Topology-aware Heterogeneous Multi-level Partitioner

Deliverable No: D3.4
Deliverable Title: Topology-aware Heterogeneous Multi-level Partitioner
Deliverable Publish Date: 31 March 2024

Project Title: SparCity: An Optimization and Co-design Framework for Sparse Computation
Call ID: H2020-JTI-EuroHPC-2019-1
Project No: 956213
Project Duration: 36 months
Project Start Date: 1 April 2021
Contact: sparcity-project-group@ku.edu.tr

List of partners:

Participant no.	Participant organisation name	Short name	Country
1 (Coordinator)	Koç University	KU	Turkey
2	Sabancı University	SU	Turkey
3	Simula Research Laboratory AS	Simula	Norway
4	Instituto de Engenharia de Sistemas e Computadores, Investigação e Desenvolvimento em Lisboa	INESC-ID	Portugal
5	Ludwig-Maximilians-Universität München	LMU	Germany
6	Graphcore AS*	Graphcore	Norway

*until M21

CONTENTS

1 Introduction 1
1.1 Objectives of this Deliverable 1
1.2 Work Performed 1

2 Fast Partitioning on Modern Architectures 2
2.1 Contributions by Each Partner 2
2.2 Deviations (if Any) 2

3 Distributed SpMV on Graphcore IPUs 2
3.1 Partitioning for sparse computations 2
3.2 Distributed sparse matrix-vector multiplication 3
3.2.1 Partitioning the matrix rows 3
3.2.2 Distributing the input vector 4
3.2.3 Parallel overhead 4
3.3 Numerical experiments 5
3.3.1 Example of partitioning and irregular communication 5
3.3.2 Partitioning for Graphcore GC200 IPUs 6

4 Using Graph Embedding for Graph Partitioning 9
4.1 History of Changes 13

1 INTRODUCTION

The SparCity project is funded by EuroHPC JU (the European High Performance Computing Joint Undertaking) under the 2019 call for Extreme-Scale Computing and Data-Driven Technologies for research and innovation actions. SparCity aims to create a supercomputing framework that will provide efficient algorithms and coherent tools specifically designed for maximizing the performance and energy efficiency of sparse computations on emerging High-Performance Computing (HPC) systems, while also opening up new usage areas for sparse computations in data analytics and deep learning.

Sparse computations are commonly found at the heart of many important applications, but at the same time, it is challenging to achieve high performance when performing sparse computations. SparCity delivers a coherent collection of innovative algorithms and tools for enabling high efficiency of sparse computations on emerging hardware platforms. More specifically, the objectives of the project are:

- to develop a comprehensive application and data characterization mechanism for sparse computation based on state-of-the-art analytical and machine-learning-based performance and energy models,
- to develop advanced node-level static and dynamic code optimizations designed for massive and heterogeneous parallel architectures with complex memory hierarchy for sparse computation,
- to devise topology-aware partitioning algorithms and communication optimizations to boost the efficiency of system-level parallelism,
- to create digital SuperTwins of supercomputers to evaluate and simulate what-if hardware scenarios,
- to demonstrate the effectiveness and usability of the SparCity framework by enhancing the computing scale and energy efficiency of challenging real-life applications.
- to deliver a robust, well-supported and documented SparCity framework into the hands of computational scientists, data analysts, and deep learning end-users from industry and academia.

1.1 OBJECTIVES OF THIS DELIVERABLE

The objective of Deliverable 3.4 is to describe the development and evaluation of our work for Task 3.4 Topology-aware hierarchical partitioning. In this task a hierarchical multi-level partitioning approach is developed that can take into consideration the complex topology of modern HPC systems by applying different partitioning strategies at each level. This partitioner is especially important for large graphs coming from deep neural networks and social networks.

1.2 WORK PERFORMED

For this deliverable, there are two lines of work: The main line investigates the performance of distributed SpMV on Graphcore IPUs. On this architecture, partitioning poses an especially tough challenge because of the limited amount of memory on the processor tiles. The performance of this kernel is analyzed under different settings and different architectures; multicore CPU nodes and Graphcore GC200 IPU. In the second line, we experimented using GPUs for partitioning by exploiting a graph embedding tool. Unfortunately, we were not able to obtain good results. However, we aim to continue this line of work in the future.

2 FAST PARTITIONING ON MODERN ARCHITECTURES

2.1 CONTRIBUTIONS BY EACH PARTNER

SIMULA performed the work described in Section 3. SU and KU equally contributed to the study in Section 4.

2.2 DEVIATIONS (IF ANY)

For this task, the initial plan was developing an in-house partitioner. However, experimenting with existing partitioners such as SCOTCH that can take the topology graph into account and perform mapping, we decided that the hierarchical multi-level partitioning approach, which is the main target Task 3.4, can be built on top of them.

3 DISTRIBUTED SPMV ON GRAPHCORE IPUS

3.1 PARTITIONING FOR SPARSE COMPUTATIONS

Partitioning is essential for distributing parallel workloads. The predominant paradigm for parallel, sparse computations is to represent data and computations as a graph (or hypergraph), thereby reflecting the, often irregular, connections between the underlying data and the computations to be performed. The (hyper)graph is then partitioned to determine how the data and computations should be distributed across available processors. Moreover, the resulting partitioning must be of high quality to ensure that work is well balanced among processors and to minimise parallel overhead, e.g., due to additional synchronisation or communication that is incurred from the parallelisation itself.

In this report, we consider the impact of partitioning on distributed-memory parallel sparse matrix-vector multiplication (SpMV), a sparse linear algebra kernel that is frequently recurring in scientific computing and graph processing. It is especially important in sparse linear solvers, and it is used, for example, in the cardiac modelling application[1] that serves as one of the real-world use cases for demonstrating methods and tools developed in the SparCity project.

For sparse computations on the Graphcore IPU,[2] partitioning poses an especially tough challenge because of the limited amount of memory on the processor tiles. Each of the 1 472 tiles on a GC200 IPU is equipped with only 624 KB of SRAM, thus limiting the amount of data that can be stored on the tile and imposing severe restrictions for parallelising and distributing sparse workloads. If even a single part of the data exceeds the memory capacity of a tile after partitioning, then the computation fails and cannot be carried out without adding more memory. To use the IPUs effectively, partitioning should aim to minimise parallel overhead and, ideally, limit the memory footprint on each processor. Although the IPU presents a somewhat extreme example, high quality partitioning and low parallel overhead is equally important for other parallel computing architectures, such as clusters of multicore CPUs or GPUs.

In the following, we analyse the parallel overhead associated with performing parallel SpMV in a distributed-memory setting. We also present results from numerical experiments that concern

[1] J. Langguth et al. "Parallel performance modeling of irregular applications in cell-centered finite volume methods over unstructured tetrahedral meshes". *Journal of Parallel and Distributed Computing* 76 (2015). Special Issue on Architecture and Algorithms for Irregular Applications, pp. 120–131. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2014.10.005.

[2] Luk Burchard et al. "Enabling Unstructured-Mesh Computation on Massively Tiled AI-Processors: An Example of Accelerating In-Silico Cardiac Simulation". *Frontiers in Physics* 11 (2023). DOI: 10.3389/fphy.2023.979699.

partitioning sparse matrices for multicore CPU nodes and for the Graphcore GC200 IPU.

3.2 DISTRIBUTED SPARSE MATRIX-VECTOR MULTIPLICATION

This section describes a parallel sparse matrix-vector multiplication algorithm for P processors with distributed, private memories. After describing the algorithm and associated data structures, we briefly discuss the parallel overhead in terms of memory footprint and communication requirements that result from parallelising the computation.

Let A denote a square, N-by-N matrix, and let x and y denote vectors of size N. The aim of the SpMV operation is to compute the product,

$$y \leftarrow Ax + y, \tag{1}$$

in the case of A being a sparse matrix. Suppose therefore that A has K nonzeros $a_{i_1,j_1}, a_{i_2,j_2}, \ldots, a_{i_K,j_K} \in \mathbb{R}$, where i_k and j_k denote the row and column, respectively, of the k-th nonzero for $k = 1, 2, ..., K$. Generally speaking, we assume that K is much smaller than the total number of matrix entries $N \times N$.

3.2.1 PARTITIONING THE MATRIX ROWS

For the purpose of computing the matrix-vector product in parallel, it is natural to perform a rowwise decomposition of the matrix A. This is easily seen after expressing the matrix-vector product componentwise, i.e.,

$$y_{i_k} \leftarrow \sum_k a_{i_k,j_k} x_{j_k} + y_{i_k}, \tag{2}$$

and noting that every row y_i can be computed independently.

The first step is therefore to partition the rows of A into P parts. A simple choice is to partition the set $\{1, 2, ..., N\}$ into contiguous blocks of equal size (e.g., each block having $\lfloor N/P \rfloor$ rows, plus some remainder if N is not divisible by P). It is, however, common practice to apply a graph partitioner[3] to divide the matrix rows in a way that balances the number of nonzeros in each part and, as we discuss later, also reduces the need for communication in parallel computations.

If A is symmetric, then the graph to be partitioned is precisely the undirected graph that arises from considering A as an adjacency matrix. Otherwise, if A is unsymmetric, an undirected graph is instead obtained by taking the symmetrisation $A + A^T$ as the adjacency matrix. If necessary, weights can be assigned to nodes (or edges) of the graph to reflect certain costs associated with rows (or nonzeros). For the purpose of this report, unweighted graphs are used to perform the partitioning in the numerical experiments described later.

Given a row partitioning into P parts, the matrix is decomposed accordingly, $A = \sum_{p=1}^{P} A^{(p)}$. The matrix $A^{(p)}$ consists of all nonzeros from A that belong to the m rows $i_1, i_2, ..., i_m$ that were assigned to the p-th part of the row partitioning. The output vector y is likewise decomposed, $y = \sum_{p=1}^{P} y^{(p)}$, such that $y^{(p)}$ consists of the same subset of m rows as the matrix $A^{(p)}$. The

[3] G. Karypis and V. Kumar. "Multilevel algorithms for multi-constraint graph partitioning". *Proceedings of the IEEE/ACM SC98 Conference* (1998). DOI: 10.1109/sc.1998.10018; François Pellegrini and Jean Roman. "Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs". *High-Performance Computing and Networking*. Springer Berlin Heidelberg, 1996, pp. 493–498; Peter Sanders and Christian Schulz. "Think Locally, Act Globally: Highly Balanced Graph Partitioning". *Experimental Algorithms, 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings*. Vol. 7933. Springer, 2013, pp. 164–175; E. G. Boman et al. "The Zoltan and Isorropia Parallel Toolkits for Combinatorial Scientific Computing: Partitioning, Ordering, and Coloring". *Scientific Programming* 20.2 (2012), pp. 129–150.

SpMV operation in Eq. (2) is thus reduced to independent matrix-vector products,

$$y^{(p)} \leftarrow A^{(p)}x + y^{(p)}, \tag{3}$$

for each part $p = 1, 2, ..., P$. Now, after distributing $A^{(p)}$ and $y^{(p)}$ to process p, these matrix-vector products may be computed in parallel.

3.2.2 Distributing the input vector

To achieve a scalable SpMV algorithm, it is also necessary to decompose and distribute the input vector x. Note that a given input vector value, x_j, may be needed in several of the matrix-vector products in Eq. (3). In fact, the value x_j is needed to compute $y^{(p)}$ if the matrix $A^{(p)}$ contains one or more nonzeros in column j. A straightforward, rowwise distribution of x is therefore not sufficient, and more effort is needed to suitably decompose the input vector.

First, note that to compute $y^{(p)}$, we need only the values of x that correspond to non-empty columns of $A^{(p)}$, i.e., columns having one or more nonzeros. Second, for a given part of the matrix, $A^{(p)}$, we categorise its non-empty columns into the following three groups:

1. Columns owned exclusively by the given part, meaning that nonzeros of A in those columns are located only in the rows belonging to $A^{(p)}$ and not in any other rows.

2. Shared columns owned by the given part, meaning any column j such that row j belongs to $A^{(p)}$, according to the rowwise matrix partitioning, and, additionally, one or more nonzeros of A are located in rows belonging to another part $A^{(q)}$, such that $p \neq q$.

3. Shared columns owned by other parts, meaning any column j such that row j belongs to $A^{(q)}$, according to the rowwise matrix partitioning, and, additionally, one or more nonzeros of A are located in rows belonging to $A^{(p)}$, where $p \neq q$.

The categories identified above can be used to explain the communication needed to compute a given part of the output vector $y^{(p)}$.

More precisely, the first category relates to input vector values x_j that are only needed to compute a single part of the output vector, $y^{(p)}$. As a result, distributing x_j together with $y^{(p)}$ and $A^{(p)}$ means that no communication is required with respect to x_j. The remaining two categories, on the other hand, concern input vector values that are needed to compute multiple parts of the output vector. These values must therefore be exchanged before computing the corresponding matrix-vector products. Suppose that the input vector x is partitioned and distributed in the same way as the output vector y. Then the value x_j must be sent from process p to one or more other processes if column j belongs to the second category. Conversely, the value x_j must be received by process p from another process if column j belongs to the third category.

3.2.3 Parallel overhead

The parallel overhead of the distributed SpMV algorithm can be attributed to matrix columns that are shared by different parts of the matrix after partitioning. If a given input vector value x_j is assigned to a process p, then the process incurs communication overhead proportional to the number of other processes having nonzeros in column j in their part of the matrix. Moreover, a process incurs an overhead in terms of memory footprint for every non-empty column j in its part of the matrix for which it does not own the corresponding input vector value x_j.

sender		recipient 1	2	3	4
	1	-	65.20	21.07	24.54
	2	53.55	-	49.52	7.40
	3	21.59	58.84	-	60.92
	4	24.45	7.51	53.58	-

(a) *block partitioning*

sender		recipient 1	2	3	4
	1	-	39.27	0	19.45
	2	38.72	-	38.02	25.24
	3	0	38.05	-	38.70
	4	19.39	25.14	38.40	-

(b) *METIS*

sender		recipient 1	2	3	4
	1	-	39.91	10.80	31.87
	2	40.23	-	27.18	5.70
	3	10.89	26.63	-	31.45
	4	31.12	5.77	31.55	-

(c) *SCOTCH*

Table 1 *Message sizes (in KiB) for each pair of communicating processes when performing distributed SpMV with the "heart03" matrix partitioned into 4 parts. Results are shown for three different partitioning methods.*

3.3 NUMERICAL EXPERIMENTS

In the following experiments, we consider sparse matrices obtained from a cardiac monodomain solver,[4] which has also been ported to Graphcore's IPU.[5] We partition the matrices using three different methods, including a simple block partitioning that divides matrix rows into equal-sized contiguous blocks, as well as partitioning the matrix rows using the graph partitioners METIS[6] and SCOTCH.[7] The graph partitioners are configured with their default options.

3.3.1 EXAMPLE OF PARTITIONING AND IRREGULAR COMMUNICATION

As an initial example, we consider the matrix "heart03", which consists of 1 607 708 rows and columns and a total of 23 597 002 nonzeros, and partition it into 4 parts. First, we note that the block partitioning results in exactly the same number of rows in each part, whereas METIS and SCOTCH produce partitions where the number of rows in each part differs by less than 1 %. Moreover, the number of nonzeros in each part differs by 5.1 %, 2.2 % and 4.0 % for block partitioning, METIS and SCOTCH, respectively. These results are expected because the matrix in question has almost exactly the same number of nonzeros in every row, which is a property of the particular numerical method from which the matrix arises.

Next, Table 1 shows the communication volume (in KiB) of each process needed to perform the SpMV operation after partitioning the "heart03" matrix into 4 parts. We note that the communication varies considerably between processes. For example, the ratios between the largest and smallest (non-empty) messages are 8.8×, 2.0× and 7.1× for block partitioning, METIS and SCOTCH, respectively. Also, the partitioning produced by METIS requires fewer messages, because processes 1 and 3 do not need to exchange data.

In spite of the graph partitioners' ability to balance the number of rows and nonzeros between parts, as explained above, the communication between processors is far from being balanced. Moreover, in practice, the speed of point-to-point communications between pairs of processors depends on whether they are located, for instance, on the same socket, on a different socket on the same node, or on different nodes.[8] As a result, the imbalance in communication may be

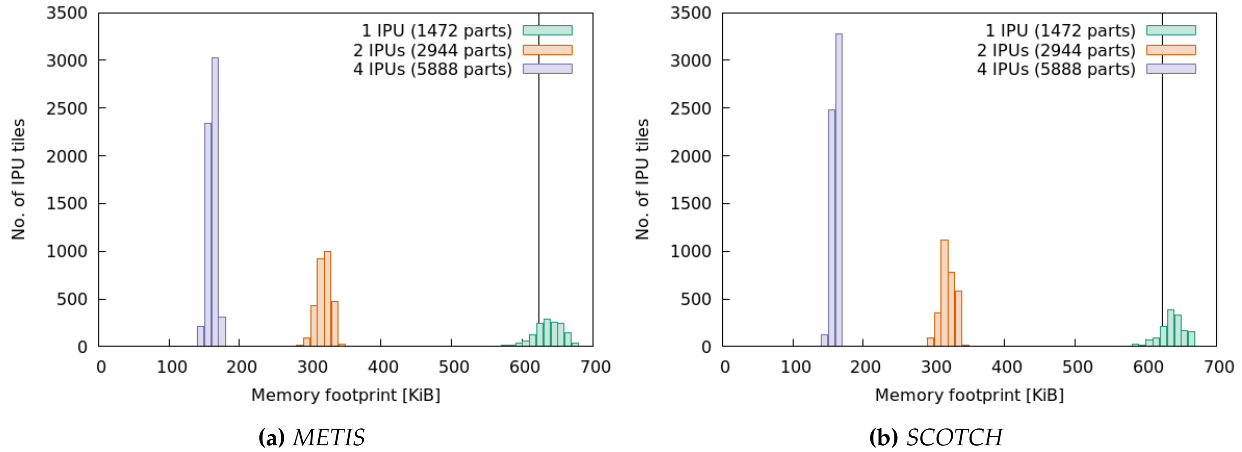
[4]Langguth et al., "Parallel performance modeling of irregular applications in cell-centered finite volume methods over unstructured tetrahedral meshes".

[5]Burchard et al., "Enabling Unstructured-Mesh Computation on Massively Tiled AI-Processors: An Example of Accelerating In-Silico Cardiac Simulation".

[6]Karypis and Kumar, "Multilevel algorithms for multi-constraint graph partitioning".

[7]Pellegrini and Roman, "Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs".

[8]Andreas Thune et al. "Detailed Modeling of Heterogeneous and Contention-Constrained Point-to-Point MPI Communication". *IEEE Transactions on Parallel and Distributed Systems* 34.5 (2023), pp. 1580–1593. DOI: 10.1109/TPDS.2023.3253881.



(a) *METIS* **(b)** *SCOTCH*

Figure 1 *Memory footprint (in KiB) per IPU tile for parallel SpMV with the "hearto5" matrix on up to 4 GC200 IPUs.*

exacerbated if processes with higher communication volumes are placed far apart or suffer from a slow interconnection.

A hierarchical or topology-aware mapping of the parallel SpMV computation onto the underlying processors can, in principle, account for both irregular communication patterns and non-uniform speed of communication between processors. The general idea is that pairs of submatrices having the highest communication volumes should be mapped to pairs of processors with high-speed communication links. One option is to use more advanced features of the SCOTCH[9] graph partitioner, which can perform topology-aware graph partitioning using a dual recursive bipartitioning method.[10] Various common topologies are supported, including hypercubes, meshes, tori, and trees.

3.3.2 Partitioning for Graphcore GC200 IPUs

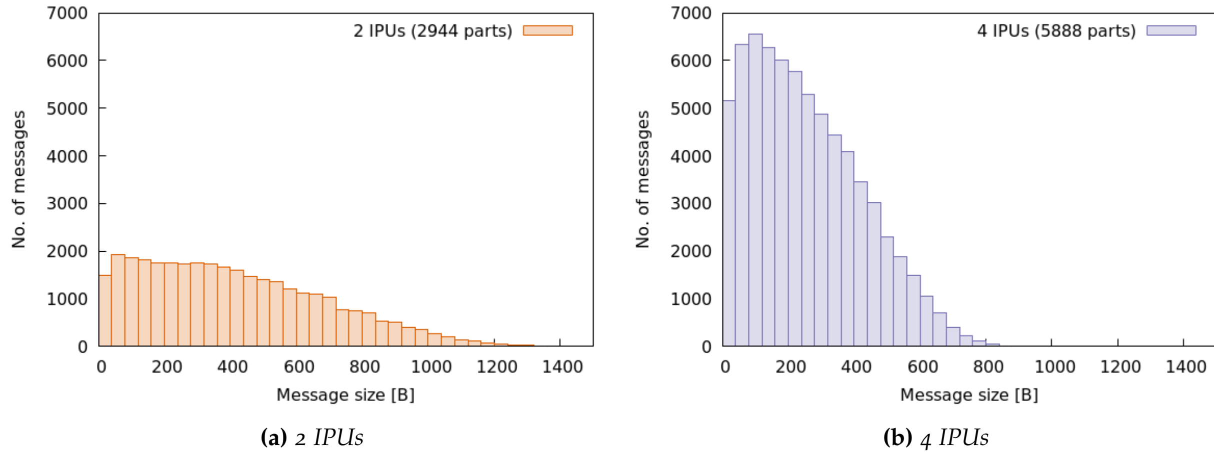
In this section, we partition the matrix "hearto5" for multiple Graphcore IPUs. The matrix has 7 205 076 rows and columns and 107 994 304 nonzeros. The memory capacity of a single IPU is 897 MiB, but the memory footprint for storing the sparse matrix (in ELLPACK format) and both vectors in single precision floating point is 878.9 MiB and is therefore unlikely to fit on a single IPU.

Figure 1 shows the memory footprint per IPU tile needed for the parallel SpMV after partitioning the matrix for up to 4 IPUs using both METIS and SCOTCH. Both partitioners produce similar results and show that 2 IPUs are needed because the memory footprint exceeds the 624 KiB capacity on many tiles.

As we also observed in the previous subsection, the message sizes involved in the point-to-point communications of parallel SpMV can vary greatly. Figure 2 shows histograms of the message sizes in the case of partitioning for 2 and 4 IPUs using METIS. The smallest messages are typically only a few bytes, but the largest messages are about 1 300 and 800 bytes when partitioning for 2 and 4 IPUs, respectively. Moreover, as shown in Figure 3, the communication volume per tile also varies considerably. Here we divide the communication into 1) messages between tiles on the same IPU and 2) messages between tiles on different IPUs. In any case,

[9]Pellegrini and Roman, "Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs".

[10]François Pellegrini. "Static mapping by dual recursive bipartitioning of process architecture graphs". *Proceedings of IEEE Scalable High Performance Computing Conference*. IEEE. 1994, pp. 486–493.

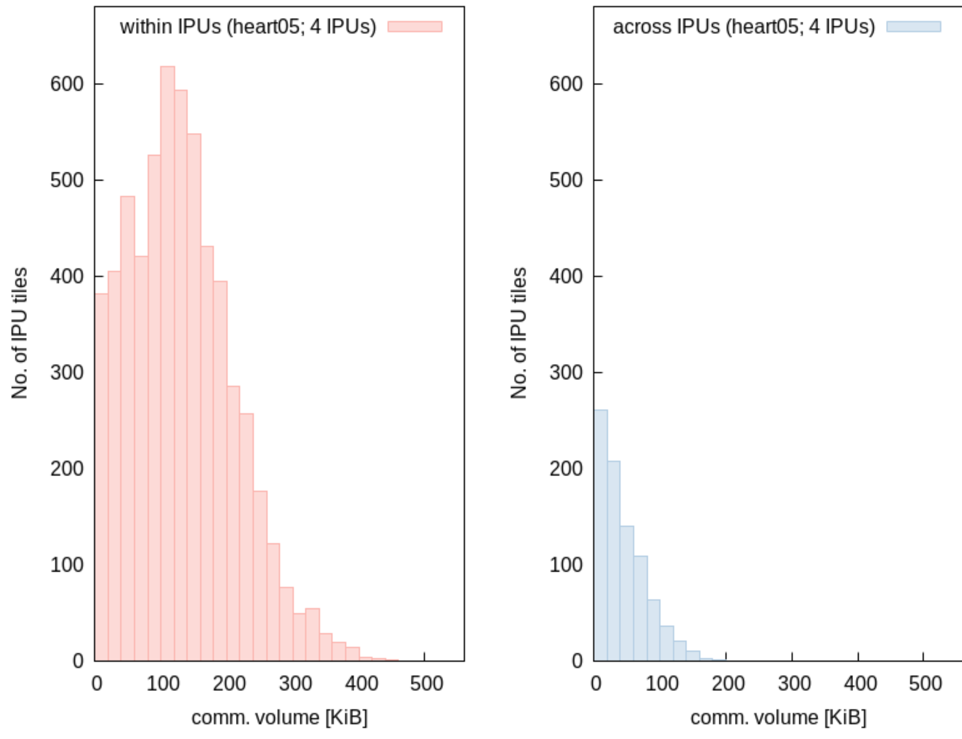


(a) *2 IPUs* **(b)** *4 IPUs*

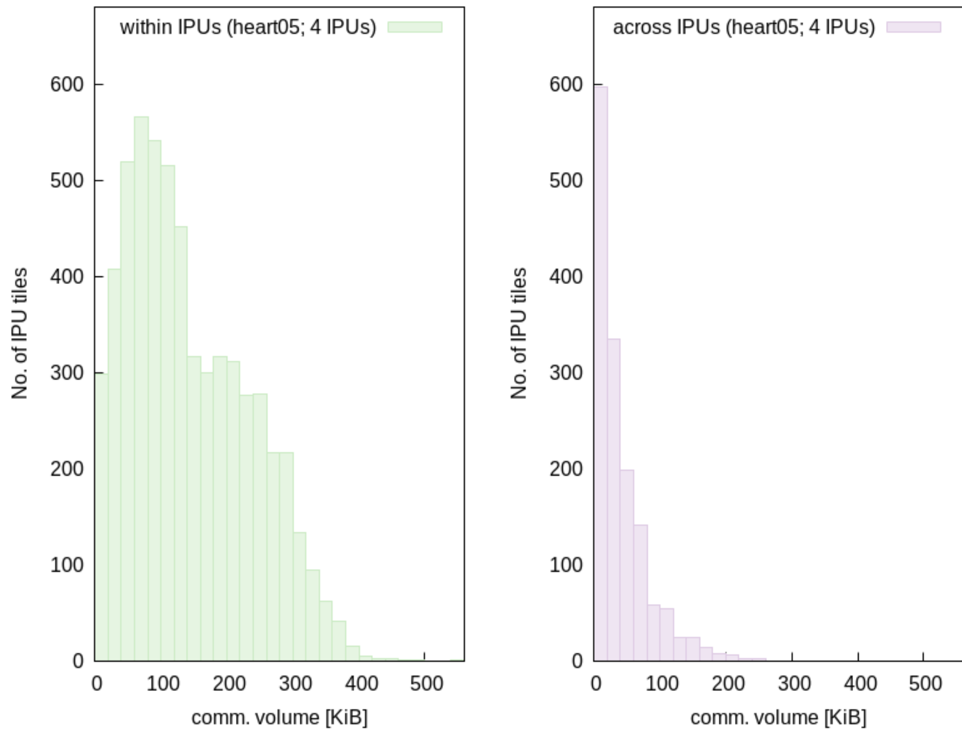
Figure 2 *Message sizes (in B) for parallel SpMV with the "heart05" matrix partitioned using METIS for 2 or 4 GC200 IPUs.*

whereas the communication volume within IPUs is about 150 KiB or less for most IPU tiles. However, some IPU tiles must communicate up to nearly 500 KiB within an IPU and 250 KiB across IPUs.

The observed imbalance of communication load across processors is bound to affect the performance of parallel SpMV computations on highly parallel architectures, such as the GC200 IPUs. Standard graph partitioners aim to reduce the communication volume, e.g., by minimising metrics such as the number of edges cut, but they do not currently allow for balancing the communication load among processors.



(a) *METIS*



(b) *SCOTCH*

Figure 3 *Communication volume (in KiB) per IPU tile for parallel SpMV with the "heart05" matrix partitioned using METIS and SCOTCH for 4 GC200 IPUs. The communication volumes within IPUs and across IPUs are shown separately.*

4 USING GRAPH EMBEDDING FOR GRAPH PARTITIONING

Graph embedding has emerged as a fundamental technique for representing complex relational data structures, such as social networks and citation networks, in a lower-dimensional space amenable to machine learning algorithms. However, embedding large graphs efficiently while preserving their structural information poses challenges due to computational complexity and resource constraints. GOSH,[11] a GPU-based tool whose development was partially funded by the SparCity project, has drawn attention for its ability to address these challenges by efficiently embedding large graphs on a single GPU. Building upon this foundation, in this report, the enhancements made on top of the GOSH algorithm, building upon its efficient potential graph embedding capabilities are described in detail. A high-level figure of how GOSH works in a multi-level fashion is given in Figure 4.

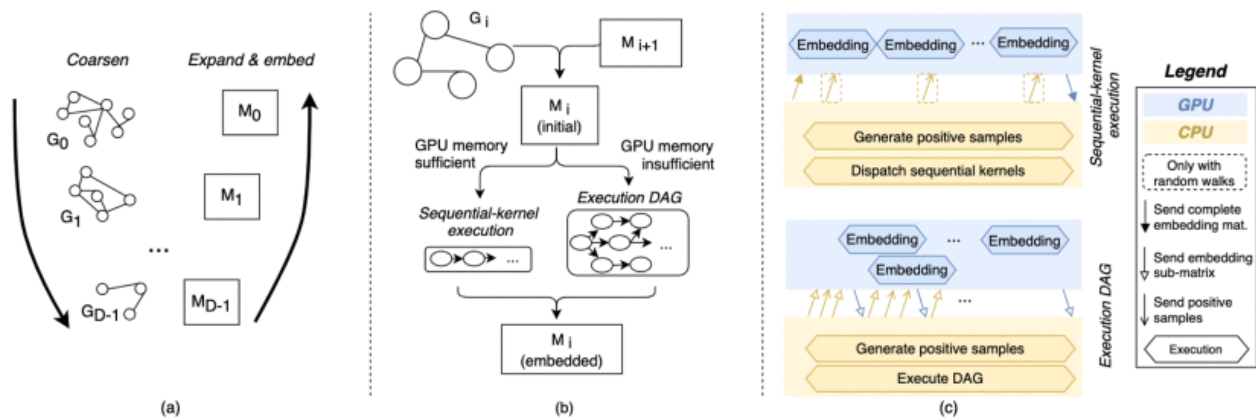


Figure 4 *An overview of GOSH: (a) The input graph, G_O, is iteratively coarsened into smaller graphs until an exit condition is met. Then, starting from the smallest, G_{D-1}, each graph G_i is embedded. (b) To embed G_i, the embedding $\mathbf{M}_{i+1}$ is projected onto G_i to initialize $\mathbf{M}_i$ which is then fine tuned. If G_i and $\mathbf{M}_i$ fits to the GPU memory GOSH carries out Sequential-Kernel Execution. Otherwise, it carries out Directed Acyclic Graph Execution to perform the embedding on a single GPU.*

The motivation comes from the fact that the embeddings can be successfully used for edge prediction as well as node classification. Hence, given the embeddings in a d-dimensional space, a well-defined *ordering* of these embeddings will generate a good partitioning. For instance, Figure 5 the results for link prediction of GOSH and other tools. As the figure shows, the AUC-ROC scores are extremely accurate. Hence, the graph embeddings can indeed learn the topology of the graph which can be useful for partitioning.

In addition to this relationship, we also aim to improve the embedding process and hence have a faster, high-quality partitioning tool. First, we need to state that GOSH uses a Stochastic Gradient Descent (SGD) based optimization technique with positive/negative updates on the embedding vectors. Our focus lies on integrating a vertex similarity measure, specifically Jaccard similarity, into this update procedure. We assume in this way this local information can be propagated to the global embedding space.

A similarity measure is a way of measuring how vertices are related (concerning their neighbourhood) to each other in a quantitative manner: the value gets higher when the neighbourhoods

[11] Amro Alabsi Aljundi, Taha Atahan Akyildiz, and Kamer Kaya. “Boosting Graph Embedding on a Single GPU”. *IEEE Transactions on Parallel and Distributed Systems* 33.11 (2022), pp. 3092–3105. DOI: 10.1109/TPDS.2021.3129617.

Algorithm	G	Time (s)	Speedup	AUC ROC	G	Time (s)	Speedup	AUC ROC	G	Time (s)	Speedup	AUC ROC	G	Time (s)	Speedup	AUC ROC
VERSE	com-dblp	128.02	1.0×	97.76	youtube	700.26	1.0×	**97.99**	com-lj	7472.89	1.0×	98.91	com-orkut	25020.28	1.0×	**98.28**
MILE		122.55	1.04×	97.70		1847.18	0.38×	94.55		3046.88	2.45×	85.89		9347.67[a]	2.68×	90.22
GV-fast		5.96	21.47×	95.72		22.48	31.15×	97.12		146.21	51.11×	98.33		456.00	45.87×	97.75
GV-slow		8.80	14.56×	97.48		34.96	20.19×	97.08		236.00	31.66×	98.36		746.25	33.53×	97.30
PBG		21.29	6.01 ×	97.73		74.42	9.41×	97.07		497.06	15.03 ×	98.40		1102.90	22.69×	98.42
GOSH-fast		0.53	241.44×	97.02		1.89	370.88×	97.67		11.38	656.55×	98.28		30.52	819.68×	**98.86**
GOSH-normal		1.41	90.95×	**97.79**		6.11	114.65×	**98.00**		39.44	189.48×	**98.74**		211.64	211.64×	98.66
GOSH-slow		2.83	45.20×	**97.95**		12.64	55.39×	**98.00**		85.26	87.65×	98.72		261.57	95.66×	98.37
GOSH-NoCoarse		19.17	6.68×	97.13		90.46	7.74×	96.51		655.00	11.41×	96.57		2228.92	11.23×	96.88

Figure 5 *VERSE and GOSH uses* $t = 16$. *MILE is a sequential tool. GPU-based tools use the V100 GPU. The speedup values are given w.r.t. VERSE. AUCROC scores are in percentage.*

are more alike. Although we focused on Jaccard similarity, various vertex similarity measurements in the literature will be investigated in the future. Jaccard similarity offers a valuable metric for measuring the similarity between sets, commonly employed in various domains, including graph analysis. Leveraging Jaccard similarity enables the capture of structural similarities between vertices based on their neighbourhood information. However, computing the similarity for all the vertex pairs is not a practical task. To integrate Jaccard similarity into the embedding process within the GOSH algorithm, a fast and efficient data structure, Bloom Filters (BF), have been used. A BF is a probabilistic data structure that is based on hashing. It is extremely space efficient and is typically used to add elements to a set and test if an element is in a set. The expectation behind this integration was to enrich the embedding process by considering local neighbourhood similarities, thereby enhancing the overall quality of the embeddings produced by the GOSH algorithm. To comprehensively assess the effectiveness of this approach, an extensive array of experiments were executed. These tests were carefully planned to see how well adding Jaccard similarity affected the quality of the embeddings and how fast the convergence was obtained. This multi-faceted evaluation was underpinned by the exploration of various configurations, encompassing a spectrum of parameters such as different numbers of positive and negative samples, learning rates, and alpha values.

We carefully selected to mirror real-world graph structures. These datasets spanned across different kinds of domains, each characterized by unique graph structures and challenges. For instance, datasets from domains such as Computational Fluid Dynamics Problems (e.g. "garon1"), Circuit Simulation Problems (e.g., "Hamm/add20"), Acoustics Problems (e.g., "Cote/tmplate"), Structural Problems (e.g., "HB/blckhole"), and Networks (e.g., "web-edu"), were selected to ensure a comprehensive evaluation across various application scenarios.

Each dataset is processed by the GOSH under a fine-tuned configuration. Subsequently, the resulting embedding values were used to sort the vertices of the respective graphs, followed by partitioning into distinct groups. In the final phase of analysis, the cut metric was computed for each of the previously mentioned partitioning approaches, thereby facilitating a granular assessment of the partitioning quality. As a baseline, the state-of-the-art METIS graph partitioner[12] was employed for partitioning the datasets into groups, serving as a benchmark for the comparison. To summarize our results, we have found that

- For social networks, the results are promising. There are cases where the partitioning quality matches with state-of-the-art tools such as METIS.

[12] George Karypis and Vipin Kumar. *METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices*. 1998.

- However, for meshes and graphs with high diameters such as road networks, the results are far from what the state-of-the-art generates.
- Integrating the similarity metrics to the SGD process is a tedious task. Although there are cases, runs, etc., that do not make the gradients explode, one needs to carefully select the learning rate schedule or normalize the similarity correctly. This is in fact why we cannot provide numerical results since we consider them they are not robust.

In conclusion, the enhancements introduced to the GOSH algorithm, such as Jaccard similarity, have been demonstrated to be promising. However, this study is far from being completed and there are various variants to be experimented with. We will continue to work on this problem in the near future.

REFERENCES

Aljundi, Amro Alabsi, Taha Atahan Akyildiz, and Kamer Kaya. "Boosting Graph Embedding on a Single GPU". *IEEE Transactions on Parallel and Distributed Systems* 33.11 (2022), pp. 3092–3105. DOI: 10.1109/TPDS.2021.3129617.

Boman, E. G. et al. "The Zoltan and Isorropia Parallel Toolkits for Combinatorial Scientific Computing: Partitioning, Ordering, and Coloring". *Scientific Programming* 20.2 (2012), pp. 129–150.

Burchard, Luk et al. "Enabling Unstructured-Mesh Computation on Massively Tiled AI-Processors: An Example of Accelerating In-Silico Cardiac Simulation". *Frontiers in Physics* 11 (2023). DOI: 10.3389/fphy.2023.979699.

Karypis, G. and V. Kumar. "Multilevel algorithms for multi-constraint graph partitioning". *Proceedings of the IEEE/ACM SC98 Conference* (1998). DOI: 10.1109/sc.1998.10018.

Karypis, George and Vipin Kumar. *METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices*. 1998.

Langguth, J. et al. "Parallel performance modeling of irregular applications in cell-centered finite volume methods over unstructured tetrahedral meshes". *Journal of Parallel and Distributed Computing* 76 (2015). Special Issue on Architecture and Algorithms for Irregular Applications, pp. 120–131. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2014.10.005.

Pellegrini, François. "Static mapping by dual recursive bipartitioning of process architecture graphs". *Proceedings of IEEE Scalable High Performance Computing Conference*. IEEE. 1994, pp. 486–493.

Pellegrini, François and Jean Roman. "Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs". *High-Performance Computing and Networking*. Springer Berlin Heidelberg, 1996, pp. 493–498.

Sanders, Peter and Christian Schulz. "Think Locally, Act Globally: Highly Balanced Graph Partitioning". *Experimental Algorithms, 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings*. Vol. 7933. Springer, 2013, pp. 164–175.

Thune, Andreas et al. "Detailed Modeling of Heterogeneous and Contention-Constrained Point-to-Point MPI Communication". *IEEE Transactions on Parallel and Distributed Systems* 34.5 (2023), pp. 1580–1593. DOI: 10.1109/TPDS.2023.3253881.

4.1 HISTORY OF CHANGES

Version	Author(s)	Date	Comment
0.1	Kamer Kaya	20/03/2024	First draft
0.2	Beyza Cavusoglu	22/03/2024	Section 4
0.3	James Trotter	27/03/2024	Section 3
0.4	Kamer Kaya	29/03/2024	Finalizing and proofreading
0.4.1	Didem Unat	29/03/2024	Minor changes and proofreading

Table 2 *Document History of Changes*