \in E \\ 0 & \text{otherwise} \end{cases} \quad \text{and} \quad E_{uv} = 1$$

This way we can compute the matrix-vector product P^Tx for stochastic x as

$$P^Tx = (1-\omega)L^Tx + \omega\mathbf{1},$$

only having to evaluate the sparse matrix-vector product L^Tx. We will later use the calculation of the PageRank vector of a distributed graph using the Power iteration and Rayleigh quotient iteration as an application example for distributed SpMV.

4. Data Replication

The sparse all-to-all communication employed in distributed SpMV implementations, when combined with a high-quality (hyper-)graph partition, provides a good trade-off between computation imbalance and communication volume in regular graphs/matrices, which are usually found in scientific computing applications. In these applications, no single or few vertices cause a large amount of computation or communication by themselves. By contrast, in network graphs, the connectivity (before partitioning) or communication volume (after partitioning) can be strongly lopsided towards a few important vertices. In consequence, these vertices are both difficult to handle in partitioning algorithms (as moving any of them in a local optimization phase can cause large changes in the cut size and imbalance) and the distributed implementation itself (where they can be responsible for a larger number of p2p messages).

To address these issues, we want to introduce a special treatment for these "important" vertices: We trade off communication for computation by storing the data belonging to these vertices in every distributed node – thus *replicating* it – and use suitable collective operations instead of p2p communication to collect the contributions from other vertices.

We first want to show how the distributed SpMV implementation from Section 3.1 can be extended to handle replicated vertices. We then also extend the communication model based on 1D partitioning from Section 3.2 to the replicated case and propose several heuristics which can be used to choose the vertices to be replicated. Finally, we show how our approach to data replication could be extended using a two-level or even multilevel partitioning.

4.1. Basic Idea

We will first illustrate how the replicated vertices can be incorporated and updated in the *send-first* model

$$y_u = \overbrace{\sum_{\substack{uv \in E \\ v \in V_i}} w_{uv} x_v + \sum_{\substack{uv \in E \\ v \notin V_i}} w_{uv}}^{\text{stored locally}} \overbrace{x_v}^{\text{p2p}}$$

For a partition $V = V_R \dot{\cup} V_1 \dot{\cup} \ldots \dot{\cup} V_k$ where V_R denotes the set of replicated vertices, we use k distributed nodes where node i stores only data relevant to vertices in $V_R \cup V_i$ – the data from V_R is then replicated to every node. When computing the entries of y for a non-replicated vertex $u \in V_i$, we can simply treat replicated vertices like local vertices,

which do not need to be exchanged with other nodes:

$$y_u = \overbrace{\sum_{\substack{uv\in E\\ v\in V_R}} w_{uv}x_v + \sum_{\substack{uv\in E\\ v\in V_i}} w_{uv}x_v + \sum_{\substack{uv\in E\\ v\notin V_R\cup V_i}} w_{uv}}^{\text{stored locally}} \overbrace{x_v}^{\text{p2p}}$$

When $u \in V_R$ is replicated, we can locally compute the contributions of V_R and V_i to u, but the contributions from other parts V_j needs to be collected as well. For this, we can use an All-Reduce collective operation: We locally compute the contributions of V_i to u and All-Reduce them to arrive at the sum of contributions from $V_1 \cup \cdots \cup V_k$ at every distributed node. Formally, this means we compute

$$y_u = \overbrace{\sum_{\substack{uv\in E\\ v \text{ replicated}}} w_{uv}x_v}^{\text{stored locally}} + \overbrace{\sum_{\substack{uv\in E\\ v \text{ local}}} w_{uv}x_v + \sum_{\substack{uv\in E\\ v \text{ non-local}}} w_{uv}x_v}^{\text{combined via All-Reduce}}$$

To simplify the notation, we will assume like in section 2.5 that the columns and rows of A are reordered such that x and y are partitioned into consecutive ranges, which gives us the following block-structure of A, where x_R and y_R store the entries belonging to replicated vertices:

$$\begin{pmatrix} y_R \\ y_1 \\ \vdots \\ y_k \end{pmatrix} = \begin{pmatrix} A_{RR} & A_{R1} & \cdots & A_{Rk} \\ A_{1R} & A_{11} & \cdots & A_{1k} \\ \vdots & \vdots & \ddots & \vdots \\ A_{kR} & A_{k1} & \cdots & A_{kk} \end{pmatrix} \cdot \begin{pmatrix} x_R \\ x_1 \\ \vdots \\ x_k \end{pmatrix}$$

Based on this partition, we can amend Algorithm 3.1 to incorporate the replicated vertices:

Algorithm 4.1: Distributed SpMV with replication

Parallel process storing part p with its local vectors x_l, y_l, replicated vectors x_R, y_R and corresponding blocks $A_{RR}, A_{Rl}, A_{l\bullet}/A_{\bullet l}$ of the whole matrix A.

Send-first

- **for** $q = 1, \ldots, k,\ q \neq p$ **do**
  - $x_{snd} \leftarrow \text{gather}(x_l, N^{\leftarrow}(V_q) \cap V_p)$
  - Send x_{snd} to Part q
  - Receive x^q_{rcv} from Part q
- **end**
- $y_R \leftarrow A_{RR}x_R + \text{allreduce}(A_{Rl}x_l)$
- $y_l \leftarrow A_{l\bullet} \cdot (x_R | x_l | x^1_{rcv} | \cdots | x^k_{rcv})^T$

Compute-first

- $(y_l | y^1_{snd} | \cdots | y^k_{snd})^T \leftarrow A_{\bullet l} \cdot x_l$
- $y_l \leftarrow y_l + A_{lR}x_R$
- $y_R \leftarrow A_{RR}x_R + \text{allreduce}(A_{Rl}x_l)$
- **for** $q = 1, \ldots, k,\ q \neq p$ **do**
  - Send y^q_{snd} to Part q
  - Receive y_{rcv} from Part q
  - $y_l \leftarrow y_l + \text{scatter}(y_{rcv}, N(V_q) \cap V_p)$
- **end**

The differences to Algorithm 3.1 are in the treatment of data as local and non-local: 1. for the computation of y_l we treat x_R and x_l as local data, but for the computation of y_R we need to use the All-Reduce operation and 2. for the p2p-communication, we only exchange data from non-replicated vertices x_l/y_l.

4.2. Communication Model

We now want to show how the one-dimensional hypergraph models for the SpMV communication volume introduced in section 3.1 can be extended to account for replicated nodes. We start off with a square matrix A and its corresponding graph $G = (V, E)$. For simplicity, we describe the operation only the terms of vertices and edges of G, but keep in mind that these operations correspond to changes in the rows/columns and entries of A.

When replicated, a vertex no longer needs to take part in the all-to-all operation, which leads to reduced communication both for the vertex itself as well as all neighboring vertices from other parts. The associated multiplication with the A_{RR} are executed on every node and can simply be ignored in the balancing constraints. In case we had a large number of replicated vertices, it can be necessary to modify the imbalance limit ε. The computation $A_{Rl}x_l$ would have occurred in the compute-first model anyways, in the send-first model, we need to incorporate it internally. Conversely, the operation $A_{lR}x_R$ would have been computed in the send-first model, but needs to additionally be considered in the compute-first model.

In total, we need to make the following changes in the hypergraph models to incorporate a set R of replicated vertices:

Definition 4.1: 1D hypergraph partitioning with replicated vertices

We start off with a row-wise/column-wise k-way hypergraph partition $\mathcal{V} = \mathcal{V}_1 \cup \cdots \cup \mathcal{V}_k$ of a matrix A and modify it as follows:

	Send-first	Compute-First
Vertices/Pins	Remove all vertices/pins corresponding to replicated rows/columns	
Nets/Hyperedges	Remove all nets corresponding to replicated columns/rows	
Vertex weights	Add $nnz(A, R, i)$ to $w(r_i)$	Add $nnz(A, j, R)$ to $w(c_j)$
Imbalance limit	With $w_R = nnz(A, R, R)$ and $\overline{w} = nnz(A)/k$ replace ε by $\varepsilon' = \varepsilon \frac{\overline{w}}{\overline{w} - w_R/k}$ [a]	

[a] This transformation is only valid under the assumption that the replication itself is balanced, i.e., the amount of computation caused by replicated vertices was the same in every part before replication.

To compute the total communication volume, we assume that the runtime ratio between an All-Reduce operation and a single p2p message on a single vector entry is $\alpha : 1$. The

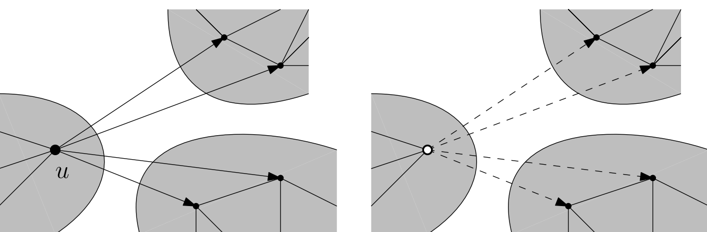


Figure 4.1.: Effect of replication on the bottleneck communication volume

communication volume caused by the part p is then, in terms of the hypergraph cut connectivity (Definition 2.7):

$$c(\mathcal{V}_p) + \alpha|R| \tag{4.1}$$

The modified hypergraph models incorporating replication give us an additional advantage: If we remove the replicated vertices and nets from the hypergraph, we can re-partition the modified hypergraph without these usually "heavy" vertices, which can make it easier for the hypergraph partitioner to satisfy the balance criterion, thus potentially speeding up the partitioning or leaving more freedom for optimizing the communication volume. In extreme cases, the replication could even cause the remaining graph to become disconnected, which again could simplify partitioning, as long as the partitioning algorithms can deal with disconnected graphs and the resulting bin-packing problem.

Based on this communication model, we can now turn to the choice of replicated vertices, which has a great impact on the effectiveness of replication, both for speeding up the partitioning as well as for optimizing the communication time.

4.3. Replication Heuristics

The most important ingredient for the use of replication is a good choice of replicated vertices. Their replication should greatly reduce the total or bottleneck communication volume and potentially make the remaining graph more partitioner-friendly.

First of all, we want to show the impact of a single replication on the overall communication volume: as Figure 4.1 shows, the replication of a vertex u has an effect on both the part V_i containing u as well as all parts that are connected to u by a cut-edge. The communication volume of V_i gets decreased by the number of other parts V_j that u is connected to, while at the same time the communication volume of all neighboring parts V_j gets decreased by 1. When we want to optimize the bottleneck communication volume in a graph, this has an important effect: With a good choice of "important" vertices, we can simultaneously decrease the communication volume of many parts.

We propose four replication heuristics for such a choice of vertices: Three heuristics based solely on the distribution of certain vertex properties, and a fourth heuristic that tries to greedily optimize the communication volume based on the exact communication model from section 4.2.

First we start of with a most naïve heuristic: As network graphs approximately follow a power-law degree distribution, it stands to reason that the vertex degree is a good predictor

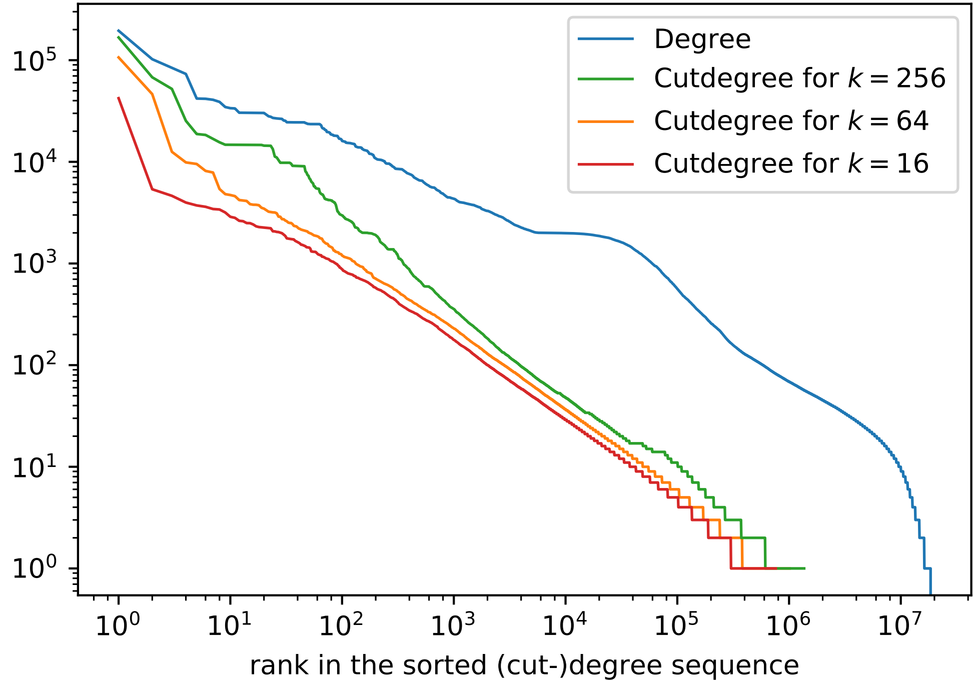


Figure 4.2.: Degree and cut-degree distribution in the `uk-2002` web graph

for the connection of a vertex to the whole graph. We would expect that vertices of high degree also leads to a high cut degree and connectivity when partitioned in a graph or hypergraph model. Figure 4.2 and Table 5.1 show a justification for this approach in that for a graph with an approximate power-law degree distribution, the cut-degrees of a partitioned graph follow such a power-law degree as well, especially for large cut-degrees.[1] This replication heuristic would have the advantage that we can compute it even without partitioning the graph. Other candidates for such an importance metric we can use to choose vertices for replication are the cut-degree and cut connectivity of a vertex, as they give a good prediction for the communication volume reduction when replicating this vertex. Thus we have the following replication heuristics purely based on per-vertex metrics:

1. **Degree Replication**
   We replicate the m vertices with the largest degree in G

2. **Cut-Degree Replication**
   We replicate the m vertices with the largest cut-degree in the partitioned G

3. **Connectivity Replication**
   We replicate the m vertices with the largest cut connectivity in the partitioned G

[1] We can identify a power-law distribution as a straight line in a log-log plot of the value against their decreasing rank

Finally, we want to show how the communication model from section 4.2 can be used in a greedy optimization approach attempting to minimize the communication time based on the communication volume:

Algorithm 4.2: Greedy replication

Based on a graph $G = (V, E)$ with a k-way partition $V = V_1 \cup \dots \cup V_k$.
Compute the cut connectivity $c(v)$ for all vertices $v \in V$
Compute the cut connectivity $c(V_i)$ for all parts $i = 1, \dots, k$
$R \leftarrow \emptyset$
for $i = 1, \dots, m$ **do**
  $p \leftarrow \text{argmax}_{i=1}^{k} c(V_i)$
  $u \leftarrow \text{argmax}_{u \in V_p} c(u)$ (resp. $\text{argmax}_{u \in V} c(u)$)
  $R \leftarrow R \cup \{u\}$
  Update $c(v)$ for neighbors of u
  Update $c(V_i)$ for p and neighboring parts of u
end

This replication approach has the advantage that it continuously updates the communication volume of the remaining unreplicated vertices and thus might be able to better adapt to changes in the graph structure due to the replication. These updates can be implemented using two combined addressable priority queues for parts and vertices, ordered by $c(V_i)$ and $c(v)$ respectively.

However, the greedy replication approach in itself can produce sub-optimal results in terms of the bottleneck communication volume – as the experimental results in the communication model will later show. The reason for this can be understood based on an extreme case: we want to look only at a single replication step in a graph of the following structure, visualized in Figure 4.3. We have a bottleneck part p, where the vertex u has the largest connectivity and v has a lower connectivity. u is only connected to c_1 parts with very low communication volume, but there are $c_2 < c_1$ other bottleneck parts with the same communication volume as p that are all connected to v. Replicating u would not lower the bottleneck volume, as it only lowers the communication volume of p and the parts of lower communication volume, but the c_2 parts neighboring v still cause the overall bottleneck. On the other hand, replicating v would lower the volume of part p by c_2 and the volume of all neighboring parts by 1, which decreases the total bottleneck volume by 1 as well.

It should be noted that for all replication heuristics we proposed here, we only looked at the communication volume of the resulting SpMV implementation. However, replication can also have a large impact on the amount and balance of computation that has to be executed on every distributed node. The aforementioned replication heuristics can of course also optimize the computation as a side-effect, but that was not the central goal for their choice.

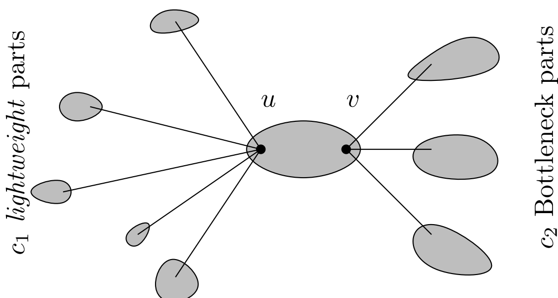


Figure 4.3.: Sketch of a counterexample graph showing the non-optimality of greedy replication

4.4. Two-Level Replication

In this section, we want to summarize how the current approach to replication can be extended based on a two-level graph partition, while also showing the basic tools necessary to extend this to a true multi-level approach. Two-level replication has the potential advantage that we can replicate vertices that would only cause a lot of communication within a small group of parts in a one-level replication scheme.

We start off with a graph $G = (V, E)$ partitioned with replicated vertices $V = V_R \dot{\cup} V_1 \dot{\cup} \dots \dot{\cup} V_{k_1}$. Each part corresponds to an induced subgraph $G_i = G[V_i]$, which can again be partitioned with replication in a k_2-way partition $V_i = V_{R_i} \dot{\cup} V_{i1} \dot{\cup} \dots \dot{\cup} V_{ik_2}$.

We can use this two-level replicated partition to implement a distributed SpMV as follows: Arranging the distributed nodes in a $k_1 \times k_2$ grid, the node at coordinates (i, j) stores the data relevant to vertices from $V_R \cup V_{R_i} \cup V_{ij}$. As before, we reorder the entries of x and y such that the vertex ranges $V_R, V_{R_1}, V_{11}, \dots, V_{1k_2}, V_{R_2} V_{21} \dots V_{k_1k_2}$ are consecutive:

$$
\begin{pmatrix} y_R \\ y_{R_1} \\ y_{11} \\ \vdots \\ y_{1k_2} \\ y_{R_2} \\ y_{21} \\ \vdots \\ y_{k_1k_2} \end{pmatrix}
=
\begin{pmatrix}
A_{RR} & \overbrace{A_{RR_1} & A_{R11} & \cdots & A_{R1k_2}}^{\text{corresponds to } V_1} & \overbrace{A_{RR_2} & A_{R21} & \cdots & A_{Rk_1k_2}}^{\text{corresponds to } V_2, \dots, V_{k_1}} \\
A_{R_1R} & A_{R_1R_1} & A_{R_111} & \cdots & A_{R_11k_2} & A_{R_1R_2} & A_{R_121} & \cdots & A_{R_1k_1k_2} \\
A_{11R} & A_{11R_1} & A_{1111} & \cdots & A_{111k_2} & A_{11R_2} & A_{1121} & \cdots & A_{11k_1k_2} \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\
A_{1k_2R} & A_{1k_2R_1} & A_{1k_211} & \cdots & A_{1k_21k_2} & A_{1k_2R_2} & A_{1k_221} & \cdots & A_{1k_2k_1k_2} \\
A_{R_2R} & A_{R_2R_1} & A_{R_211} & \cdots & A_{R_21k_2} & A_{R_2R_2} & A_{R_221} & \cdots & A_{21k_1k_2} \\
A_{21R} & A_{21R_1} & A_{2111} & \cdots & A_{211k_2} & A_{21R_2} & A_{2121} & \cdots & A_{21k_1k_2} \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\
A_{k_1k_2R} & A_{k_1k_2R_1} & A_{k_1k_211} & \cdots & A_{k_1k_21k_2} & A_{k_1k_2R_2} & A_{k_1k_221} & \cdots & A_{k_1k_2k_1k_2}
\end{pmatrix}
\cdot
\begin{pmatrix} x_R \\ x_{R_1} \\ x_{11} \\ \vdots \\ x_{1k_2} \\ x_{R_2} \\ x_{21} \\ \vdots \\ x_{k_1k_2} \end{pmatrix}
$$

Based on this ordering, we can formulate the distributed SpMV with two-level replication. We have three kinds of communication here: inter-group communication between nodes (p, i), (p, j), intra-group communication between the "leaders" $(p, 1)$, $(q, 1)$ of two groups p, q and global communication between nodes (p, q), (s, t):

- The inter-group communication solely consists of reductions and broadcasts used to compute the contributions to locally or globally replicated vertices V_{R_p}, V_R. The

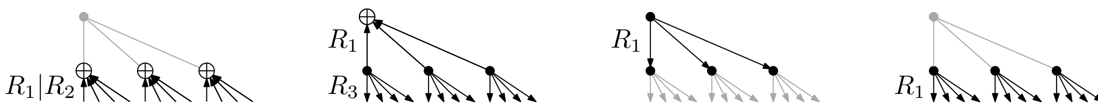


Figure 4.4.: Four phases of two-level All-Reduce in a (3,4)-way partition:
1. Reduce R_1, R_2 to local groups, 2. Reduce R_1 from local group leaders, Broadcast R_3 to local groups, 3. Broadcast R_1 to local group leaders, 4. Broadcast R_1 to local groups

contributions first get reduced and stored at the group leader and can then, after adding the contributions from other parts, be broadcast to the whole group again.

- The intra-group communication using the group leaders also consists of reductions and broadcasts to compute the contributions to globally replicated vertices V_R. Additionally, we need to compute the contribution of locally replicated vertices V_{R_q} from other parts $q \neq p$ to the locally replicated vertices V_{R_p} from part p, which can be implemented using an All-To-All communication between the group leaders. Here the choice of group leader for each group could be used to balance the communication bottleneck within a group.

- The global communication is used for two tasks. As before, we need to exchange data to compute the contributions from non-replicated vertices V_{qj} from other parts $(q, j) \neq (p, i)$ to local non-replicated vertices V_{pi} in a global All-To-All operation. Secondly, we need to collect the contributions from other non-replicated vertices $V_{qj}, q \neq p$ to locally replicated vertices V_{R_p}. There are two ways this could be implemented, depending on the communication network characteristics: Either the group leader is solely responsible for collecting these contributions and broadcasting them afterwards, or each node (p, i) in the group communicates with its counterparts (q, i) in other groups, and the contributions get accumulated using an All-Reduce operation. The first approach minimizes the amount of messages that need to be exchanged, while the second approach balances the communication better within the group. In the following description, we solely show the first approach.

Algorithm 4.3 shows the complete distributed SpMV implementation in its send-first variant. Note that the three reduction/broadcast operations R_1, R_2, R_3 can be combined into a single collective operation without additional synchronization: Figure 4.4 shows how we can execute the reductions R_1, R_2 within the group first and using the intermediate results to complete the reduction part of the All-Reduce operation R_1 and the broadcast operation R_3.

The general approach of recursively partitioning and subdividing can be repeated in multiple levels, constructing a hierarchy of replicated and local vertices. For hierarchies with many levels, it can also make sense to study the all-to-all communication further, as much of the communication volume will occur within a group. However, this is out of the scope of this work due to its complexity.

Algorithm 4.3: Distributed SpMV with two-level replication

Parallel process storing part (p, q) with its local vectors x_{pq}, y_{pq}, second-level replicated vectors x_{R_p}, y_{R_p}, first-level replicated vectors x_R, y_R and corresponding blocks $A_{RR}, A_{RR_p}, A_{Rpq}, A_{R_p\bullet}, A_{pq\bullet}$ of the whole matrix A. We only show the send-first implementation here.

for $(s, t) = (1, 1), \ldots, (k_1, k_2),\ (s, t) \neq (p, q)$ **do**
  if $t = 1$ **then** $x_{snd} \leftarrow \text{gather}(x_{pq}, N^{\leftarrow}(V_{R_s} \cup V_{st}) \cap V_{pq})$
  else $x_{snd} \leftarrow \text{gather}(x_{pq}, N^{\leftarrow}(V_{st}) \cap V_{pq})$
  Send x_{snd} to Part (s, t)
  Receive x_{rcv}^{st} from Part (s, t)
end
$\hat{y}_R^1 \leftarrow \text{allreduce}_{(s,t)=(1,1)}^{(k_1,k_2)}(A_{Rst} \cdot x_{st})$ (R_1)
if $q = 1$ **then**
  $\hat{y}_R^2 \leftarrow \text{allreduce}_{s=1}^{k_1}(A_{RR_s} \cdot x_{R_s})$ (R_2)
  $\hat{y}_{R_p} \leftarrow A_{R_s\bullet} \cdot (x_{rcv}^{11} | \cdots | x_{rcv}^{k_1 k_2})^T$
  Broadcast y_R, y_{R_p} to $(p, 1), \ldots, (p, k_2)$ (R_3)
end
$y_R \leftarrow A_{RR} \cdot x_R + \hat{y}_R^1 + \hat{y}_R^2$
$y_{R_p} \leftarrow A_{R_pR} \cdot x_R + A_{R_pR_p} \cdot x_{R_p} + \hat{y}_{R_p}$
$y_{pq} \leftarrow A_{pq\bullet} \cdot (x_R | x_{R_p} | x_{pq} | x_{rcv}^{11} | \cdots | x_{rcv}^{k_1 k_2})^T$

5. Evaluation

5.1. Experimental Setup

We evaluated our theoretical ideas and approaches both based on communication models as well as in experiments on a HPC cluster.

5.1.1. Implementation

We implemented a distributed sparse matrix-vector and sparse matrix-dense matrix algorithm. For graph and hypergraph partitioning, we used KaHIP [30] with its `fastsocial` configuration using clustering-based coarsening and PaToH [33] with its `SPEED` configuration, both with deg +1 vertex weights and an imbalance limit of 3%.[1] For our largest input graphs, we were only able to use KaHIP, as PaToH terminated unexpectedly. All codes were compiled using GCC 7.3.0 and the flags `-O3 -march=native`. We used the Intel Math Kernel Library (MKL) version 2019.0.0 in double precision with its OpenMP parallelization and 32bit indices for all low-level linear algebra kernels (spmv, spmm, axpy, dot, ...) except for the `gather` and `scatter` kernels. These kernels are used to collect or distribute the data to and from the all-to-all operation, and were implemented by hand using OpenMP for parallelization. We used OpenMPI 3.1 as the basis for our distributed implementation.

5.1.2. Environment

Our distributed benchmarks were executed on the ForHLR II cluster, which has 1152 nodes, each with two deca-core Intel Xeon E5-2660 v3 processors (Haswell) with a regular clock frequency of 2,6 GHz (maximum 3,3 GHz) and 64 GB main memory. The nodes are connected by an InfiniBand 4X EDR Interconnect with a theoretical bandwidth of 54.54 GBits/s per node and 100 GBits/s total.

5.1.3. Data Sets

We used a variety of real-world networks and generated graphs based on the Barabási-Albert and Random Hyperbolic graph model as input graphs for our experiments. As a sanity check, we also added a Delaunay triangulation as an example of a graph that has a more regular structure, and should thus not lead to any advantages by data replication. Table 5.1 lists all of these graphs with their basic properties and degree distribution.

[1] We also evaluated hMETIS and KaHyPar as alternative hypergraph partitioners, but the former proved difficult to integrate due to its binary-only distribution and the latter failed to produce a partition within an acceptable time-frame even for medium-sized hypergraphs

Name	Vertices	Edges	Type	Ref.	Degrees
Real-world networks					
cnr-2000	326k	5.48M	Web graph	[52][53]	
web-Google	357k	4.19M	Web graph	[54]	
coPapersCiteseer	434k	32.1M	Co-authorship	[55]	
coPapersDBLP	540k	30.5M	Co-authorship	[55]	
as-skitter	555k	11.6M	Internet topology	[54]	
amazon-2008	735k	70.5M	Book similarity	[52][53]	
eu-2005	863k	32.3M	Web graph	[52][53]	
in-2004	1.38M	27.2M	Web graph	[52][53]	
uk-2002	18.5M	524M	Web graph	[52][53]	
arabic-2005	22.7M	1.1G	Web graph	[52][53]	
Generated graphs					
ba-1M-10	1M	20M	Barabási-Albert	[44]	
RHG-1-10	1M	20.1M	Random Hyperbolic	[46]	
RHG-1-20	1M	40M	Random Hyperbolic	[46]	
RHG-10-100	10M	200M	Random Hyperbolic	[46]	
RHG-10-200	10M	399M	Random Hyperbolic	[46]	
Non-network graphs					
del-24	16.8M	101M	Delaunay triangulation	[56]	

Table 5.1.: Input graphs with vertex and (directed) edge count, type classification, source and a log-log plot of degree (blue) and cut-degree distribution for partition with $k = 16, 64, 256$ parts (red, yellow, green) like in Figure 4.2

We evaluated our implementation in the theoretical communication model using partitions of the input graphs into $k = 16, 64$ and 256 parts with an imbalance limit of 3% and 4 different random seeds. In the practical implementation, we used only 16-way and 64-way partitions with the same parameters. Each parallel task uses all 20 cores of a single distributed node for the shared-memory linear algebra kernels. This way, we only measure latencies and bandwidths of the interconnection network, no MPI data transfers via shared memory.

We also measured the runtime difference between the compute-first and send-first implementation using the CSC and CSR format without replication. For replicated vertices, we only used the compute-first implementation with CSR.

5.1.4. Experiments in the Communication Model

For each input graph, we computed multiple partitions with deg +1 weights. We then used our replication heuristics `degree`, `cutdegree`, `connectivity` and `greedy` to determine the 1% most “important” vertices (at most 10000 vertices). Based on these replication orders, we evaluated the communication volume (4.1) when replicating the 0 up to 10000 most important vertices. For the resulting plots, we assume that the bandwidth of All-To-All communication is equal to the All-Reduce bandwidth ($\alpha = 1$).

5.1.5. Experiments on the HPC Cluster

We measured the runtime of a single distributed SpMV computation for every input graph in different configurations. To be able to differentiate between communication latencies and bandwidth effects, we computed the SpMV product with multiple right-hand sides:

$$(y_1|\cdots|y_r) = A \cdot (x_1|\cdots|x_r)$$

While for small r, the setup overhead of the All-To-All and All-Reduce operations plays a larger role, for larger r the communication bandwidth becomes the limiting factor. We executed the SpMV computation with $r = 1, 16, 256$ for graphs with less than $10M$ vertices and with $r = 1, 8, 32$ for the largest graphs. This limit is necessary as we additionally measured the runtime of the same (shared-memory) SpMV computation on a single distributed node as a reference solution, which limits our memory usage. To check our computations for validity, we compare the result from the reference SpMV with the distributed result.

We executed each operation within the distributed SpMV implementation 10 times, and recorded the average runtime on each distributed node. This allows us to both measure the total (bottleneck) execution time as well as the computation time balance.

Finally, to evaluate the performance impact of replication in the application context on a number of graphs which seem to favor replication in terms of runtime. We implemented the PageRank algorithm using a simple power iteration as in Algorithm 2.4 as well as the GMRES algorithm to compute the solution to a linear system based on the graph Laplacian

5.2. Experimental Results

5.2.1. Communication Model

The replication heuristics evaluated in the communication model show very promising results, as listed in Table 5.2. After replicating only a small number of vertices (rarely more than 1000), the communication volume is drastically reduced, sometimes by up to 90%. We could observe the best results with either the `cutdegree` or the `greedy` heuristic. Despite it being closest to the actual communication volume, the `connectivity` heuristic resulted in the smallest reduction on almost all graphs. As expected, replication worked best on network graphs, while we could only observe a small effect on `del-24`. The Barabási-Albert graph `ba-10M-1` proved to be very difficult to partition, which is why we excluded it from the runtime measurements. In the random hyperbolic graphs, we noticed a large drop in communication volume when replicating only the most important 100-200 vertices, but no reductions afterwards. This can be explained by the geometric structure of the graphs, where the removal of a few vertices from the "center" of the graph causes it to become almost disconnected. This is also reflected in the communication quotient graph of the partition. Figure 5.1 shows the effect of replication on the SpMV communication for three input graphs, where the disconnection is most visible for `RHG-10-100`. A second interesting observation is the following: The more parts our partition has, the fewer vertices make up the optimal replication count for almost all graphs. This is likely due to the fact that the quotient graph of the partition is much more sparse for a larger k, so replicating a vertex has a smaller impact on the overall communication volume.

We did not observe any significant differences between the partitions generated by KaHIP and PaToH, or by repartitioning the graphs after replication, which is why we only report results for KaHIP in this thesis. This result is again surprising, since KaHIP optimizes the "wrong" objective, while PaToH optimizes for the correct hypergraph partitioning objective, but consistent with observations from larger hypergraph partitioning surveys that found the largest discrepancies between graph and hypergraph partitioning in non-symmetric graphs [57], while we only used symmetric input matrices/undirected graphs.

5.2.2. Practical Measurements

The practical runtimes of the distributed SpMV implementations are very mixed compared to their communication model, while also strongly dependent on the number of right-hand sides we computed at the same time. As Table 5.3 shows, for $r = 1$, we only find small differences in the SpMV runtime with replication compared to unreplicated SpMV, even for the best replication heuristics. For some heuristics and graphs, the runtime was even significantly higher, like `degree` on `arabic-2005` or `eu-2005`. However, the generated `RHG` graphs and even `del-24` showed surprisingly large reductions in runtime up to 50%. Moving to a larger number of right-hand sides, the effects of replication becomes stronger and more consistent: On almost all network graphs, we observe a reduction in runtime by at least 20% by using the `degree` or `cutdegree` replication heuristic, on the largest instance

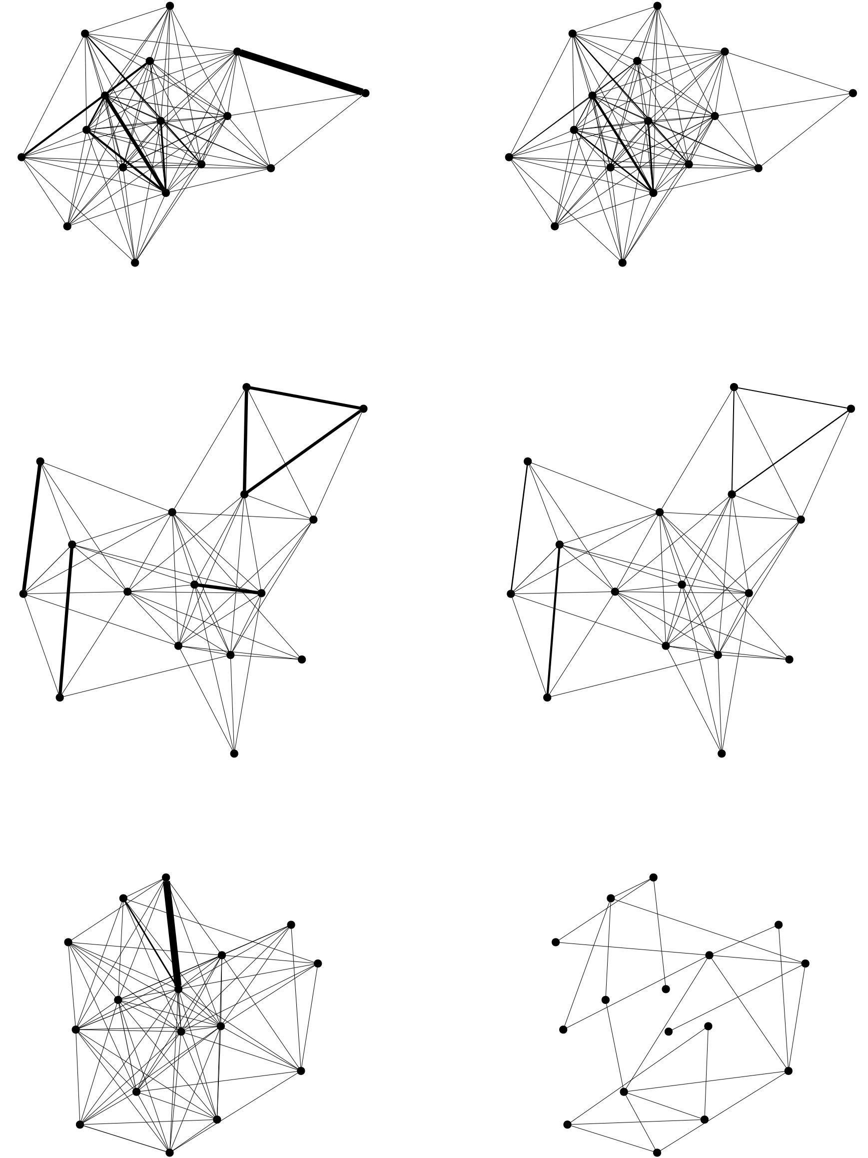


Figure 5.1.: Communication graphs for unreplicated (left) and replicated (right) SpMV with $k = 16$ nodes and input graphs `in-2004` (top), `cnr-2000` (middle) and `RHG-10-100` (bottom). The points represent distributed nodes, the edge thickness is proportional to the communication between the two incident parts.

Table 5.2.: Minimal communication volume when replicating with different heuristics for $k = 64$ with an $1 : 1$ relationship between p2p-volume and All-Reduce volume

graph	no replication	degree	cutdegree	connectivity	greedy
cnr-2000	22.4k	3.16k	**2.14k**	18.9k	14.5k
web-Google	2.46k	1.66k	**1.43k**	1.56k	1.92k
coPapersCiteseer	14.1k	15.1k	13.6k	13.9k	**11.3k**
coPapersDBLP	27.1k	27.2k	26.1k	26.5k	**22.8k**
as-skitter	35.5k	16.4k	**13.9k**	17.4k	19.0k
amazon-2008	17.4k	17.6k	17.0k	17.2k	**14.6k**
eu-2005	40.2k	27.0k	**21.9k**	28.0k	37.2k
in-2004	16.8k	8.70k	**8.43k**	13.5k	14.5k
uk-2002	87.9k	75.7k	**47.9k**	75.3k	78.2k
arabic-2005	346k	144k	**117k**	187k	306k
ba-1M-10	221k	201k	201k	202k	**196k**
RHG-1-10	15.7k	548	**431**	570	15.2k
RHG-1-20	3.44k	890	**692**	896	3.11k
RHG-10-100	5.70k	401	**313**	430	5.50k
RHG-10-200	8.95k	1.43k	**885**	1.44k	8.65k
del-24	2.53k	2.63k	2.61k	2.62k	**2.52k**

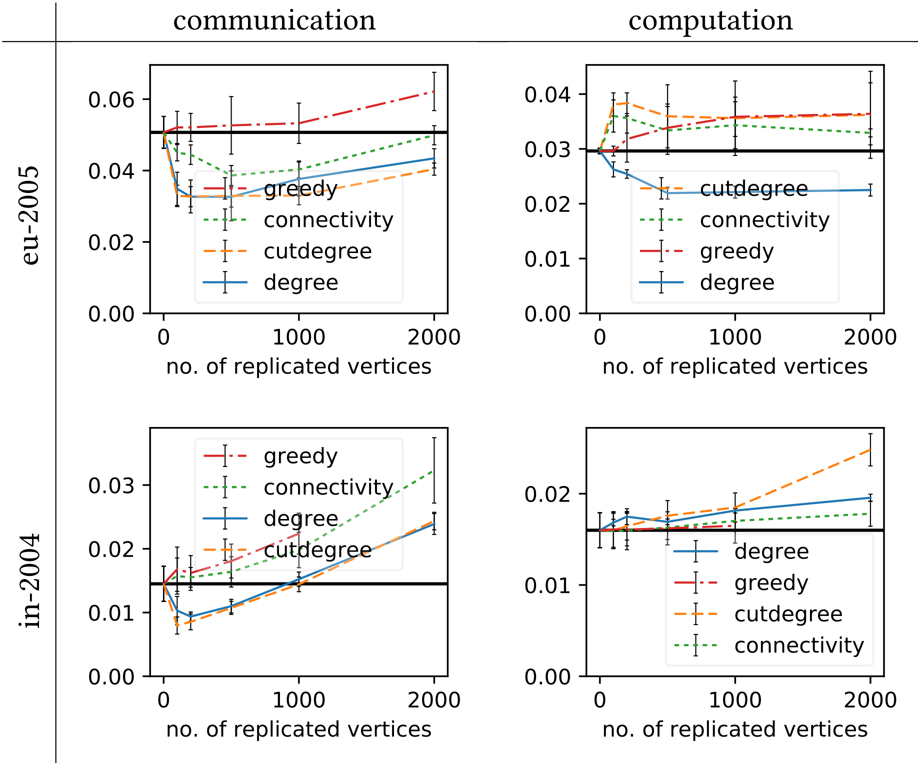


Figure 5.2.: Runtime breakdown for `eu-2005` and `in-2004` when computing SpMV with $r = 256$ right-hand sides

graph	no repl.	degree	cutdegree	connectivity	greedy
			$r = 1$		
cnr-2000	0.52(± 0.16)	0.44(± 0.11)	**0.33**(± 0.01)	0.52(± 0.09)	0.47(± 0.02)
web-Google	0.56(± 0.45)	0.42(± 0.15)	0.44(± 0.15)	**0.34**(± 0.00)	0.35(± 0.01)
coPapersCiteseer	0.81(± 0.33)	0.85(± 0.30)	**0.55**(± 0.04)	0.86(± 0.39)	0.58(± 0.02)
coPapersDBLP	**0.66**(± 0.03)	1.08(± 0.70)	0.71(± 0.07)	1.04(± 0.19)	0.70(± 0.02)
as-skitter	0.89(± 0.28)	0.93(± 0.40)	**0.63**(± 0.07)	0.94(± 0.42)	0.76(± 0.09)
amazon-2008	0.78(± 0.42)	0.73(± 0.41)	0.63(± 0.27)	0.79(± 0.36)	**0.52**(± 0.02)
eu-2005	0.85(± 0.04)	3.23(± 1.79)	**0.70**(± 0.01)	0.94(± 0.20)	0.84(± 0.03)
in-2004	0.89(± 0.38)	**0.67**(± 0.05)	0.77(± 0.14)	0.74(± 0.14)	0.78(± 0.21)
uk-2002	4.72(± 0.19)	4.86(± 0.17)	**4.64**(± 0.18)	4.78(± 0.11)	4.79(± 0.17)
arabic-2005	15.88(± 1.97)	26.09(± 2.22)	**15.12**(± 2.54)	16.03(± 2.33)	15.74(± 2.14)
RHG-1-10	0.71(± 0.66)	0.56(± 0.33)	0.43(± 0.19)	**0.35**(± 0.01)	0.63(± 0.23)
RHG-1-20	**0.34**(± 0.01)	1.19(± 0.90)	0.38(± 0.03)	0.59(± 0.23)	0.49(± 0.17)
RHG-10-100	2.72(± 1.84)	2.23(± 1.54)	**1.25**(± 0.14)	1.43(± 0.19)	1.26(± 0.29)
RHG-10-200	2.25(± 0.17)	2.34(± 0.40)	**2.02**(± 0.03)	2.07(± 0.07)	2.04(± 0.02)
del-24	1.66(± 1.76)	3.51(± 0.49)	1.12(± 0.31)	1.73(± 0.98)	**0.81**(± 0.28)
			$r = 256$		
cnr-2000	21.62(± 2.45)	8.35(± 0.52)	**6.96**(± 0.29)	22.22(± 1.32)	22.13(± 1.50)
web-Google	**3.77**(± 0.17)	4.54(± 0.16)	4.47(± 0.17)	4.41(± 0.08)	4.74(± 0.21)
coPapersCiteseer	**15.76**(± 0.49)	16.76(± 0.39)	16.69(± 0.46)	16.51(± 0.46)	16.52(± 0.53)
coPapersDBLP	**26.12**(± 0.76)	32.11(± 5.20)	27.04(± 1.00)	31.22(± 4.00)	27.30(± 0.21)
as-skitter	36.33(± 4.14)	22.57(± 0.72)	**22.42**(± 0.63)	35.66(± 2.00)	36.08(± 2.30)
amazon-2008	20.97(± 2.93)	22.40(± 2.90)	20.86(± 2.21)	21.78(± 3.58)	**20.65**(± 1.21)
eu-2005	75.80(± 5.56)	**45.56**(± 2.44)	57.60(± 4.25)	70.21(± 6.41)	76.61(± 5.98)
in-2004	24.42(± 3.38)	23.37(± 0.50)	**20.75**(± 2.58)	26.10(± 4.11)	25.80(± 3.38)
RHG-1-10	26.82(± 1.37)	10.45(± 1.15)	**10.22**(± 0.90)	10.79(± 1.17)	28.15(± 1.01)
RHG-1-20	11.54(± 0.64)	**9.75**(± 0.58)	9.76(± 0.51)	10.11(± 0.54)	12.59(± 0.56)
			$r = 32$		
uk-2002	22.84(± 1.21)	22.66(± 0.83)	**21.14**(± 0.37)	23.50(± 1.27)	22.91(± 1.18)
arabic-2005	76.26(± 6.29)	**41.08**(± 1.64)	47.63(± 2.84)	55.97(± 3.89)	69.69(± 5.47)
RHG-10-100	**6.04**(± 0.09)	6.20(± 0.15)	6.41(± 0.12)	6.04(± 0.21)	6.72(± 0.05)
RHG-10-200	7.50(± 0.77)	7.38(± 0.17)	**7.32**(± 0.19)	7.35(± 0.09)	7.81(± 0.71)
del-24	8.83(± 0.14)	8.75(± 0.38)	8.68(± 0.34)	**8.55**(± 0.07)	8.57(± 0.25)

Table 5.3.: Total bottleneck runtime (ms) for distributed SpMV on 64 nodes for different replication heuristics, input graphs and numbers of right-hand sides r. Each graph was partitioned with 4 different random seeds using KaHIP. We only list the minimum runtime with its standard deviation among all replication counts.

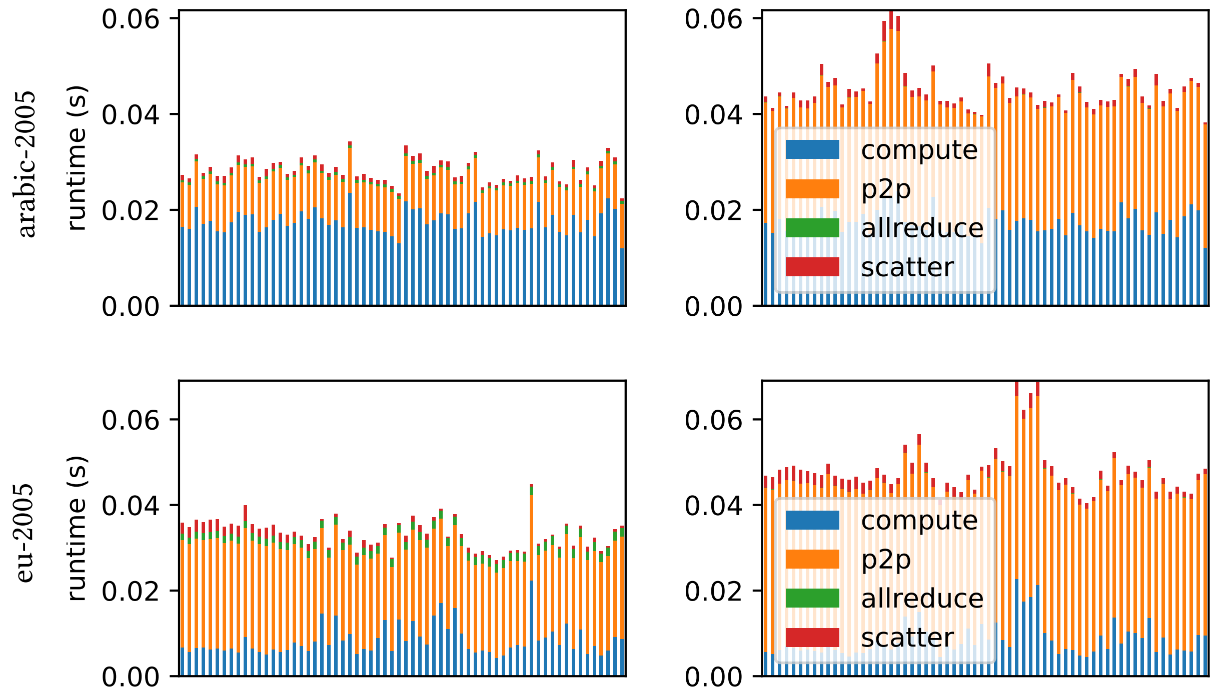


Figure 5.3.: Runtime breakdown by node for SpMV computation with $r = 32$ (`arabic-2005`) or $r = 256$ (`eu-2005`) right-hand sides and $k = 64$ nodes for an unreplicated and replicated partition using the `cutdegree` heuristic with 200 replicated vertices.

`arabic-2005` the runtime was even almost halved. On the non-network graph `del-24`, the effect is substantially smaller than for $r = 1$, which is consistent with our expectations.

Figure 5.2 shows how the choice of replication heuristic can impact both computation and communication at the same time: While we see no large effect for `in-2004`, `eu-2005` shows a substantial reduction in both computation and communication runtime for the `degree` replication heuristic.

Finally, Figure 5.3 breaks down the overall computation and communication distribution over all 64 nodes of a distributed SpMV with and without replication. We can see that the runtime for p2p communication gets reduced drastically, while the overall runtime of computation, All-Reduce and scatter (after the p2p communication) only changes slightly. We especially have a large impact on a few bottleneck nodes, which is exactly what we wanted to achieve with replication. We see the same behavior for both large and small input graphs.

5.2.3. PageRank and GMRES

The benchmarks for $r = 1$ already do not predict a large margin of improvement by data replication, but nevertheless we could measure a small effect in a few matrices. For example on `in-2004`, 10000 iterations of the PageRank algorithm took 17.5 s with `cutdegree` replication and 19.5 s without replication, both on $k = 16$ nodes. On the larger

`RHG-10-200` graph however, the same computation took 68.1 s with replication and 58.3 s without. Figure 5.4 shows us the convergence of the PageRank iteration on two graphs. The Rayleigh quotients of `in-2004` oscillate strongly around 1 before finally reaching (numerical) convergence after about 250 iterations. the Rayleigh quotients in the `RHG` graph converge on the Eigenvalue 1 much faster, but overall, convergence only seems to be reached after roughly 500 iterations. This is understandable due to the much larger size, and also might suggest that the spectrum of `in-2004` has a smaller spectral gap below the dominant Eigenvalue 1.

The attempt to use GMRES to build an inverse or Rayleigh quotient iteration were rather unsuccessful: While the residuals initially decreased quickly, they tended to stagnate for a long while, which is why we only show such a residual plot for regular GMRES and GMRES(k) in Figure 5.5. The runtime of the SpMV computations tends to diminish in importance for increasingly large Krylov spaces, as the inner products $h_i j$ and orthogonalizations therewith then tend to dominate. It should also be noted that the shifts in the Rayleigh quotient iteration as well as preconditioners like the Jacobi iteration would make no difference here, as the diagonal entries of the PageRank matrix are all identical and the shifts don't influence the Krylov spaces:

$$m > 1 : \quad K_m(A - \rho I, b) = K_m(A, b)$$

Still, due to the fast convergence and usually large enough spectral gap, the normal power iteration should be sufficient for the PageRank application. For other numerical problems on network graphs, like computations on the Laplacian matrix, other properties like the positive semi-definiteness of this matrix enable the use more stable and efficient numerical algorithms in practice.

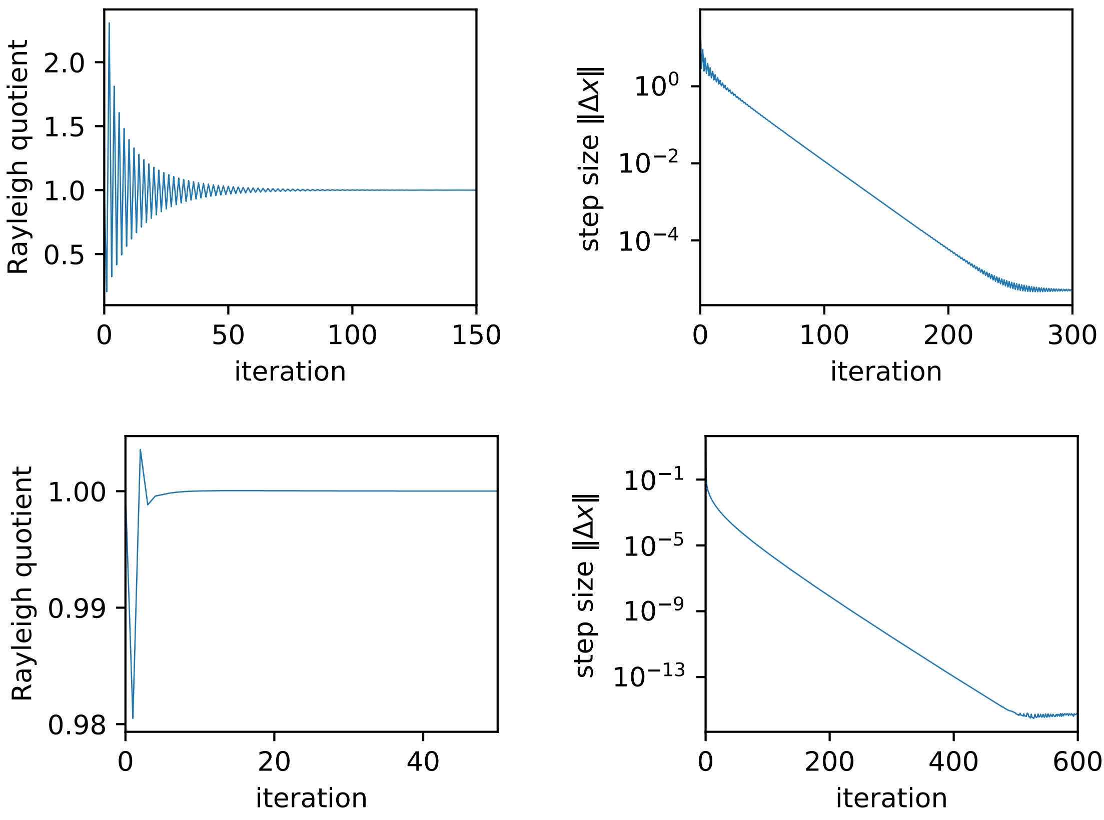


Figure 5.4.: Rayleigh quotients and Δx for the power iteration computing the PageRank vector of `in-2004` (top) and `RHG-10-200` (bottom) on $k = 16$ nodes with replication.

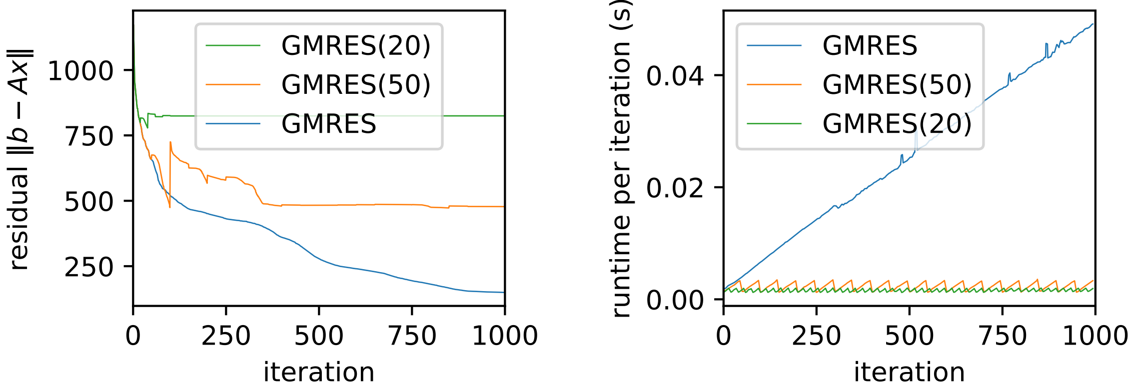


Figure 5.5.: Stagnation of GMRES(k) residuals as well as runtime per iteration on the PageRank matrix of `in-2004`, computed on 16 nodes with replication.

6. Conclusion

The experimental results paint a very mixed picture of the effectiveness of data replication for communication optimization. We were able to show in the communication model that replication might lead to a large reduction in p2p communication without much overhead from the All-Reduce operation, especially for the `cutdegree` heuristic. However, these results are only partly reflected in the runtime results, where other heuristics like the `degree` heuristic proved surprisingly effective. This was also due to the fact that we did not optimize for computation balance, which the `degree` heuristic seems to have a larger impact on. An interesting question would be which structural properties of a network graph make it a good candidate for vertex replication. Based on what we know about the input graphs and their sources, it is difficult to make an educated guess – the graphs on which we observed significant speed-ups (`cnr-2002`, `as-skitter`, `eu-2005`, `arabic-2005`) are difficult to separate from the other graphs like `in-2004`, where the theoretical model predicted a large reduction in communication volume, but in practice only a small corresponding reduction in runtime could be observed. For these instances, a closer investigation of the communication structure might prove fruitful. Finally, the PageRank implementation shows a small speedup in a few graphs, but due to the small per-vertex element size, these results are far from conclusive, especially due to the dependence of the runtime results on many variables, especially the random seeds used during partitioning.

The topic of replication as a communication optimization technique provides many possible extensions. It could be possible to include vertex replication as an optional step in local search algorithms of (hyper-)graph partitioning frameworks. During exploratory benchmarks, we could only find negligible benefits of repartitioning a graph after a small set of replicated vertices had been removed, but the integration of replication into the partitioning pipeline might provide additional flexibility: It could be used both to deal with vertices of large weight that make it difficult to satisfy imbalance constraints as well as for vertices with a large connectivity. The recursive bisection approach often employed in these frameworks would potentially work well with a true multilevel replication approach as sketched in Section 4.4. Orthogonal to the integration with partitioning algorithms, the communication model and replication heuristics could be extended to include latencies due to message count as well as contentions in the interconnection network of distributed nodes, similar to the work by Acer et al. [17]. On the theoretical side, it would be interesting to study the hyperbolic embedding of network graphs mentioned by Krioukov et al. [45], where it might be possible to generally model the effect of replication on network graphs. Finally, the results from this thesis were solely based on the distributed SpMV as a model problem. Despite its prototypical communication structure, it might be interesting how other, similar graph algorithms fare using replication. It might be especially interesting to extend vertex-centric graph frameworks to transparently incorporate replication.

A. Bibliography

[1] E. Cuthill and J. McKee. “Reducing the Bandwidth of Sparse Symmetric Matrices”. In: *Proceedings of the 1969 24th National Conference*. ACM ’69. 1969, pp. 157–172. DOI: 10.1145/800195.805928.

[2] Ajit Agrawal, Philip Klein, and R. Ravi. “Cutting down on Fill Using Nested Dissection: Provably Good Elimination Orderings”. In: *Graph Theory and Sparse Matrix Computation*. Ed. by Alan George, John R. Gilbert, and Joseph W. H. Liu. Red. by Avner Friedman and Willard Miller. Vol. 56. 1993, pp. 31–55. DOI: 10.1007/978-1-4613-8369-7_2.

[3] John Mellor-Crummey, David Whalley, and Ken Kennedy. “Improving Memory Hierarchy Performance for Irregular Applications Using Data and Computation Reorderings”. In: *International Journal of Parallel Programming* 29.3 (2001), pp. 217–247. DOI: 10.1023/A:1011119519789.

[4] Laxman Dhulipala et al. “Compressing Graphs and Indexes with Recursive Graph Bisection”. In: *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’16. 2016, pp. 1535–1544. DOI: 10.1145/2939672.2939862.

[5] Y. Saad and M. Schultz. “GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems”. In: *SIAM Journal on Scientific and Statistical Computing* 7.3 (1986), pp. 856–869. DOI: 10.1137/0907058.

[6] A. Greenbaum, V. Pták, and Z. Strakoš. “Any Nonincreasing Convergence Curve is Possible for GMRES”. In: *SIAM Journal on Matrix Analysis and Applications* 17.3 (1996), pp. 465–469. DOI: 10.1137/S0895479894275030.

[7] Jörg Liesen and Petr Tichý. “The Worst-Case GMRES for Normal Matrices”. en. In: *BIT Numerical Mathematics* 44.1 (2004), pp. 79–98. DOI: 10.1023/B:BITN.0000025083.59864.bd.

[8] R. V. Mises and H. Pollaczek-Geiringer. “Praktische Verfahren der Gleichungsauflösung .” en. In: *ZAMM - Journal of Applied Mathematics and Mechanics / Zeitschrift für Angewandte Mathematik und Mechanik* 9.2 (1929), pp. 152–164. DOI: 10.1002/zamm.19290090206.

[9] A. Knyazev. “Toward the Optimal Preconditioned Eigensolver: Locally Optimal Block Preconditioned Conjugate Gradient Method”. In: *SIAM Journal on Scientific Computing* 23.2 (Jan. 1, 2001), pp. 517–541. DOI: 10.1137/S1064827500366124.

[10] J. Duersch et al. “A Robust and Efficient Implementation of LOBPCG”. In: *SIAM Journal on Scientific Computing* 40.5 (Jan. 1, 2018), pp. C655–C676. DOI: 10.1137/17M1129830.

[11] Grzegorz Malewicz et al. “Pregel: A System for Large-scale Graph Processing”. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’10. 2010, pp. 135–146. DOI: 10.1145/1807167.1807184.

[12] Yucheng Low et al. “Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud”. In: *Proc. VLDB Endow.* 5.8 (2012), pp. 716–727. DOI: 10.14778/2212351.2212354.

[13] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. “Near linear time algorithm to detect community structures in large-scale networks”. In: *Physical Review E* 76.3 (2007), p. 036106. DOI: 10.1103/PhysRevE.76.036106.

[14] Jie Chen and Ilya Safro. “Algebraic Distance on Graphs”. en. In: *SIAM Journal on Scientific Computing* 33.6 (2011), pp. 3468–3490. DOI: 10.1137/090775087.

[15] Bora Uçar and Cevdet Aykanat. *A library for parallel sparse matrix-vector multiplies.* Tech. rep. Department of Computer Engineering, Bilkent University, 2005. DOI: 10.1137/16M1105591.

[16] Ü. Çatalyürek and C. Aykanat. “Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication”. In: *IEEE Transactions on Parallel and Distributed Systems* 10.7 (1999), pp. 673–693. DOI: 10.1109/71.780863.

[17] Seher Acer, Oguz Selvitopi, and Cevdet Aykanat. “Addressing Volume and Latency Overheads in 1D-parallel Sparse Matrix-Vector Multiplication”. en. In: *Euro-Par 2017: Parallel Processing*. Ed. by Francisco F. Rivera, Tomás F. Pena, and José C. Cabaleiro. Lecture Notes in Computer Science. 2017, pp. 625–637.

[18] Ü. Çatalyürek, C. Aykanat, and B. Uçar. “On Two-Dimensional Sparse Matrix Partitioning: Models, Methods, and a Recipe”. In: *SIAM Journal on Scientific Computing* 32.2 (2010), pp. 656–683. DOI: 10.1137/080737770.

[19] Joseph E. Gonzalez et al. “PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs”. In: Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12). 2012, pp. 17–30.

[20] Sebastian Schlag et al. “Scalable Edge Partitioning”. In: ed. by Stephen Kobourov and Henning Meyerhenke. Jan. 2019. DOI: 10.1137/1.9781611975499.

[21] Miroslav Fiedler. “A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory”. In: *Czechoslovak Mathematical Journal* 25.4 (1975), pp. 619–633.

[22] Dorothy Kucar, Shawki Areibi, and Anthony Vannelli. “Hypergraph Partitioning Techniques”. In: *Dynamics of Continuous, Discrete and Impulsive Systems* 11 (2004), p. 29.

[23] B. W. Kernighan and S. Lin. “An efficient heuristic procedure for partitioning graphs”. In: *The Bell System Technical Journal* 49.2 (Feb. 1970), pp. 291–307. DOI: 10.1002/j.1538-7305.1970.tb01770.x.

[24] C. M. Fiduccia and R. M. Mattheyses. “A Linear-Time Heuristic for Improving Network Partitions”. In: *19th Design Automation Conference*. 19th Design Automation Conference. June 1982, pp. 175–181. DOI: 10.1109/DAC.1982.1585498.

[25] Peter Sanders and Christian Schulz. "Engineering Multilevel Graph Partitioning Algorithms". In: *Algorithms – ESA 2011*. Ed. by Camil Demetrescu and Magnús M. Halldórsson. Lecture Notes in Computer Science. 2011, pp. 469–480.

[26] Tobias Heuer, Peter Sanders, and Sebastian Schlag. "Network Flow-Based Refinement for Multilevel Hypergraph Partitioning". In: *17th International Symposium on Experimental Algorithms (SEA 2018)*. 2018, 1:1–1:19.

[27] George Karypis and Vipin Kumar. "Multilevel Graph Partitioning Schemes". In: *Proc. 24th Intern. Conf. Par. Proc., III*. 1995, pp. 113–122.

[28] H. Meyerhenke, P. Sanders, and C. Schulz. "Parallel Graph Partitioning for Complex Networks". In: *IEEE Transactions on Parallel and Distributed Systems* 28.9 (Sept. 2017), pp. 2625–2638. DOI: 10.1109/TPDS.2017.2671868.

[29] Tobias Heuer and Sebastian Schlag. "Improving Coarsening Schemes for Hypergraph Partitioning by Exploiting Community Structure". In: *16th International Symposium on Experimental Algorithms, (SEA 2017)*. 2017, 21:1–21:19.

[30] Peter Sanders and Christian Schulz. "Think Locally, Act Globally: Highly Balanced Graph Partitioning". In: *Experimental Algorithms*. Ed. by Vincenzo Bonifaci, Camil Demetrescu, and Alberto Marchetti-Spaccamela. Lecture Notes in Computer Science. 2013, pp. 164–175.

[31] George Karypis and Vipin Kumar. "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs". In: *SIAM J. Sci. Comput.* 20.1 (Dec. 1998), pp. 359–392. DOI: 10.1137/S1064827595287997.

[32] Sebastian Schlag et al. "k-way Hypergraph Partitioning via *n*-Level Recursive Bisection". In: *18th Workshop on Algorithm Engineering and Experiments, (ALENEX 2016)*. 2016, pp. 53–67.

[33] Ümit Çatalyürek and Cevdet Aykanat. "PaToH (Partitioning Tool for Hypergraphs)". In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. 2011, pp. 1479–1487. DOI: 10.1007/978-0-387-09766-4_93.

[34] George Karypis and Vipin Kumar. "Multilevel K-way Hypergraph Partitioning". In: *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*. DAC '99. 1999, pp. 343–348. DOI: 10.1145/309847.309954.

[35] Karen D. Devine et al. "Parallel Hypergraph Partitioning for Scientific Computing". In: 2006.

[36] A. Clauset, C. Shalizi, and M. Newman. "Power-Law Distributions in Empirical Data". In: *SIAM Review* 51.4 (2009), pp. 661–703. DOI: 10.1137/070710111.

[37] Anna D. Broido and Aaron Clauset. "Scale-free networks are rare". In: *Nature Communications* 10.1 (Mar. 4, 2019), pp. 1–10. DOI: 10.1038/s41467-019-08746-5.

[38] Duncan J. Watts and Steven H. Strogatz. "Collective dynamics of 'small-world' networks". In: *Nature* 393.6684 (June 1998), pp. 440–442. DOI: 10.1038/30918.

[39] Reza Bakhshandeh et al. "Degrees of Separation in Social Networks". In: *Fourth Annual Symposium on Combinatorial Search*. Fourth Annual Symposium on Combinatorial Search. July 5, 2011.

[40] Reuven Cohen and Shlomo Havlin. *Complex Networks: Structure, Robustness and Function.* 2010. DOI: 10.1017/CBO9780511780356.

[41] Paul W. Holland and Samuel Leinhardt. "Transitivity in Structural Models of Small Groups". In: *Comparative Group Studies* 2.2 (May 1, 1971), pp. 107–124. DOI: 10.1177/104649647100200201.

[42] Albert-László Barabási and Réka Albert. "Emergence of Scaling in Random Networks". en. In: *Science* 286.5439 (1999), pp. 509–512. DOI: 10.1126/science.286.5439.509.

[43] Konstantin Klemm and Víctor M. Eguíluz. "Growing scale-free networks with small-world behavior". In: *Physical Review E* 65.5 (May 8, 2002), p. 057102. DOI: 10.1103/PhysRevE.65.057102.

[44] Vladimir Batagelj and Ulrik Brandes. "Efficient generation of large random networks". en. In: *Physical Review E* 71.3 (2005). DOI: 10.1103/PhysRevE.71.036113.

[45] Dmitri Krioukov et al. "Hyperbolic geometry of complex networks". In: *Physical Review E* 82.3 (2010), p. 036106. DOI: 10.1103/PhysRevE.82.036106.

[46] Moritz von Looz, Henning Meyerhenke, and Roman Prutkin. "Generating Random Hyperbolic Graphs in Subquadratic Time". en. In: *Algorithms and Computation.* Ed. by Khaled Elbassioni and Kazuhisa Makino. Lecture Notes in Computer Science. 2015, pp. 467–478.

[47] Luca Gugelmann, Konstantinos Panagiotou, and Ueli Peter. "Random Hyperbolic Graphs: Degree Sequence and Clustering". In: *Automata, Languages, and Programming.* Ed. by Artur Czumaj et al. Lecture Notes in Computer Science. 2012, pp. 573–585.

[48] Alex Bavelas. "Communication Patterns in Task-Oriented Groups". In: *The Journal of the Acoustical Society of America* 22.6 (1950), pp. 725–730. DOI: 10.1121/1.1906679.

[49] Linton C. Freeman. "A Set of Measures of Centrality Based on Betweenness". In: *Sociometry* 40.1 (1977), pp. 35–41. DOI: 10.2307/3033543.

[50] Lawrence Page et al. *The PageRank citation ranking: Bringing order to the web.* Tech. rep. Stanford InfoLab, 1999.

[51] Kurt Bryan and Tanya Leise. "The $25,000,000,000 Eigenvector: The Linear Algebra behind Google". en. In: *SIAM Review* 48.3 (2006), pp. 569–581. DOI: 10.1137/050623280.

[52] P. Boldi and S. Vigna. "The webgraph framework I: compression techniques". en. In: *Proceedings of the 13th conference on World Wide Web - WWW '04.* 2004, p. 595. DOI: 10.1145/988672.988752.

[53] Paolo Boldi et al. "Layered Label Propagation: A Multiresolution Coordinate-free Ordering for Compressing Social Networks". In: *Proceedings of the 20th International Conference on World Wide Web.* WWW '11. 2011, pp. 587–596. DOI: 10.1145/1963405.1963488.

[54] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. http://snap.stanford.edu/data. 2014.

[55] Robert Geisberger, Peter Sanders, and Dominik Schultes. "Better Approximation of Betweenness Centrality". en. In: *2008 Proceedings of the Tenth Workshop on Algorithm Engineering and Experiments (ALENEX)*. Ed. by J. Ian Munro and Dorothea Wagner. 2008, pp. 90–100. DOI: 10.1137/1.9781611972887.9.

[56] M. Holtgrewe, P. Sanders, and C. Schulz. "Engineering a scalable high quality graph partitioner". In: *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. 2010, pp. 1–12. DOI: 10.1109/IPDPS.2010.5470485.

[57] Sivasankaran Rajamanickam and Erik Boman. "Parallel partitioning with Zoltan: Is hypergraph partitioning worth it?" In: *Contemporary Mathematics*. Ed. by David Bader et al. Vol. 588. Jan. 2013, pp. 37–52. DOI: 10.1090/conm/588/11711.

B. Additional Experimental Results

B.1. Theoretical Communication Model

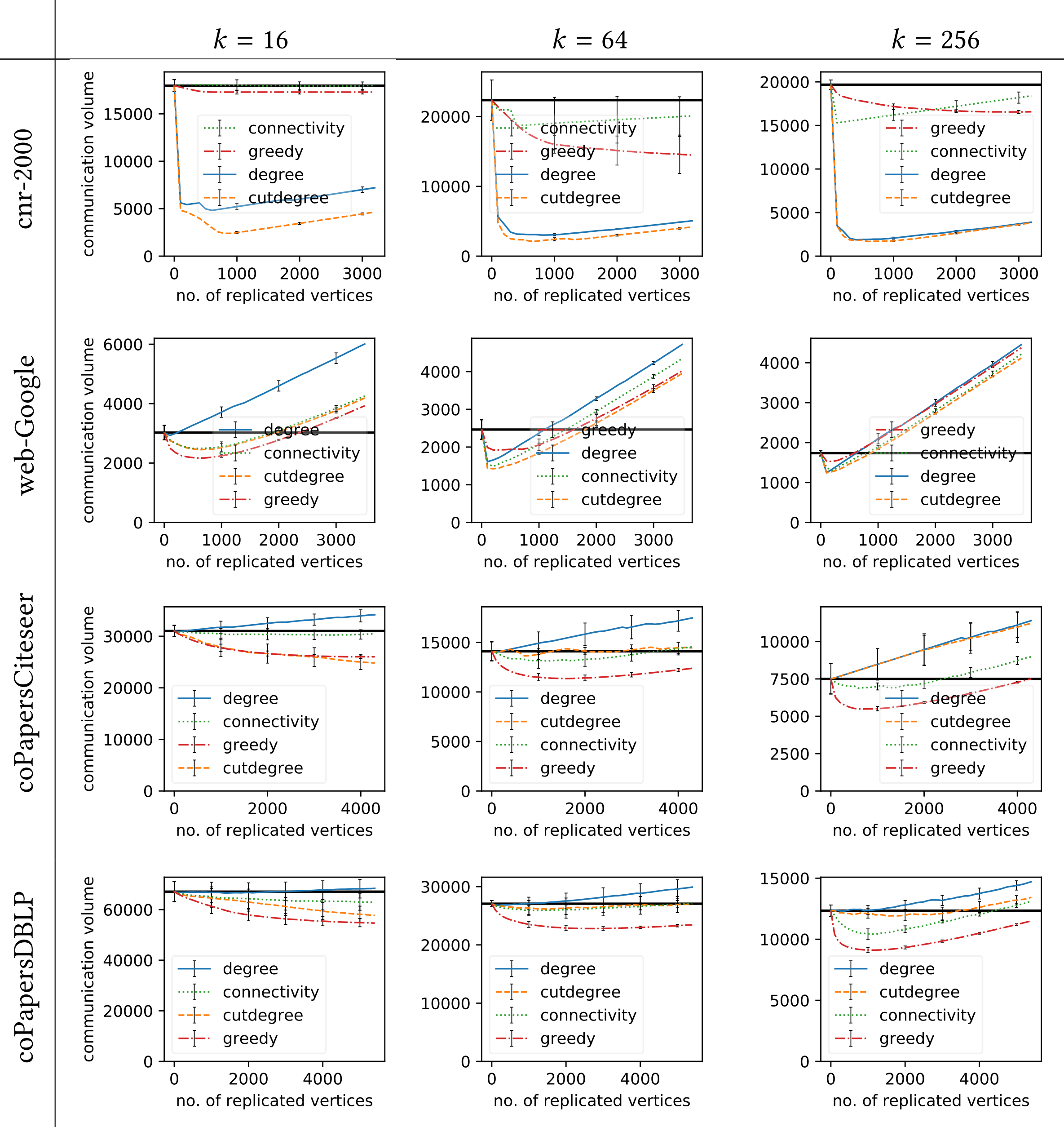
Table B.1.: Minimal communication volume when replicating with different heuristics for $k = 16, 256$. See Table 5.2 for $k = 64$

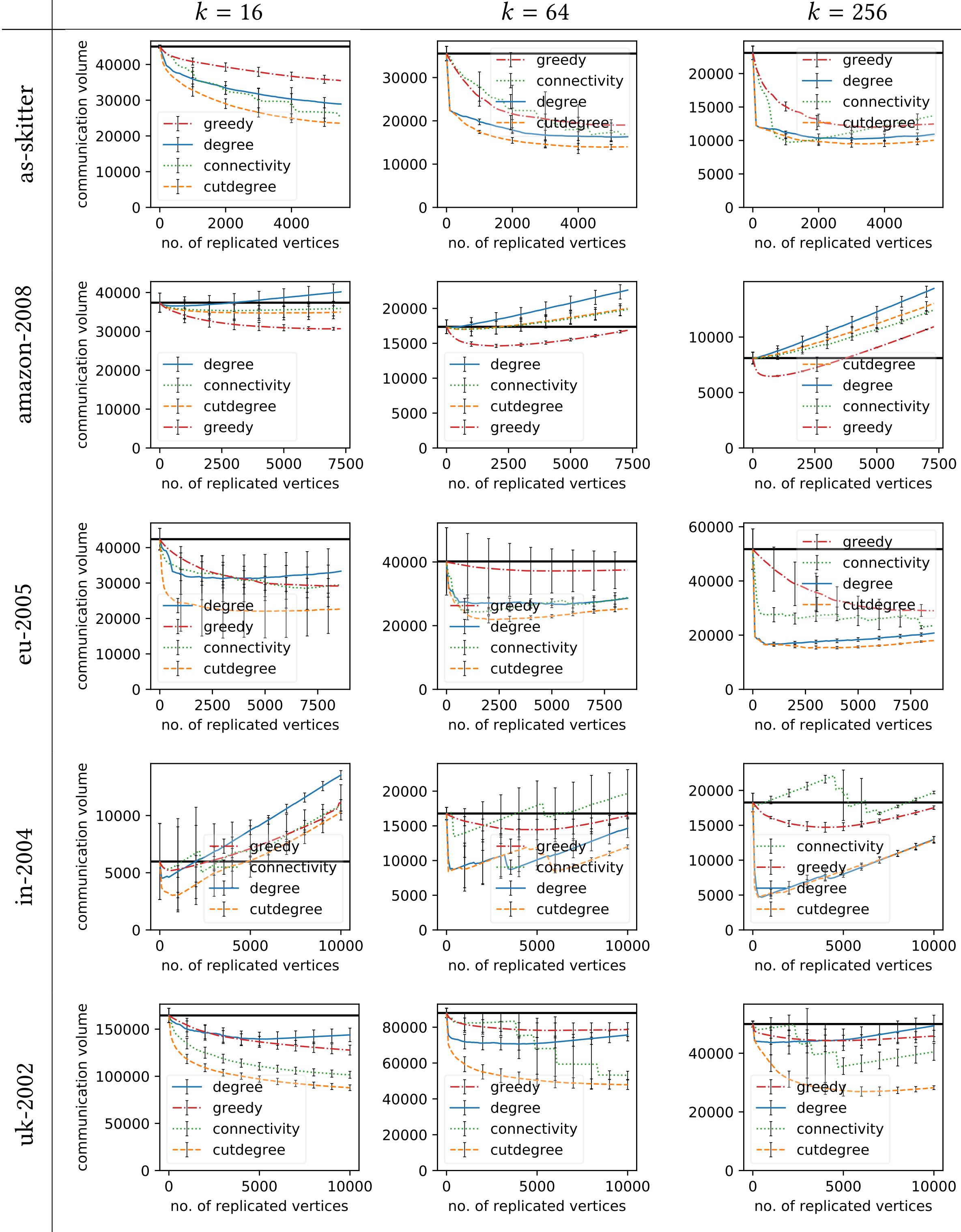
graph, $k = 16$	no replication	`degree`	`cutdegree`	`connectivity`	`greedy`
cnr-2000	18.0k	6.71k	**2.40k**	18.0k	17.3k
web-Google	3.03k	3.27k	2.45k	2.78k	**2.17k**
coPapersCiteseer	31.0k	33.4k	**24.8k**	30.9k	26.0k
coPapersDBLP	67.1k	68.4k	57.7k	64.5k	**54.7k**
as-skitter	45.0k	28.9k	**23.6k**	24.9k	35.5k
amazon-2008	37.4k	37.3k	34.7k	36.0k	**30.7k**
eu-2005	42.4k	32.4k	**22.0k**	32.5k	29.2k
in-2004	5.98k	6.22k	**3.02k**	5.67k	5.19k
uk-2002	165k	144k	**87.9k**	110k	128k
arabic-2005	304k	185k	**133k**	208k	256k
ba-1M-10	561k	518k	**518k**	552k	535k
RHG-1-10	9.45k	258	**198**	258	9.34k
RHG-1-20	1.40k	612	**395**	644	1.13k
RHG-10-100	1.40k	257	**203**	254	1.25k
RHG-10-200	2.17k	510	**429**	514	2.03k
del-24	**4.65k**	4.75k	4.72k	4.74k	4.66k

graph, $k = 256$	no replication	`degree`	`cutdegree`	`connectivity`	`greedy`
cnr-2000	19.7k	3.49k	**1.68k**	15.6k	16.5k
web-Google	1.74k	1.34k	**1.24k**	1.37k	1.53k
coPapersCiteseer	7.51k	8.08k	7.59k	7.27k	**5.49k**
coPapersDBLP	12.3k	12.4k	11.9k	11.2k	**9.11k**
as-skitter	23.1k	11.2k	**9.50k**	11.3k	12.0k
amazon-2008	8.10k	8.10k	8.11k	8.10k	**6.46k**
eu-2005	51.7k	20.2k	**15.3k**	27.4k	29.0k
in-2004	18.3k	9.88k	**4.68k**	18.1k	14.7k
uk-2002	50.0k	44.5k	**26.9k**	48.7k	44.3k
arabic-2005	332k	93.8k	**84.7k**	172k	276k
ba-1M-10	65.1k	63.4k	63.4k	63.5k	**61.5k**
RHG-1-10	4.93k	998	**517**	901	4.52k
RHG-1-20	6.26k	1.39k	**1.02k**	1.34k	5.06k

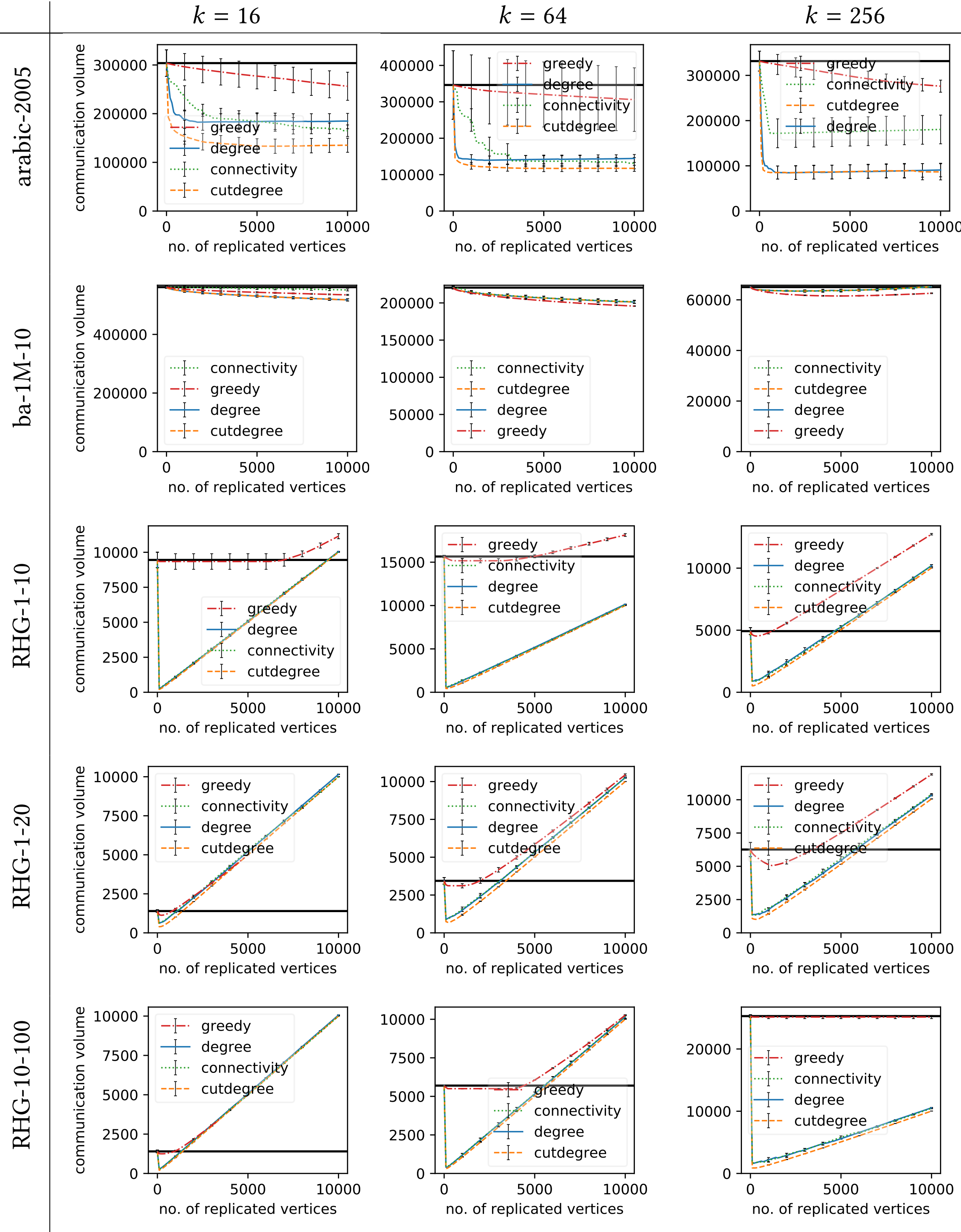
RHG-10-100	25.3k	1.62k	**880**	1.63k	25.1k
RHG-10-200	28.6k	4.98k	**1.54k**	4.30k	28.2k
del-24	**1.51k**	1.61k	1.60k	1.61k	1.51k

Table B.2.: Theoretical communication volume (4.1) for different replication heuristics and counts based on a 1 : 1 relationship between All-To-All and All-Reduce bandwidth for all input graphs. Each graph was partitioned into $k = 16, 64, 256$ parts using KaHIP with 4 different random seeds. The plots show the average bottleneck communication volume and its standard deviation

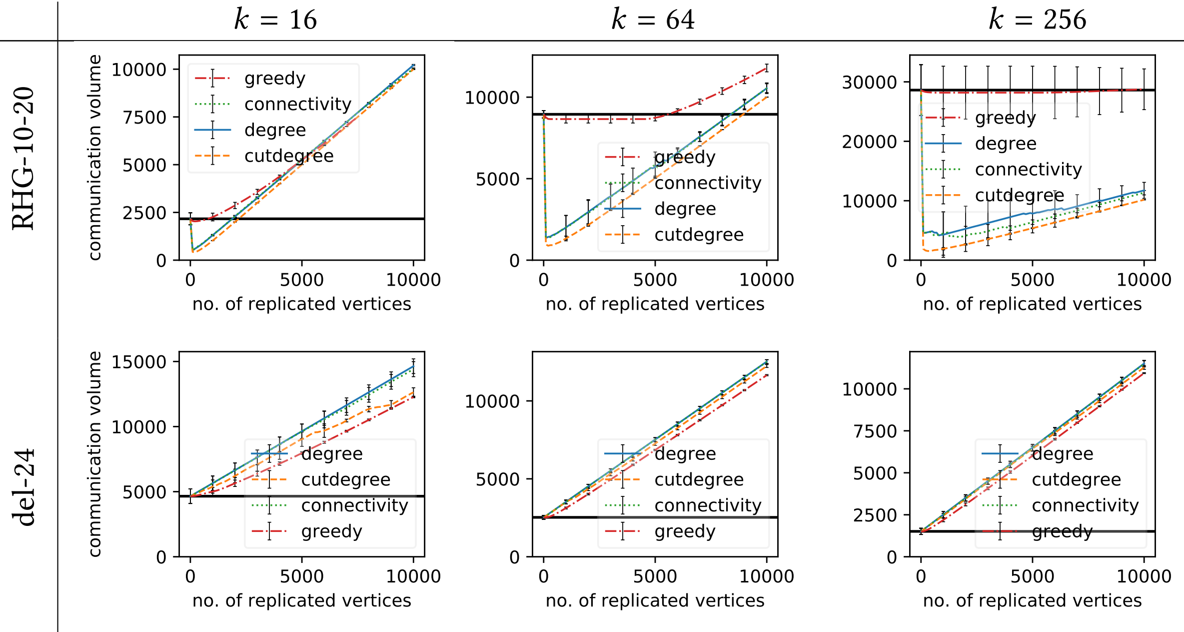




k = 16
k = 64
k = 256
as-skitter
amazon-2008
eu-2005
in-2004
uk-2002
communication volume
no. of replicated vertices
greedy
degree
connectivity
cutdegree



k = 16
k = 64
k = 256
arabic-2005
ba-1M-10
RHG-1-10
RHG-1-20
RHG-10-100
communication volume
no. of replicated vertices
greedy
degree
connectivity
cutdegree



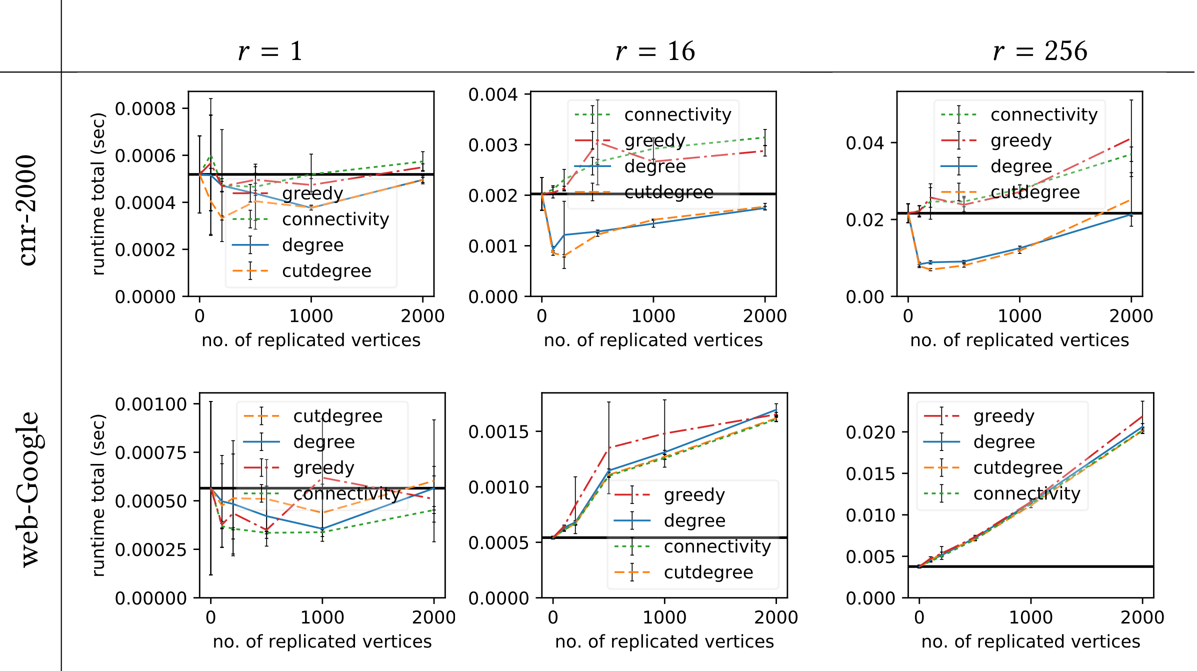
B.2. Runtime Measurements for SpMV

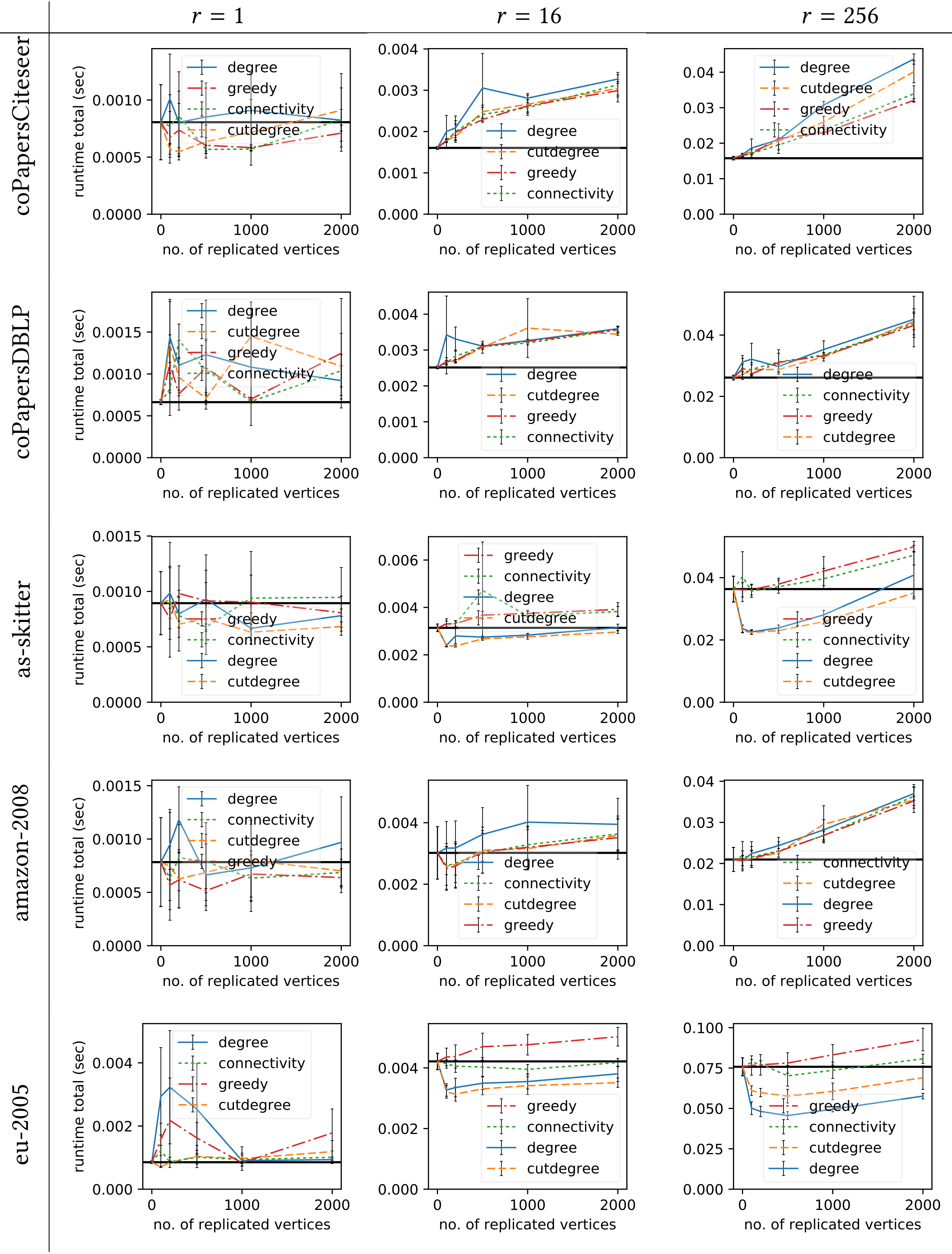
Table B.3.: Total bottleneck runtime (ms) for distributed SpMV on 16 nodes for different replication heuristics, input graphs and numbers of right-hand sides r like in Table 5.3

graph	no repl.	`degree`	`cutdegree`	`connectivity`	`greedy`
			$r = 1$		
cnr-2000	0.36(± 0.05)	0.33(± 0.01)	**0.27**(± 0.05)	0.37(± 0.03)	0.36(± 0.02)
web-Google	**0.16**(± 0.01)	0.47(± 0.20)	0.18(± 0.00)	0.18(± 0.00)	0.18(± 0.00)
coCiteseer	**0.85**(± 0.08)	1.14(± 0.13)	0.97(± 0.18)	0.94(± 0.21)	0.87(± 0.06)
coDBLP	0.77(± 0.04)	0.96(± 0.15)	0.90(± 0.35)	1.85(± 1.17)	**0.77**(± 0.05)
as-skitter	0.62(± 0.03)	0.57(± 0.10)	**0.49**(± 0.01)	0.63(± 0.07)	0.61(± 0.02)
amazon-2008	0.67(± 0.36)	0.66(± 0.28)	0.53(± 0.21)	0.42(± 0.00)	**0.42**(± 0.00)
eu-2005	1.24(± 0.14)	1.46(± 0.28)	**1.24**(± 0.07)	1.25(± 0.16)	1.31(± 0.25)
in-2004	1.18(± 0.36)	1.95(± 1.55)	1.22(± 0.47)	1.24(± 0.60)	**1.10**(± 0.44)
uk-2002	12.14(± 0.59)	12.10(± 0.42)	**12.06**(± 0.54)	12.26(± 0.48)	12.09(± 0.45)
arabic-2005	29.05(± 1.99)	31.00(± 2.58)	**28.16**(± 1.53)	29.41(± 1.48)	29.14(± 1.08)
RHG-1-10	1.07(± 1.56)	0.72(± 0.68)	**0.25**(± 0.01)	0.72(± 0.73)	0.28(± 0.00)
RHG-1-20	1.57(± 2.17)	2.75(± 2.25)	1.22(± 0.50)	2.47(± 1.14)	**0.37**(± 0.01)
RHG-10-100	3.94(± 0.60)	4.09(± 0.31)	3.90(± 0.24)	4.05(± 0.37)	**3.61**(± 0.02)
RHG-10-200	7.15(± 1.30)	6.74(± 0.13)	6.73(± 0.02)	6.75(± 0.09)	**6.55**(± 0.06)
del-24	3.24(± 0.75)	6.20(± 3.84)	**2.63**(± 0.17)	3.05(± 0.63)	2.85(± 0.69)
			$r = 256$		
cnr-2000	27.40(± 4.98)	14.18(± 1.81)	**13.52**(± 1.36)	29.75(± 6.06)	29.40(± 5.94)
web-Google	**9.33**(± 0.14)	16.08(± 1.58)	11.16(± 0.36)	13.27(± 2.47)	11.58(± 1.54)
coCiteseer	**33.54**(± 0.39)	39.28(± 3.54)	35.09(± 1.40)	40.21(± 8.12)	36.88(± 4.36)

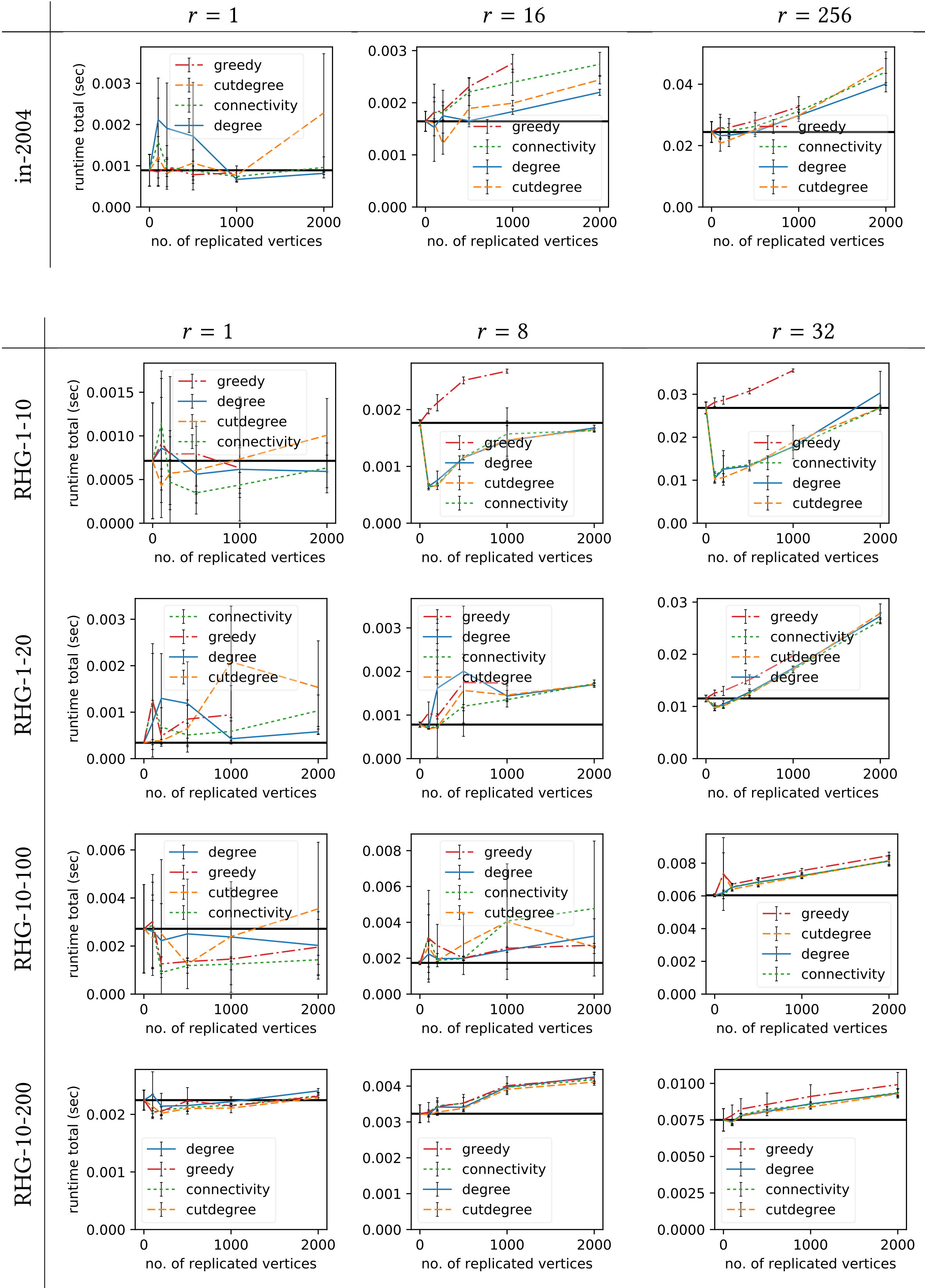
coDBLP	60.43(± 10.91)	58.73(± 1.87)	**56.77**(± 0.48)	57.39(± 0.45)	59.82(± 7.10)
as-skitter	57.44(± 3.94)	53.18(± 9.36)	**47.65**(± 2.16)	58.58(± 2.44)	58.13(± 3.05)
amazon-2008	**36.46**(± 0.59)	37.26(± 0.50)	37.02(± 0.52)	37.15(± 0.51)	36.90(± 0.37)
eu-2005	122.07(± 10.87)	**79.00**(± 4.68)	111.57(± 3.92)	138.14(± 20.09)	118.67(± 10.98)
in-2004	53.29(± 1.65)	**43.10**(± 1.39)	53.34(± 1.75)	56.72(± 2.55)	53.04(± 2.04)
RHG-1-10	**37.73**(± 2.19)	41.69(± 0.95)	39.80(± 3.11)	41.54(± 1.79)	39.19(± 2.07)
RHG-1-20	**21.54**(± 0.15)	23.19(± 1.42)	22.23(± 0.06)	22.79(± 1.41)	22.58(± 0.28)
			$r = 32$		
uk-2002	56.95(± 2.60)	56.98(± 3.82)	**54.63**(± 3.46)	56.54(± 3.68)	56.85(± 2.95)
arabic-2005	145.29(± 11.16)	**112.20**(± 5.72)	113.58(± 11.43)	145.60(± 10.25)	141.11(± 8.67)
RHG-10-100	**13.12**(± 0.20)	14.99(± 0.18)	14.96(± 0.22)	15.11(± 0.28)	13.37(± 0.25)
RHG-10-200	**17.35**(± 0.56)	18.53(± 0.34)	18.53(± 0.42)	18.35(± 0.28)	17.46(± 0.60)
del-24	**29.54**(± 0.44)	30.36(± 0.10)	30.22(± 0.20)	30.91(± 0.74)	30.09(± 0.12)

Table B.4.: Distributed SpMV runtime for different replication heuristics and counts as well as different numbers r of right-hand sides on the ForHLR2 cluster. Each graph was partitioned with 4 different random seeds into $k = 64$ parts, each kernel was executed 10 times. The plots show the average bottleneck runtime with its standard deviation.





r = 1
r = 16
r = 256
coPapersCiteseer
coPapersDBLP
as-skitter
amazon-2008
eu-2005
runtime total (sec)
no. of replicated vertices
degree
greedy
connectivity
cutdegree



r = 1
r = 16
r = 256
in-2004
r = 1
r = 8
r = 32
RHG-1-10
RHG-1-20
RHG-10-100
RHG-10-200
runtime total (sec)
no. of replicated vertices
greedy
cutdegree
connectivity
degree

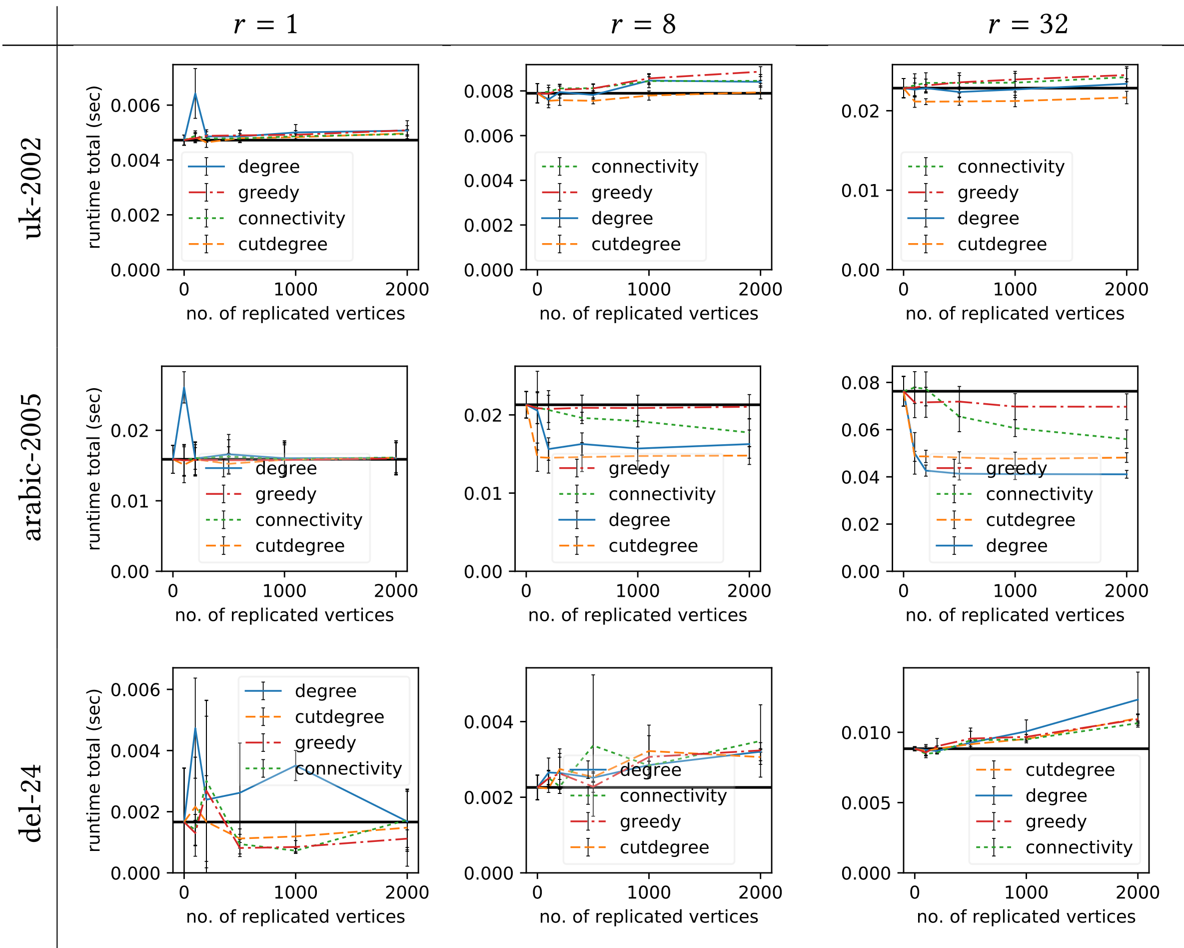
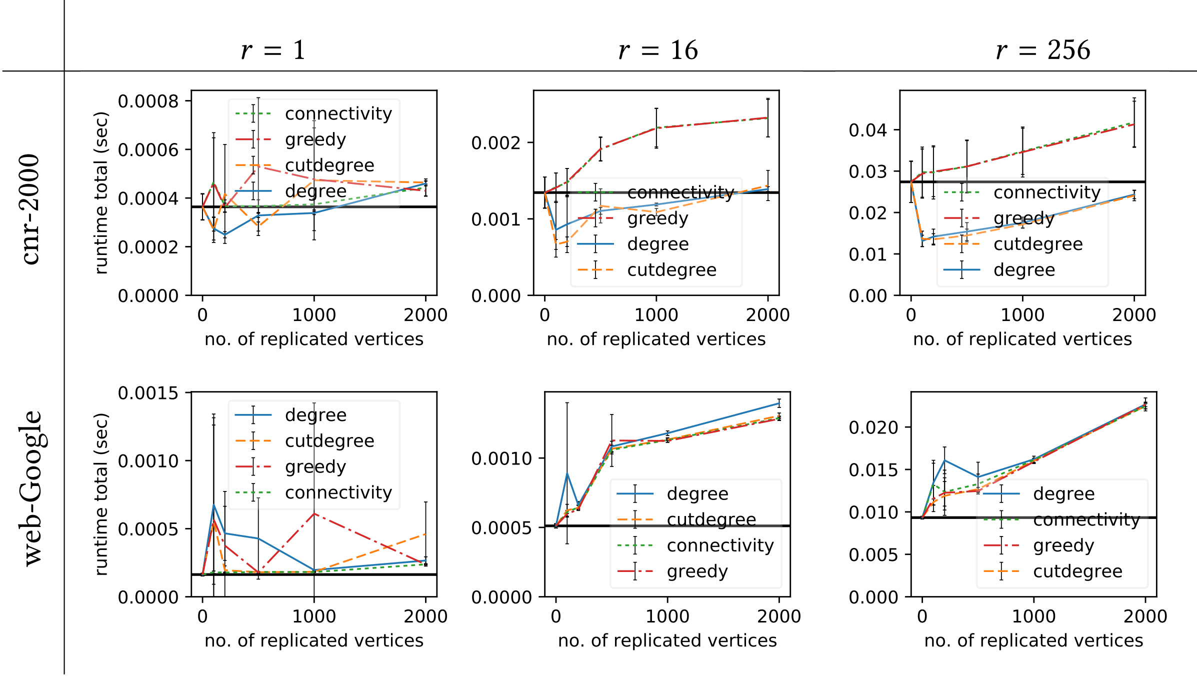
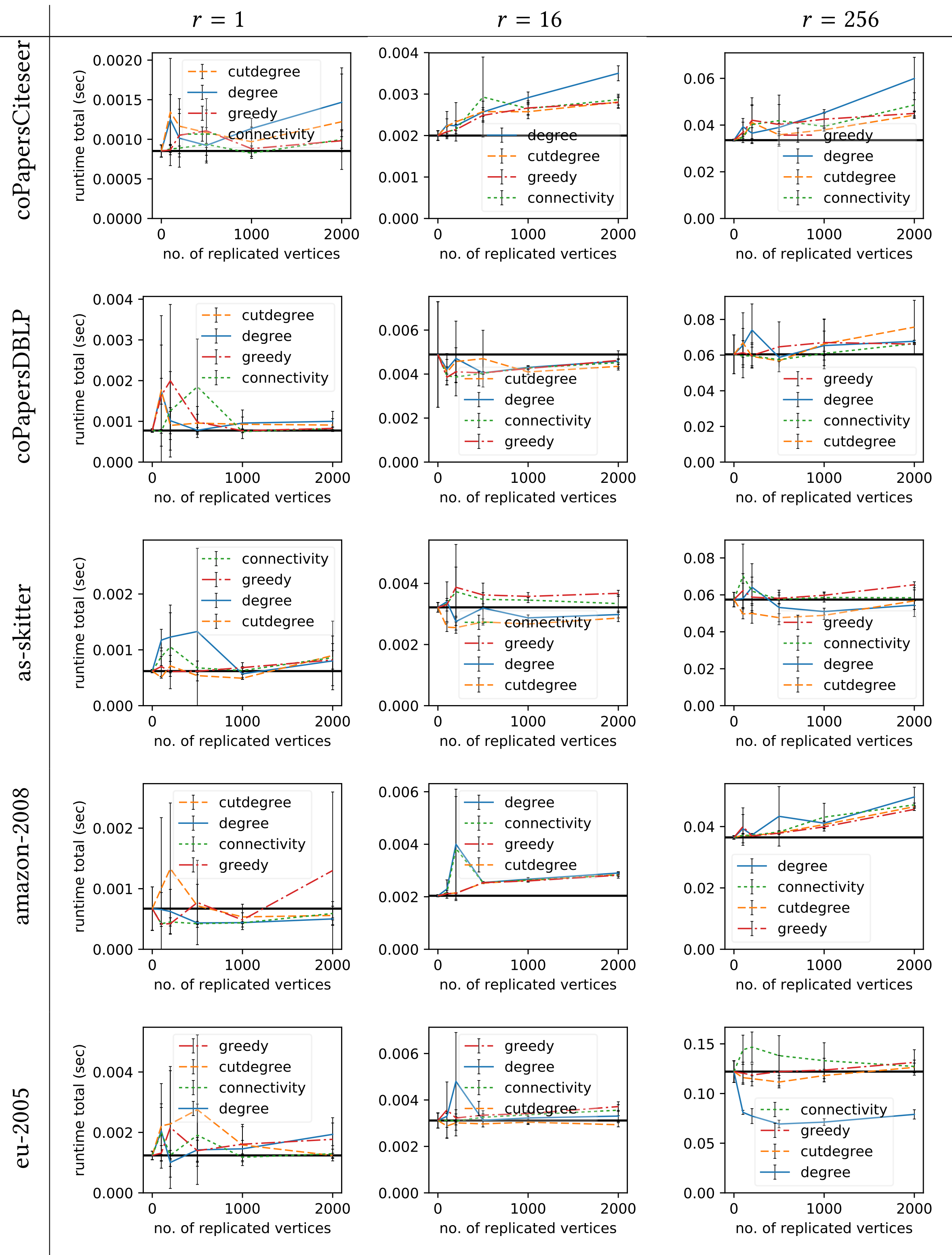
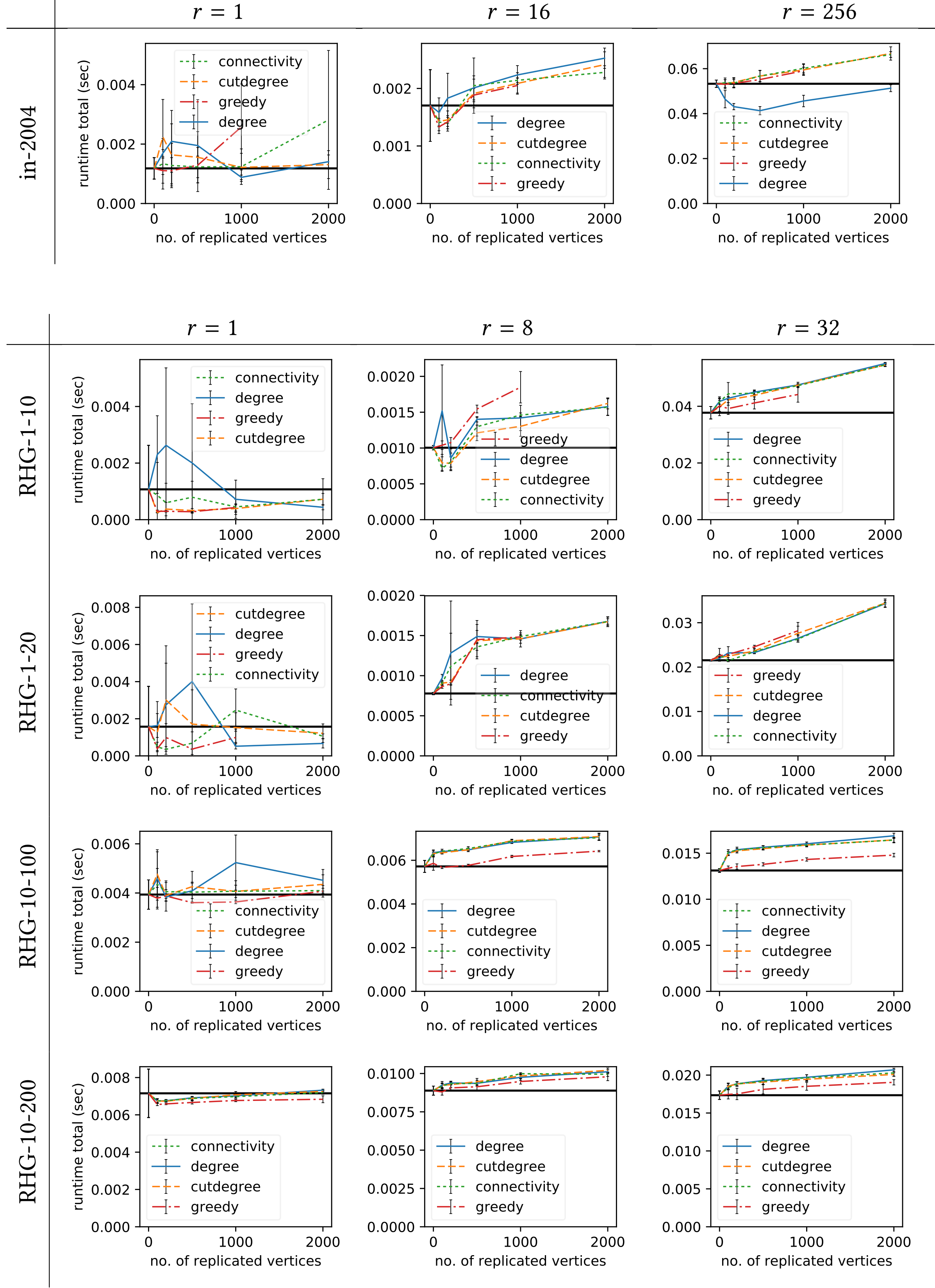


Table B.6.: Distributed SpMV runtime like in Table B.4 for $k = 16$ parts

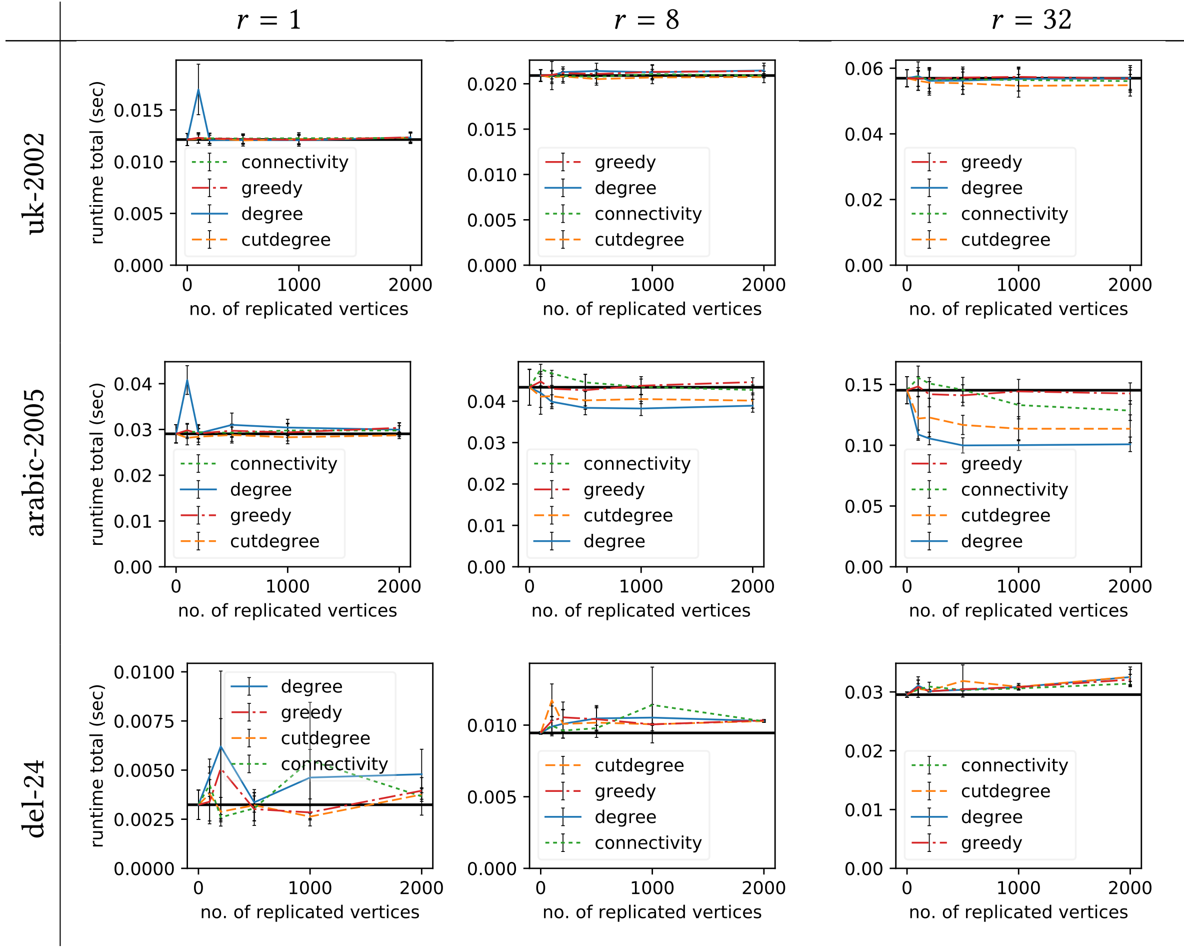




r = 1
r = 16
r = 256
coPapersCiteseer
coPapersDBLP
as-skitter
amazon-2008
eu-2005
runtime total (sec)
no. of replicated vertices
cutdegree
degree
greedy
connectivity



r = 1
r = 16
r = 256
in-2004
runtime total (sec)
no. of replicated vertices
r = 1
r = 8
r = 32
RHG-1-10
RHG-1-20
RHG-10-100
RHG-10-200
connectivity
cutdegree
greedy
degree



r = 1
r = 8
r = 32
uk-2002
arabic-2005
del-24
runtime total (sec)
no. of replicated vertices
connectivity
greedy
degree
cutdegree