PERFORMANCE MODELS AND IMPACT OF VECTOR ARCHITECTURE ON GRAPH ALGORITHMS

by

Md Maruf Hossain

A dissertation submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
Computer Science

Charlotte

2022

Approved by:

Erik Saule, Ph.D.

Yonghong Yan, Ph.D.

Dong Dai, Ph.D.

Arun Ravindran, Ph.D.

This work dedicates to my respectful parents, siblings, and beloved wife and kids whose constant support makes this paper possible. They always inspire me. At the same time, my thanks also go to my advisor who's advice worked for this paper.

ACKNOWLEDGEMENTS

I would like to express my gratitude to my advisor Dr. Erik Saule for providing support and guidance. I got to learn a lot more about high-performance computing on graph algorithms which will be very helpful for me. In the end, I would like to thank my parents, siblings, and wife. Without them, I would not have been able to focus on my objectives. I also like to thank my dissertation committee for their guidance and supports.

ABSTRACT

MD MARUF HOSSAIN. Performance Models and Impact of Vector Architecture on Graph Algorithms (Under the direction of DR. ERIK SAULE)

Representing modern data using graphs has become more prominent in recent years. The gradual increasing size and complex structure of graphs made many classical algorithms slow. Meanwhile, graphs like social and web networks evolve over time. Traditional graph analysis that expects a fixed data set is not sufficient for these kinds of temporal graphs. It is necessary to take advantage of the modern HPC architecture and customize the applications to enhance the performance and to keep the runtime of the analysis low.

First, this dissertation analyzes modern vector architectures for graph analysis kernels. We evaluate the performance achieved on two different architectures (Skylake and Cascade Lake) and show that good hardware support for scatter instructions is necessary to fully leverage vector processing for graph partitioning problems. Then we build a performance model that analyzes the performance of graph applications for given computer architecture. We study sparse matrix-vector multiplication (SpMV) as a representative application for performance models on distributed systems, since many applications rely on basic sparse linear algebra operations from numerical solvers to graph analysis algorithms. We proposed the first model of runtime for SpMV on distributed memory machines that accounts for platform and graph structure.

Then we consider dynamically evolving graphs. In this work, we show the performance analysis of *pagerank* on the temporal graphs. Many algorithms have been proposed to compute Pagerank on evolving graphs; most of them focus on incremental ways to study temporal graphs. But the question is "if the complete data is available at the beginning of the process, and we want to know the properties of the graph over time, should we still use the streaming graph algorithm for temporal graph analysis?" We call this type of temporal analyses *Postmortem* graph analysis. In *Postmortem* analysis, one performs graph analysis on multiple subgraphs based on well-defined time-interval. In contrast, streaming algorithms mainly focus on gradually updating properties based on events like edge addition or deletion graph analysis. We show that *Postmortem* graph analysis can provide better pagerank performance on

temporal graphs than a streaming analysis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

Graphs are at the center of most modern applications today: city and road analysis [1], social media analysis [2, 3, 4], biological data processing and medical research [5, 4], academic networks [6], intelligence [7]. And with the advent of the big data era, graph size has grown exponentially in recent years. Graphs structures are so complex that the same algorithms can generate a diverse range of results of an application for different graphs. Although graph abstraction has been used for centuries, there is a renewed interest in graph processing driven in large part by emerging applications in social network analysis [8, 9, 10], science [11], and speech and image recognition [12, 13]. Lots of effort and research are going on to find out the better algorithms, implementations, frameworks, and hardware platforms for relevant graph processing applications.

In this dissertation, we will investigate how better implementation of applications can take advantage of the modern computer architecture to improve graph analysis. We believe this dissertation will enable to choose the better hardware which will enhance the performance of the application. Now, performance analysis could be done different ways like run time improvement, efficient memory usages, energy efficient *etc.* In this paper, we choose run time analysis as a performance analysis. Then we question ourselves, we can provide performance gain by comparing run time against state-of-art algorithms, but can we build a performance model that can predict run time of a application for the specific CPU architecture? In this work, we proposed multiple mechanisms to answer the questions and also discuss the difficulties to achieve the goal. We did not limited our research on only static graph, we extend our analysis towards temporal graphs as well. This dissertation could change the thought process of graph analysis for the offline temporal graph in good way.

Better hardware support is always useful to enhance the performance of any application. There have been lots of effort to improve modern processors to cope with complex graph structures and provide better support for different frameworks. Recent interest in fast graph algorithms was met with a new look at how computer architectures can be leveraged. GPUs have been understandably popular because of high flop rate, high memory bandwidth, and high power efficiency for graph

problems [14]. CPU architectures have reacted by increasing core count but also by increasing SIMD width in a move to catch up in terms of performance and energy efficiency. In particular, Intel developed AVX-512 originally for their Xeon Phi line, and are in the process of porting the instruction set in their regular CPU offering. In Chapter 3, we show that a good hardware implementation of scatter instructions is critical for efficiently vectorize graph partitioning problems.

In this work, we propose performance models for static and temporal graphs that enhance the performance of the graph algorithms with the help of vector architecture. We aim to present a performance model for graph applications that can predict the run time based on the size and structure of the graphs and the system architecture. But it is not a straightforward task to build a model for real-life graphs because of the complexity of the graph structure and the difficult pattern of the random memory accesses. Especially, if we think about the distributed system. In this work, we look for the performance model for the distributed system. Chapter 4 mainly focus on the performance model for SpMV on distributed systems. Sparse matrix-vector multiplication (*SpMV*) is one of the fundamental operations in sparse linear algebra. It is critical to solving linear systems and is widely used in engineering and scientific applications [15, 16, 17]. In this work, we propose a linear and polynomial *Support Vector Regression*(*SVR*) [18] model to predict and analyze the run time of *SpMV* on distributed systems for different ways to execute the operation.

So far, we investigated hardware support, complex graph structure, and performance model for graph applications. But, we need to consider the dynamic characteristics of the graph as well. Almost all of the social graphs evolve over time and it is important to record the time-line of these changes. These dynamic networks are called temporal graphs. The temporal graph changes through different events, such as edge addition, deletion, vertex addition, *etc*. The interest in evolving graphs is growing over time and problems like diameter change [19], the rank of web pages change [20] on the web have been investigated. One of the most common graph analyses on the temporal graph is PageRank on the dynamic networks. In this chapter 5, we propose a different graph analysis for the temporal graph; we call it *Postmortem Graph Analysis*. When a temporal graph show offline behavior it gives the opportunity to choose either streaming or non-streaming solutions. Now, if a graph shows offline behavior and we need to perform a sequence of graph analysis for the sliding window interval then call it postmortem graph analysis.

CHAPTER 2

BACKGROUND

Each chapter of this work has its own concise introduction, background, analysis, experimental setup, results and summary. But in this chapter, we are going to give basic background of all the graph kernel and algorithms that we used for the work.

2.1 Notations of Graph

A graph is denoted by $G = (V, E)$ where V and E represent the vertex and edge set respectively. Edges are represented by (u, v) pair and are associated with an edge weight $\omega : E \rightarrow \mathbb{R}^+$. We use ζ to represent the community set and communities are represented by distinct integers. We use $N(u)$ to represent the neighbor set of a vertex $u \in V$. The *volume* of a node and a community are defined as $vol(u) = \sum_{\{u,v\}:v \in N(u)} \omega(u, v) \ + \ 2 \times \omega(u, u)$ and $vol(\zeta) = \sum_{u \in \zeta} vol(u)$ respectively.

2.2 Louvain Method

The Louvain Method, first proposed by Blondel *et al.* [21], is one of the most popular methods to extract communities from a large network. It is a greedy multilevel algorithm that uses modularity as the objective function [22]. The *modularity* is defined as the fraction of edges that fall within the partitions minus the expected fraction that would be within the partition if the edges are distributed randomly. If a vertex u moves from its community C to the neighboring community D, then the modularity gain is defined by the following equation 2.1,

$$\Delta mod(u, C \rightarrow D) = \frac{\omega(u, D/\ \{u\}) - \omega(u, C/\ \{u\})}{\omega(E)} + \frac{(vol(C/\ \{u\}) - vol(D/\ \{u\})) * vol(u)}{2 * \omega(E)^2} \tag{2.1}$$

where V and E represent the vertex and edge set of a graph G respectively. Edges are represented by (u, v) pair and are associated with an edge weight $\omega : E \rightarrow \mathbb{R}^+$. We use $N(u)$ to represent the neighbor set of a vertex $u \in V$. The *volume* of a node and a community are defined as $vol(u) = \sum_{\{u,v\}:v \in N(u)} \omega(u, v) \ + \ 2 \times \omega(u, u)$ and $vol(C) = \sum_{u \in C} vol(u)$ respectively.

Based on the highest positive modularity gain, it moves to the neighbor community. The process iteratively continue until there are no vertices that move to the neighbor community or it reach to the user defined threshold limit. After each *Moves Phase*, a *Coarsening Phase* applied to the graph to shrink the graph size. The above process continue until there are no scope of *Move Phase*.

2.3 Distance-1 Graph Coloring

The distance-1 graph coloring algorithm assigns colors to the vertices of the graph so that no adjacent vertices have the same color. Minimize the number of colors is an NP-hard problem [23], and that is why various heuristic algorithms have proposed for the problem. In particular, a greedy algorithm can obtain near-optimal solutions [24]. The main focus of our work is to find out best and optimal solution for a problem based on the execution time. That is why we looking for the parallel version of the speculative distance-1 graph coloring [25, 26]. In these algorithm, initially it assigns the color to the vertices and then another method find the conflicted vertices that have the same color of their neighbors. The algorithm repeat the previous two methods on the conflicted vertices until no conflicted vertices exist.

2.4 Representation formats

There are many representation formats designed to store sparse matrices. While some representation format have been designed especially to optimze SpMV [27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46], CSR and COO are generic storage format that are used in most sparse linear algebra and graph processing.

All the encoding of sparse matrices essentially aim at avoiding to represent the zeros in matrices and only represent the non-zero values in the matrix. We will assume the matrix A is of size $n \times n$ and has nnz non-zero values.

2.4.1 Compressed Storage by Row (CSR)

The Compressed Storage by Row format represents a matrix with three arrays: `rowA`, `colA`, and `valA`. Figure 2.1 shows an example matrix and its CSR representation.

The `valA` array contains all the values that are non zero in the matrix given in the order the

$$\begin{pmatrix} 1 & 2 & 3 & 0 & 0 \\ 0 & 0 & 4 & 5 & 6 \\ 0 & 2 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 3 \\ 0 & 0 & 3 & 4 & 0 \end{pmatrix}$$

(a) A matrix

rowA = [0, 3, 6, 7, 9, 11]
colA = [0, 1, 2, 2, 3, 4, 1, 0, 4, 2, 3]
valA = [1, 2, 3, 4, 5, 6, 2, 1, 3, 3, 4]

(b) CSR Representation

Figure 2.1: Matrix representation format

appear in the matrix. That array is of length nnz, the number of non zero in the matrix. The `colA` also has nnz values and it indicates for each non zero, in which column of the matrix the non zero is located.

The `rowA` array is slightly more complicated, it has $n + 1$ values. `rowA[r]` indicates the first index of the first non-zero of row r and the last value indicate the total number of non zero in the matrix.

Computing $y = Ax$ is done by computing each dimension of y independently. For a particular row r, the algorithm extracts the non-zeros with indices between $rowA[r]$ and $rowA[r + 1]$. Each non-zero j is located in column $colA[j]$ and is of value $valA[j]$. And the algorithm executes $y[r] += x[colA[j]] * valA[j]$.

2.4.2 Coordinate (COO)

The COO format is a simpler format. The non zeros do not have to be listed in the order of the matrix, but they rather can be listed in any order. As such, the non zeros of a row do not have to be listed continously. To make this possible, the `rowA` array is not of size $n + 1$ but is of size nnz.

2.5 Sparse Matrix-Vector Multiplication(SpMV) on Distributed System

To perform sparse matrix-vector multiplication(*SpMV*) on the distributed system, the most common mechanism is to split the matrix into multiple parts and perform *SpMV* on each part individually in the different processors. In this work, we explore two different type of graph partitioning(Randomize 2D-Uniform Partitioning and 1D-Row Partitioning) and perform different SpMV algorithms based on the partition. A good partitioning can ensure better load balance and brings more freedom to each MPI process to work more independently.

2.6 Pagerank

Sometimes, it is important to generalize vertices of a graph based on the significance for a particular analysis. The aim of pagerank algorithm to sort out valuable vertices or ranking them based on the popularity on the graph. Pagerank is originaly used to sort the web search results [47]. Initially, all vertices receive the same rank(usually $\frac{1}{|V|}$, where $|V|$ represents the number of vertices.). Over the time, pagerank(PR) of every vertex v is updated using the following equation 2.2,

$$PR(v) \;=\; \frac{1-d}{|V|} \;+\; d \times \sum_{u \in N_{in}(v)} \frac{PR(u)}{|N_{out}(u)|} \tag{2.2}$$

The update of the pagerank is continue until it can not update the scores more than user defined threshold value.

2.7 Betweenness Centrality(BC)

There is another popular way to decide importance of a vertex is betweenness centrality. The algorithm generates unweighted all possible shortest path δ_{st} among two vertices s and t of a graph G. Now, if the $\delta_{st}(v)$ is the number of shortest path between s and t that pass through vertex v, then the betweenness centrality(BC) for a vertex v is measured by the following equations 2.3,

$$BC(v) \;=\; \sum_{s,t \in V, s \neq v \neq t} \frac{\delta_{st}(v)}{\delta_{st}} \tag{2.3}$$

CHAPTER 3

IMPACT OF AVX-512 INSTRUCTIONS ON GRAPH PARTITIONING PROBLEMS

Graph analysis now percolates society with applications ranging from advertising and transportation to medical research. The structure of graphs is becoming more complex every day while they are getting larger. The increasing size of graph networks has made many of the classical algorithms reasonably slow. Fortunately, CPU architectures have evolved to adjust to new and more complex problems in terms of core-level parallelism and vector-level parallelism (SIMD-level).

In this chapter, we are exploring how the modern vector architecture of CPUs can help with community detection, partitioning, and coloring kernels by studying three representatives algorithms. We consider the Intel SkylakeX and Cascade Lake architectures, which support gather and scatter instructions on 512-bit vectors.

The existing vectorized graph algorithms of classic graph problems, such as BFS and PageRank, do not apply well to community detection; we show the support of gather and scatter are necessary. In particular for the implementation of the reduce-scatter patterns. We evaluate the performances achieved on the two architectures and conclude that good hardware support for scatter instructions is necessary to fully leverage the vector processing for graph partitioning problems.

3.1 Introduction

We are particularly interested here in partitioning algorithms at large:coloring [48, 24, 49], clustering [50], partitioning [51], community detection [21, 2]. Recent interest in fast graph algorithms has met with a new look at how computer architectures can leverage. GPUs have been understandably popular because of the high flop rate, high memory bandwidth, and high power efficiency for graph problems [14]. CPU architectures have reacted by increasing core count but also by increasing SIMD width in a move to catch up in terms of performance and energy efficiency. In particular, modern Intel processors support AVX-512. These SIMD operations bring the expectation to provide higher energy efficiency than increasing the number of cores.

In this work, we consider the use of these new instructions to solve graph problems in the class

of partitioning. We pick three graph partitioning algorithms, namely a speculative parallel greedy algorithm for graph coloring, and the Louvain method for modularity optimization and Clustering using Label Propagation, as representative of graph partitioning algorithms. Section 3.2 describes these three problems. And we will study their performance on two different processors architecture; Intel Cascade Lake and SkylakeX.

With support for scatter operations, we designed, in Section 3.3, a strategy called ONPL, for One Neighbor Per Lane. Scatter operations enable us to write to the color of groups of neighbors at once. The operation in the Louvain Method adds some affinity values to the neighboring communities. Because the same community may appear multiple times, we call this operation a reduce-scatter, and we provide two implementations of this operation for different use cases.

Then, we show that the vectorization of these algorithms on x86-64 processors is impractical if they do not support scatter operations. Indeed the only feasible strategy in such a case is to use the different lanes of the vector to process different vertices at the same time. While this strategy applies to classic problems like BFS or SpMV, it requires reordering the graph so that no two vertices in a block of 16 vertices are neighbors for partitioning problems. This strategy only makes sense for the Louvain Method. The derived algorithm, presented in Section 3.4, is OVPL for One Vertex Per Lane.

A Part of this work [52] is published in scientific workshop. We extended our work by analysis performance of the Louvain method and Label propagation on the *RMAT* synthesis graph in section 3.6.3. It brings the perspective of what kind of graph structure well suited for the community detection problems. Section 5.5 presents the experimental settings, the code base used as baselines, and the set of graphs to be analyzed. Section 3.6 presents experimental results which show that ONPL can outperform the scalar implementation for graph coloring and label propagation for some networks. The Louvain Method is more computationally expensive. And using *ONPL* and *OVPL* in NetworKit leads to performance improvement on both architectures.

3.2 Graph Partitioning Problems

Graph partitioning problems are seen here as a large class of graph algorithms that encompass graph coloring algorithms [48, 24, 49], partitioning to minimize edge cuts [51], modularity optimizing

community detection algorithms [21, 2], overlapping community detection algorithms [53], label propagation, and certainly many others. All these algorithms have a similar structure in that each vertex is associated with a group of vertices (or multiple groups), and when considering the neighbors of a vertex, the group the neighbor belongs to is the key information rather than the neighbor itself.

We picked three classical partitioning algorithms to represent this class, namely Greedy Graph Coloring (for graph coloring), Louvain Method and Label Propagation (for non-overlapping community detection).

3.2.1 Speculative Parallel Greedy Graph Coloring

The distance-1 graph coloring algorithm assigns colors to the vertices of the graph so that no adjacent vertices have the same color. Minimize the number of colors is an NP-hard problem [23], and that is why various heuristic algorithms have proposed for the problem. In particular, a greedy algorithm can obtain near-optimal solutions [24]. The classic parallel algorithm for graph coloring is a speculative parallel greedy algorithm [25, 26] and presented in Algorithms 1, 2, 3.

Algorithm 1 Iterative Parallel Graph Coloring

Input: $G = (V, E)$

1: $C(v) \leftarrow 0$, for all $v \in V$
2: CONF $\leftarrow V$
3: **while** CONF $\neq 0$ **do**
4: ASSIGNCOLORS(G, C, CONF)
5: CONF $\leftarrow$ DETECTCONFLICTS(G, C, CONF)
6: **end while**
7: **return** C

Algorithm 1 represents an iterative parallel graph coloring. It takes a graph G with vertex set V and edge set E as an input. It first initializes the set of colors C for all vertices by 0 and a set of conflicts **CONF** by all vertices. It will iteratively color the vertices in **CONF** using a speculative greedy algorithm. And then check whether two neighboring vertices use the same color in which

Algorithm 2 AssignColors

Input: $G = (V, E), C$, CONF

1: Allocate private FORBIDDEN with size max degree
2: **for** $v \in$ CONF in parallel **do**
3: FORBIDDEN $\leftarrow false$
4: FORBIDDEN($C(u)$) $\leftarrow true$ for $u \in adj(v)$
5: $C(v) \leftarrow \min\{i > 0 | \text{FORBIDDEN}(i) = false\}$
6: **end for**
7: **return** C

Algorithm 3 DetectConflicts

Input: $G = (V, E), C$, CONF

1: NEWCONF $\leftarrow 0$
2: **for** $v \in$CONF in parallel **do**
3: **for** $u \in adj(v)$ **do**
4: **if** $C(u) = C(v)$ and $u < v$ **then**
5: ATOMIC NEWCONF $\leftarrow$ NEWCONF $\cup\, v$
6: **end if**
7: **end for**
8: **end for**
9: **return** NEWCONF

case they are in conflict and need to be colored again.

Algorithm 2 is the algorithm that will be vectorized and handles the assignment of the color to vertices. It takes graph G, a set of color C and a set of conflicts **CONF** as input. It traverses all the conflict vertices and finds out all the forbidden colors **FORBIDDEN** for the particular vertex. To do that it iterates all its neighbors and track down their colors. Line 4 of Algorithm 2 represents this operation. After collecting all the forbidden colors, it assigns to the vertex the first color that is not in the **FORBIDDEN** set.

Algorithm 3 detects conflicts that could arise during parallel speculative coloring. It takes a graph G, a set of color C, and a previous conflict set **CONF** as input. It defines a new empty conflict set **NEWCONF**. It considers all the previous conflict set of vertices in parallel and for each visits the neighbors to detect if the edge has both ends with the same color. In that case, one of the two vertices is added to the new conflict set atomically.

3.2.2 Parallel Louvain Method

The *modularity* is defined as the fraction of edges that fall within the partitions minus the expected fraction that would be within the partition if the edges are distributed randomly. This definition enables to greedily optimize modularity by considering moving a vertex to one of its neighbor community. Indeed, if a node $u \in C$ moves to the neighboring community D, then the modularity gain is $\Delta mod(u, C \rightarrow D) = \frac{\omega(u,D/\ \{u\}) - \omega(u,C/\ \{u\})}{\omega(E)} + \frac{(vol(C/\ \{u\}) - vol(D/\ \{u\})) * vol(u)}{2 * \omega(E)^2}$

The Louvain Method, first proposed by Blondel *et al.* [21], is one of the most popular methods to extract communities from a large network. It is a greedy multilevel algorithm that uses modularity as the objective function [22]. It alternates between two phases, the *Move Phase*, and the *Coarsening Phase*. In the *Move Phase*, nodes are repeatedly moved to adjacent communities to maximize modularity. This process repeats until the communities are stable. Then, the graph goes through a *Coarsening Phase* where each community collapse into a single vertex. The coarsened graph is then recursively processed with the same two phases. In that sense, the Louvain Method is representative of multi-level partitioning algorithms, such as [51].

The *Move Phase* (Algorithm 4) considers all the vertices in the network. For each vertex $u \in V$, for each neighbor $v \in N(u)$, it calculates the modularity difference between having u in its current community and moving it to the community of v. The decision of highest modularity gain is

Algorithm 4 Louvain Method: Move Phase

Input: graph $G = (V, E, \omega)$, communities $\zeta : V \to \mathbb{N}$
Result: communities $\zeta : V \to \mathbb{N}$

repeat
  for $u \in V$ **do**
    $\delta \leftarrow \max_{v \in N(u)}\{\Delta mod(u, \zeta(u) \to \zeta(v))\}$
    if $\delta > 0$ **then**
      $C \leftarrow \zeta(\arg\max_{v \in N(u)}\{\Delta mod(u, \zeta(u) \to \zeta(v))\})$
      $\zeta(u) \leftarrow C$
    end if
  end for
until ζ stable
return ζ

retained, and it is enacted if the modularity gain δ is positive. The algorithm repeats until no vertex changes community.

For each vertex u, the *move phase* is split into two parts. First, calculate the affinity(*measures of similarity between pairs of vertices*) of each neighboring community $\zeta(v)$ by adding edge weight $\omega(u, v)$ of the each neighbor v of u. Second, assign the node to the community of highest affinity.

The affinity calculation of a vertex is the computationally expensive part of the algorithm. It is the part that we vectorize in this chapter. We do not describe the *Coarsening Phase* since we will not make any changes to it. In this work, we only investigate the performances of the *Move Phase* of the *Louvain* method.

Many parallel methods exist to detect communities in massive networks. The most recent effort is included in NetworKit [54], GRAPPOLO [55] and studied in [22, 56]. GRAPPOLO uses a different and more complex algorithm than NetworKit. For simplicity, we present the Parallel Louvain Method (PLM), used by NetworKit.

PLM [22] is a shared-memory parallelization of the Louvain Method [21]. The algorithm performs the move phase in parallel by giving each thread different vertices to compute the affinity and their assignment to communities. It then coarsens the graph and recursively performs its optimizations. The runtime of PLM is mostly dictated by the first `move` phase; the process of converging the communities on the original graph, before any coarsening in done [22]. Trying to move vertices in parallel is not a race condition free process. Indeed, the algorithm may attempt to move two adjacent

vertices simultaneously. PLM is optimistic and assumes that only a few benign race conditions will happen in practice. However, race conditions may cause the process not to converge; PLM stops the move phase after 25 iterations, whether communities have converged or not.

In practice, Parallel Community Detection codes have limited multi-core scalability [22]; in particular because of the noted convergence issues. Since using multiple core reaps little benefit, this chapter focuses on using each core more efficiently by leveraging vector SIMD operations. And we consider improving multi-core scalability orthogonal to this work.

3.2.3 Parallel Label Propagation

Raghavan [2] *et. al* introduced a label propagation community detection method to extract community of a graph by the labeling of the vertices. First, all vertices are labeled as unique number, that means each node belong to their own singleton community. Then multiple iterations over the vertex set are performed: In each iteration, every vertex adopts the most frequent label in its neighborhood (breaking ties arbitrarily). Densely connected groups of vertices thus agree on a common label, and eventually a globally stable consensus is reached, which usually corresponds to a good solution for the network.

Algorithm 5 represents the label propagation method. From line 1 to 3, each vertex assign to a singleton community which is the label of that vertex. At line 4, a variable $updated$ is initialize by the number of vertices $n = |V|$. The main purpose of the $updated$ variable is to terminate the method when the total number vertices that change their community is lower than the threshold value θ. Active vertex set V_{active} is initialized at line 5. Line 6 to 18 represent the repeative section to update the community. At the beginning of each iterative process(line 7) $updated$ variable set to 0. From line 8 to 17, for each vertex u find the neighboring label l that maximize $\sum_{v \in N(u):\zeta(v)=l} \omega(u,v)$. If $l \neq \zeta(u)$ then label l assign to vertex u, and increment the variable $updated$ by 1 and mark all neighbors of u as active. The process terminates when $updated \leq \theta$

3.3 ONPL: One Neighbor Per Lane

The first strategy that we investigate, One Neighbor Per Lane (ONPL), uses the entire vector to process different neighbors of the same vertex.

Algorithm 5 Label Propagation

Input: graph $G = (V, E)$
Result: communities $\zeta : V \to \mathbb{N}$

for $u \in V$ **do**
    $\zeta(u) \leftarrow id(u)$
end for
$updated \leftarrow n$
$V_{active} \leftarrow V$
repeat
    $updated \leftarrow 0$
    for $u \in V_{active}$ and $deg(u) > 0$ **do**
        $l^* \leftarrow \arg\max_l \{\sum_{v \in N(u):\zeta(v)=l} \omega(u, v)\}$
        if $\zeta(u) \neq l^*$ **then**
            $zeta(u) \leftarrow l^*$
            $updated \leftarrow updated + 1$
            $V_{active} \leftarrow V_{active} \cup N(u)$
        else
            $V_{active} \leftarrow V_{active} n \{u\}$
        end if
    end for
until $updated > \theta$
return ζ

3.3.1 Speculative Greedy Graph Coloring

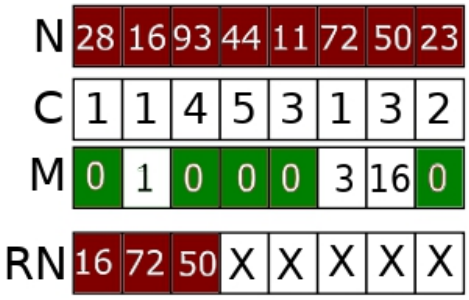
For graph coloring, the conflict detection method naturally vectorizes. Vectorization will be useful when marking which colors can not be used for a vertex. One can vectorize the loop that considers all the neighbors of a vertex. The operation boils done to loading 16 neighbors at a time with a `load` instruction; load the colors of these neighbors using a `gather` instruction. Then marking the used colors using `scatter` instruction. Identifying the first available color and identifying conflicting coloring vectorize naturally.

3.3.2 Louvain Method

Vectorized affinity calculation is complex because if two neighbors in the same community appear in the same vector, their contribution to the affinity of that community compounds. It will lead to conflicts during affinity calculation that requires resolving. We present the *One Neighbor Per Lane(*ONPL) vectorized Louvain method for community detection using intrinsic notations.

In `AVX-512`, the registers are 512 bits large so that it enables the ability to load 16 neighbors of a vertex at a time to process. Computing the affinity values requires a sequence of load, gather, addition, and scatter operations. Vectorized affinity will work well if all the vertices have their neighbors in different communities. Otherwise, blindly scattering causes some of the updates to be discarded, leading to incorrect affinity values. It requires summing the edge weights of every community before accessing the current affinity of adjacent communities. This operation is essentially a *reduce and scatter*. Unfortunately, no instruction directly does this operation. But the `AVX-512F` and `AVX-512CD` instruction sets enable two different ways to handle this. *ONPL* uses either one of them, depending on circumstances and these two instruction sets are sufficient to implement the algorithm.

Consider the extreme case where all the communities in the vector are different. It is typical at the beginning of the execution of the community detection code. In such a case, the addition and scattering can occur independently without requiring any reduction. If we know that all the lanes are independent, then no two lanes will write to the same location. Fortunately, the AVX-512CD instruction set provides `_mm512_conflict_epi32` instruction that tests each 32-bit element of an array `A` for equality with all other elements in `A` closer to the least significant bit. Each element's



(a) Conflict Detection.

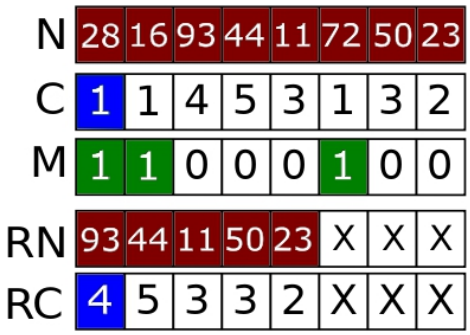
```
const   __m512i set0 = _mm512_set1_epi32(0x00000000);
/// Load at most 16 neighbor vertices.
__m512i N = _mm512_loadu_si512((__m512i *) &pnt_outEdges[i]);
/// Gather community of the neighbor vertices.
__m512i C = _mm512_mask_i32gather_epi32(set0, self_loop_mask, N, &zeta[0], 4);
/// Detect conflict of the community
__m512i C_conflict = _mm512_conflict_epi32(C);
/// Calculate mask M by comparing C_conflict with set0
const __mmask16 M = _mm512_mask_cmpeq_epi32_mask(self_loop_mask, C_conflict, set0);
```

(b) Code snippet to calculate mask M. *pnt_outEdges* represents the list of out edges, *self_loop_mask* is the mask to prevent the self-loop and *zeta* represents the list of community.

Figure 3.1: Perform reduce scatter using conflict detection. The neighbors (N) are in their Communities (C). A mask (M) is derived from C to denote the entries that will be processed (in green). Some neighbors will remain (RN) to be processed.

comparison forms a zero extended bit vector in `dst`. This instruction(`_mm512_conflict_epi32`) is the basis of the *conflict detection* method for reduce and scatter as it enables the extraction of different sets of communities and neighbors that can safely process at the same time. Figure 3.1a represents the process. Here, *N* is the list of neighbors of a vertex, and *C* is the corresponding list of communities. Instruction(`_mm512_conflict_epi32`) is applied on *C* to calculate the mask M. Figure 3.1b shows the code snippet to calculate the mask *M*. There are two techniques to handle the conflicted case: the first iteratively performs the vector operation on the non-conflicted sets and performs as many iterations of vector operations as there are non-conflicted sets; the second one applies vector operation on a non-conflicted set of neighbors only once and performs the remaining entries using purely scalar operations. Indeed, in practice, this conflict detection method uses many instructions. And it only useful if many communities can process at once. The vector will process one entry at a time with expensive vector operations if adjacent vertices belong to the same community. One can avoid the problem by performing vector operation only on the first set of independent communities and use the scalar operations afterward in the conflict detection method.

Another extreme case comes when all the communities in the vector are identical. This case arises when the process has mostly converged. In this case, an *in-vector reduction* is preferable. This method (sketched in Figure 3.2a) masks out all the entries of the vector besides the one mapping to a particular community. Figure 3.2b shows the code snippet to calculate the mask *M*. Then the edge weight mapping to this community is reduced with a masked reduction instruction `_mm512_mask_reduce_add_ps` and is finally added back to the affinity of that community.



(a) In-vector Reduction.

```
/// Load at most 16 neighbor vertices.
__m512i N = _mm512_loadu_si512((__m512i *) &pnt_outEdges[i]);
/// Gather community of the neighbor vertices.
__m512i C_vec = _mm512_mask_i32gather_epi32(set0, self_loop_mask, N, &zeta[0], 4);
/// Count the number of valid neighbors(u!=v)
sint vertex_cnt = _mm_popcnt_u32((unsigned)self_loop_mask);
/// It will compress communities to left that is not processed yet
C = _mm512_mask_compress_epi32(set0, self_loop_mask, C_vec);
/// Create a mask for Communities that is not processed
__mmask16 comm_mask = pow(2, vertex_cnt) - 1;
index * comm_not_processed = (index *)&C;
/// Handle the first community
__m512i first_comm = _mm512_set1_epi32(comm_not_processed[0]);
/// Calculate Mask for the first community
const __mmask16 M = _mm512_mask_cmpeq_epi32_mask(comm_mask, first_comm, C);
```

(b) Code snippet to calculate mask M. *pnt_outEdges* represents the list of out edges, *self_loop_mask* is the mask to prevent the self-loop and *zeta* represents the list of community.

Figure 3.2: Perform reduce scatter by compressing the communities. The neighbors (N) are in their Communities (C). A mask (M) is derived from C to denote the entries that will be processed (in green). Some neighbors(RN) and communities(RC) will remain to be processed.

In Figure 3.2a, *RN* represents the remaining vertices that are not processed yet, and *RC* is the list of their corresponding communities. Similar to the conflict detection method, there are two ways to proceed. Successive communities can use mask and reduce; however, this can lead to an issue for vertices that sit at the border of many communities causing potentially a large vector overhead. In practice, ONPL only processes vector operations for the first community of the vector and defaults to scalar implementation for the remaining communities.

The calculation of modularity from the affinity and the assignment of vertices to the community is done with simple vector processing and does not pose particular challenges.

3.3.3 ONLP: One Neighbor Per Lane Label Propagation

Nodes traverse in a parallel fashion, which brings the randomization on the node selection. For each node, it loads 16 neighbors and gathers their corresponding labels at once. For each distinct label, it sums the neighbor edge weight to create a vector with label weight. Each vector lane handles one neighbor of the vertex. Then an Intrinsic instruction $\_mm512\_reduce\_max\_ps$ applied to find out the heaviest neighbor label. A vertex participates in the next iteration if any of its neighbor labels change.

3.4 OVPL: One Vertex Per Lane

In the One Vertex Per Lane (OVPL) method, each SIMD lane processes different vertices. Initially, vertices of the graph are group into multiple blocks where the size of blocks is the multiple of the vector lanes. We have to restructure the network for the efficiency and convergence of the algorithm.

Because two vertices in a block will be processed simultaneously, OVPL requires two vertices in the same block not to be neighbors. Reordering the graph to have that property requires solving a graph coloring problem. Therefore it makes no sense to deploy OVPL for graph coloring. We only consider OVPL for the community detection problem.

3.4.1 Preprocessing

Vertices that are part of the same block will always be processed simultaneously. This property might induce race conditions that can prevent convergence. If the adjacent vertices are processed simultaneously, the affinity calculation performs on the changing information. The simplest case is a graph with two vertices that swaps their community infinitely, but the issue also appears on numerous complex networks.

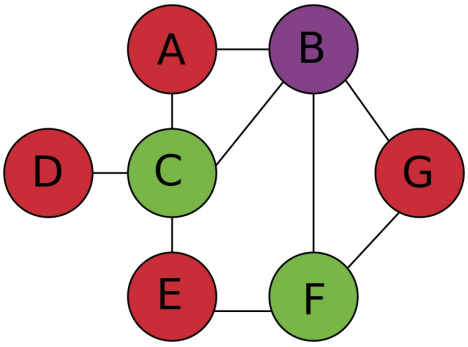
To prevent this from happening, we first solve a graph coloring problem: we allocate a color to each vertex so that no two adjacent vertices have the same color. We then group the vertices where each group holds vertices with the same color. That will make sure that no vertices are adjacent in a group. While finding the coloring with a minimal number of colors is an NP-Complete problem [23], we do not require such a high-quality solution. We use the speculative parallel greedy graph coloring algorithm [25] we described in Section 3.2.1.

After grouping the vertices, we sort the vertices in each group by non-increasing degrees. Sorting will help to minimize wasted computation during execution.

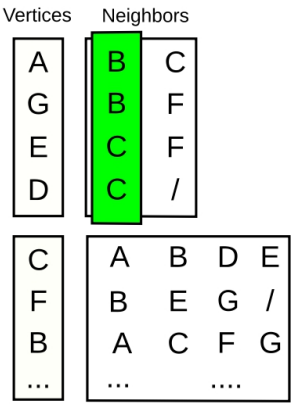
Finally, we split each group of non-adjacent vertices into small blocks of equal size equal to a multiple of the number of lanes. We reformat the vertices of each block to enable vectorization by interleaving the representation of the different vertices. That also reduces unaligned memory accesses. The format is similar to sliced ELLPACK [57]. A contiguous memory of size $max\_deg\_of\_block \times block\_size$ holds each block of vertices. The index from $(i-1) \times block\_size$ to $i \times block\_size$ will represent the i^{th} neighbor of the vertices of each block. Edge weights also follow a similar

representation.

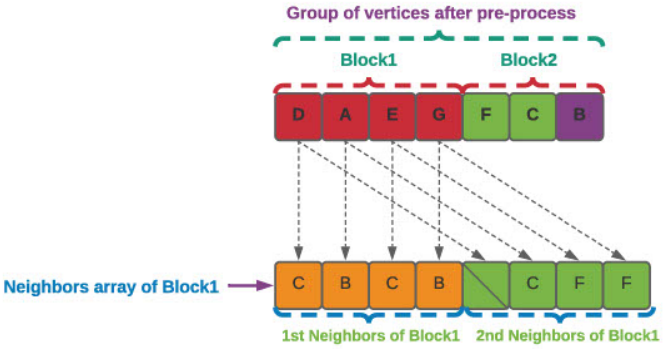
Figure 3.3 shows a sample graph and its block structure. In the example, we assume the vector length is 4 for readability (instead of 16). So, the initial block will hold vertices that are not adjacent by selecting the same color. But in the second group, there are no four vertices with the same color; that is why it contains vertices of different colors to fill the vector.



(a) Sample Graph.



(b) Abstract Memory Representation.



(c) Physical Memory Representation.

Figure 3.3: OVPL reorders the graph using a graph coloring methods and structure it in blocks of vertices so that the neighbors of the vertices of a block can be loaded in a vector (sketched in green) simultaneously.

3.4.2 Moving a Block of Vertices

Rather than moving a vertex to its most preferable community, OVPL *moves* a block of vertices at once. It calculates the *affinity* of all the vertices of a block concurrently. Therefore OVPL has a much higher memory utilization than PLM because it keeps $block\_size$ affinity structures in memory.

OVPL computes the affinity of each vertex of the block one neighbor at a time. OVPL first loads the first neighbor of each vertex of the block at once and gathers the community of the first neighbors. Then it gathers the affinity of the neighbor communities from the different affinity arrays. OVPL adds the edge weights to the obtained affinity and scatters the updated values back to the appropriate locations. Note that because of this, it was not possible to perform this vectorization on x86 processors before scatter was introduced with AVX-512.

This process repeats until all the neighbors of all the vertices of the block are processed, i.e., until the maximum degree of the block. However, some vertices may have a lower degree, so OVPL needs to check the existence of the neighbor. This check increases the number of instructions and causes the algorithm to use masked vector instructions. OVPL does not perform that check before the minimum degree of the block neighbors has been considered. The difference between the maximum and minimum degree in each block leads to wasted SIMD lanes. Preprocessing sorted the different color groups per degree to minimize the degree difference. Also representing the blocks by interleaving the vertices, enables access to the graph to aligned loads.

The assignment of vertices to new communities is done without particular optimization using a natural way of performing this task.

3.5 Experimental Settings

Hardware Platform and Operating System. We used two different machines for the two architectures we study in this chapter. We refer to the first machine as `SkylakeX`. It is a node with two Intel Xeon Gold 6154 processors (SkylakeX architecture, 18 cores per processor, no hyperthreading, 25MB L3 Cache) and 388 GB of DDR4 memory. The second machine is `Cascade Lake`, which is equipped with two Intel Xeon Gold model 6248R (Cascade Lake architecture, 24 cores per processor, no hyperthreading, 36MB L3 Cache) and 384GB GB of DDR4 memory. Both processors support Intel `AVX-512F` and `AVX-512CD` instruction sets with among others. Both machines use Linux `3.10.0`.

Software Environment. All the codes are compiled by the Intel C++ compiler `icpc` version `16.0.0.109`. Codes also compile with optimization flag `-O3` and `xCORE-AVX512` flags, so the compiler generates a binary optimized for the architecture. We pick existing established code bases

for both algorithms to confirm we start from implementations of reasonable good qualities.

We build graph coloring and community detection experiments on top of Kokkos [58] and *NetworKit* [54], respectively. We intended to compare to the original PLM implementation from [22]. During our experiments, we realized that PLM suffered from various memory management issues like large buffers were allocated and deallocated for each vertex traversed. We created a Modified PLM implementation (MPLM) that preallocates memory per thread. And then reuse the same buffer for the computation rather than deallocating and reallocating memory over and over. After confirming that MPLM is an improvement on PLM (See section 3.6.4), we will perform all other comparisons with MPLM.

Graphs. We perform our experiments on real-world data sets to avoid the bias introduced by random graph generator. We select graphs from the *Stanford Large Network Dataset Collection* (SNAP) [59] and *DIMACS* [60, 61] data sets that are well known for graph algorithm research. Graphs are from different categories like Social networks, clustering instances, sparse matrices, internet topology networks, citation networks. We expect that the coverage in the type of graphs enables deriving conclusions that are more general and bias-free than picking all graphs from a single category.

Table 3.1 presents the list of undirected graphs that we use in the experiments. The table also includes basic statistics such as the number of nodes (V), edges (E) of the graph, the maximum degree of the graph (Δ), and average degree (δ).

Collection of Result Sets. All the variants are run 25 times for each graph. The reported values of time and modularity are average of the 25 runs. For runtime, we only measure the time taken by the community detection(*Move-Phase*) and graph coloring algorithm itself, not the time spent reading the graph from the file system. We computed the 95% *confidence interval* [62] for the results of all the experiments. Once we realized the confidence intervals were very narrow and that the visible differences in the plots were statistically significant, we choose not to report them to improve figures readability.

Table 3.1: List of graphs used in the experiment

Graph	Nodes (V)	Edges (E)	Δ	δ
333SP	3,712,815	11,108,633	28	5
AS365	3,799,275	11,368,076	14	5
M6	3,501,776	10,501,936	10	5
NACA0015	1,039,183	3,114,818	10	5
NLR	4,163,763	12,487,976	20	5
Oregon-2	11,806	32,730	2,432	5
asia	11,950,757	12,711,603	9	2
belgium	1,441,295	1,549,970	10	2
delaunay_n24	16,777,216	50,331,601	26	5
europe	50,912,018	54,054,660	13	2
germany	11,548,845	12,369,181	13	2
in-2004	1,382,908	13,591,473	21,869	19
kkt_power	2,063,494	6,482,320	95	6
loc-Gowalla	196,591	950,327	14,730	9
luxembourg	114,599	119,666	6	2
netherlands	2,216,688	2,441,238	7	2
nlpkkt200	16,240,000	215,992,816	27	26
roadNet-PA	1,088,092	1,541,898	9	2
uk-2002	18,520,486	261,787,258	194,955	28

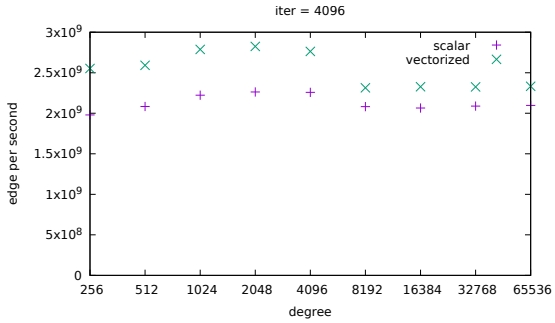


Figure 3.4: Microbenchmark performance on SkylakeX.

3.6 Performance Results

3.6.1 Microbenchmark

The microbenchmark simulates the affinity calculation of a single vertex in a fairly dense graph (with 4096 neighbors per-vertex packed along the diagonal). The code does a sequence similar to the operations of the algorithms we consider: load, gather, and scatter when running vectorially. The benchmark is written to compare a scalar implementation and a vector implementation.

The results for the SkylakeX architectures (in Figure 3.4) highlight that there are little differ-

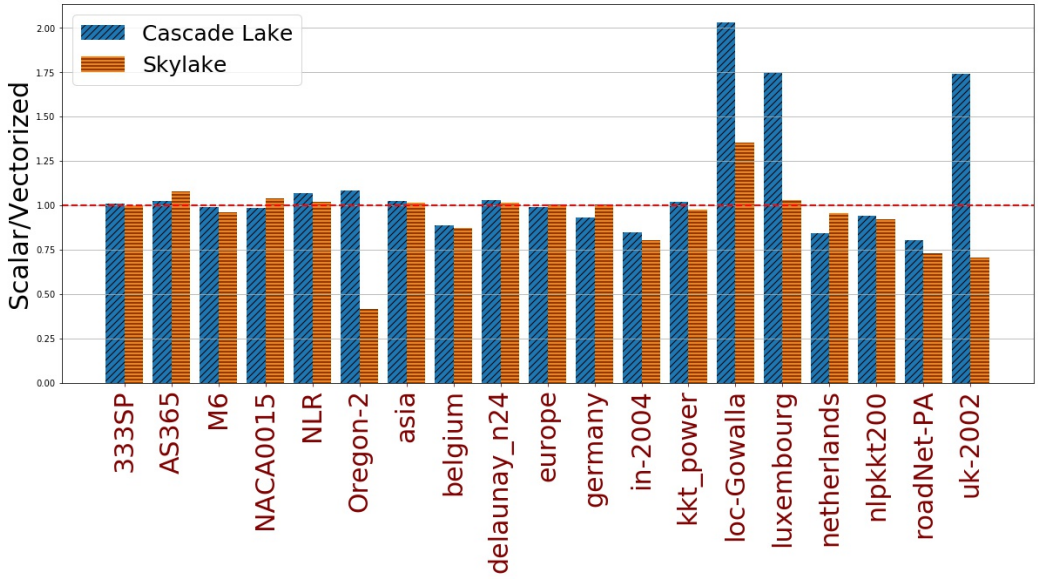


Figure 3.5: Impact of vectorization of Graph Coloring on both architectures. Y-axis represents the normalized version of the runtime comparison between scalar and vectorized. Scalar/Vectorized = 2.5 means vectorized version is 2 times faster than scalar.

ences in SkylakeX between the scalar and vectorized performance, with the vector implementation being 20% faster than the scalar one.

This sets the expectation for our problems. The microbenchmark is essentially what graph coloring does. For a graph with a large degree and the best diagonal layout, SkylakeX is only 20% faster using vector instructions than scalar ones. The community detection problem could see higher improvements because the problem is more computational.

3.6.2 Speculative Greedy Graph Coloring

The performance of the ONPL vectorization on graph coloring is displayed in Figure 3.5 for the *Cascade Lake* and *SkylakeX* architectures. Vectorized speculative graph coloring on both processors shows moderate performance enhancement for some graphs over the scalar version. Vectorized graph coloring on the Cascade Lake and SkylakeX outperform the scalar version by at most factors of 2 and 1.4. Speculative parallel graph coloring has two main parts. One is the assignment of color, and another is conflict detection. We only apply vectorization on the color assignment portion. Graph coloring has a limited opportunity for vectorization that is why it shows a moderate performance for most of the graphs.

3.6.3 Performance on R-MAT Graph

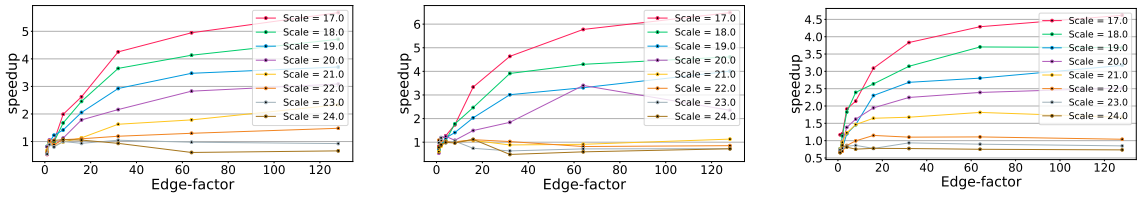
R-MAT Graph

R-MAT [63] is one of the most common synthesis graph generators which aims to maintain the power law of the natural graph. To generate a graph using R-MAT, one usually 6 attributes. First one is the *scale* which determine the number of nodes (2^{scale}) in the graph. Next attribute is the *edge-factor* which maintain the average degree of the graph. Then 4 parameters (a, b, c, d) to maintain probability distribution of edges among nodes. Usually adjacency matrix divided into 4 quad and each edge choose one quad based on the probability of that quad. R-MAT graph useful to describe results based on the structure of the graph. Here is the list of parameters we used to generate R-MAT graphs and to make it fair we perform different version of Label Propagation and Louvain method on the same graph. Table 3.2 represents parameters we used to generate R-MAT graph.

Table 3.2: R-MAT Parameters

Scale	Edge-factor	Probability Distribution
17, 18, 19, 20, 21, 22, 23, 24	1, 2, 4, 8, 16, 32, 64, 128	a=33%, b=33%, c=33%, and d=1%
		a=40%, b=30%, c=20%, and d=10%
		a=57%, b=19%, c=19%, and d=5%

Label Propagation



(a) RMAT parameters a=33%, b=33%, c=33%, and d=1%

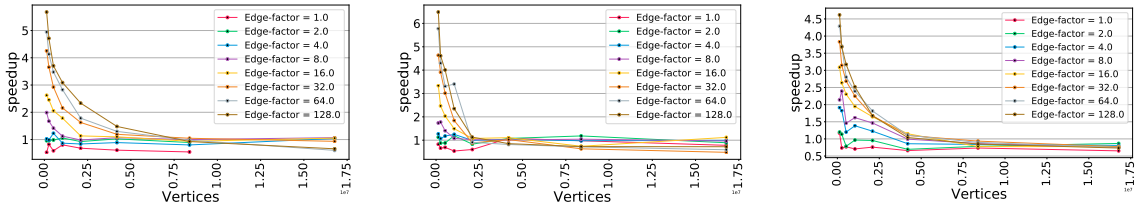
(b) RMAT parameters a=40%, b=30%, c=20%, and d=10%

(c) RMAT parameters a=57%, b=19%, c=19%, and d=5%

Figure 3.6: Performance gain of the ONPL Label propagation against scalar on the RMAT graph with different edge-factor on Cascade Lake processor.

Figure 3.6 and 3.7 shows the performance of the Label propagation on R-MAT graph on the Cascade Lake processor. We can see from Figure 3.6 that performance gain of the Label propagation increased with higher edge-factor. Now, ONPL perform vectorization on one neighbor per lane that

means higher edge-factor enable higher vectorization. We can also notice that performance of the application higher for the lower scale graph. Which gives the insight of the overall graph size has huge impact on the performance. Number of edges of a R-MAT calculated by $2^{scale} \times (2 \times edge - factor)$. Now, bigger graph brings higher cache misses because of the limitation of the memory size. So, if a graph shows higher average degree and size of the graph accommodate by the system memory then vectorized ONPL Label propagation will show a tremendous performance compare to scalar version. Figure 3.7 also provide similar evidence that smaller size (vertices) graph with higher edge-factor provide huge spike in performance gain.



(a) RMAT parameters a=33%, b=33%, c=33%, and d=1%

(b) RMAT parameters a=40%, b=30%, c=20%, and d=10%

(c) RMAT parameters a=57%, b=19%, c=19%, and d=5%

Figure 3.7: Performance gain of the ONPL Label propagation against scalar on the RMAT graph with different different number of vertices on Cascade Lake processor.

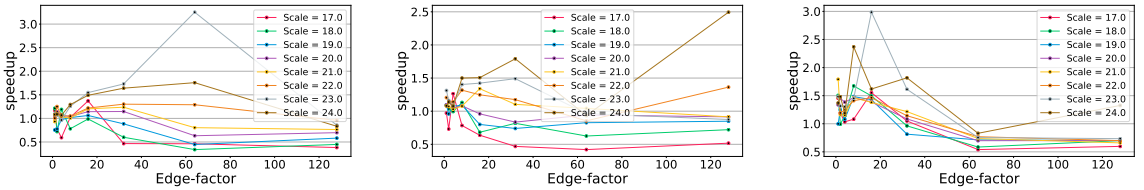
Louvain Method

Figure 3.8 and 3.9 show the performance of the Louvain method on the R-MAT graph. Louvain method shows the similar nature but the performance gain lower than the Label Propagation. One of the main reasons, the calculation of the Louvain method way much complex than Label Propagation. Memory usages is also higher in Louvain method which led more cache misses. But experiments on the R-MAT graph follow the main argument of the work that one should choose vectorized version ONPL Label propagation and Louvain for graphs with higher averages edges. If a graph shows lower average degree then scalar version is well suitable.

3.6.4 Louvain Method on NetworKit

Modified Parallel Louvain Method (MPLM)

We noticed some performance deficiencies in PLM, like threads reallocation of the memory needed for the affinity computation for each vertex that it encounters. To be able to study the impact of vec-

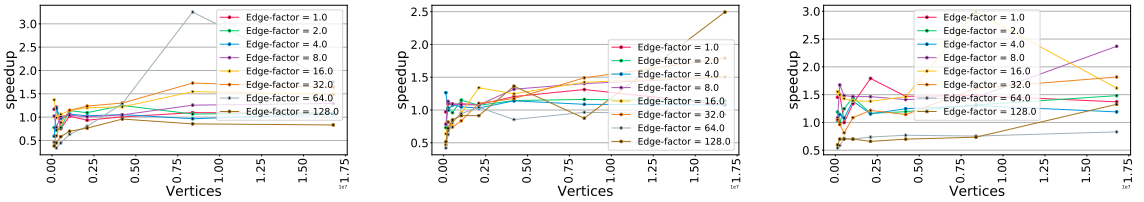


(a) RMAT parameters a=33%, b=33%, c=33%, and d=1%

(b) RMAT parameters a=40%, b=30%, c=20%, and d=10%

(c) RMAT parameters a=57%, b=19%, c=19%, and d=5%

Figure 3.8: Performance gain of the ONPL Louvain Method against scalar on the RMAT graph with different edge-factor on Cascade Lake processor.



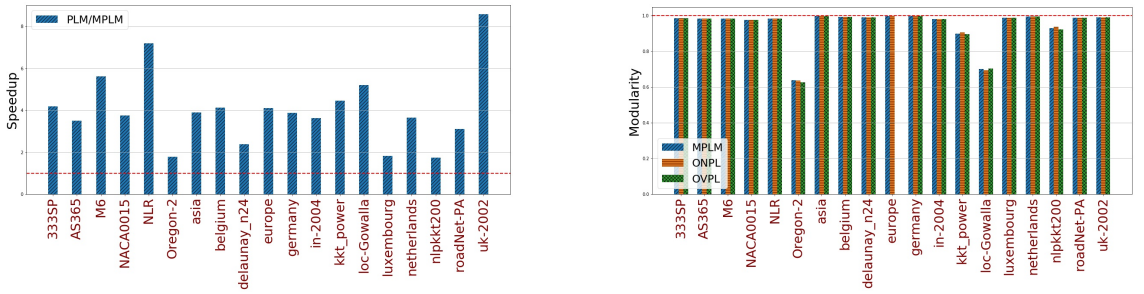
(a) RMAT parameters a=33%, b=33%, c=33%, and d=1%

(b) RMAT parameters a=40%, b=30%, c=20%, and d=10%

(c) RMAT parameters a=57%, b=19%, c=19%, and d=5%

Figure 3.9: Performance gain of the ONPL Louvain Method against scalar on the RMAT graph with different different number of vertices on Cascade Lake processor.

tor processing, we needed to make sure that the performance difference was rooted in vectorization rather than in memory management. The Modified Parallel Louvain Method (MPLM) is the code that contains various performance fixes for PLM.



(a) PLM vs MPLM speedup on the Cascade Lake(48 threads).

(b) Modularity of MPLM, ONPL, and OVPL on Cascade Lake(48 threads).

Figure 3.10: Performance and quality of the Modified PLM (MPLM) over PLM.

Figure 3.10a presents the improvement of MPLM compared to PLM for 48 threads on *Cascade Lake* for all studied graphs. Similar results observe on SkylakeX (not shown for brevity). We will use MPLM as the comparison point to see the impact of vector processing in community detection codes.

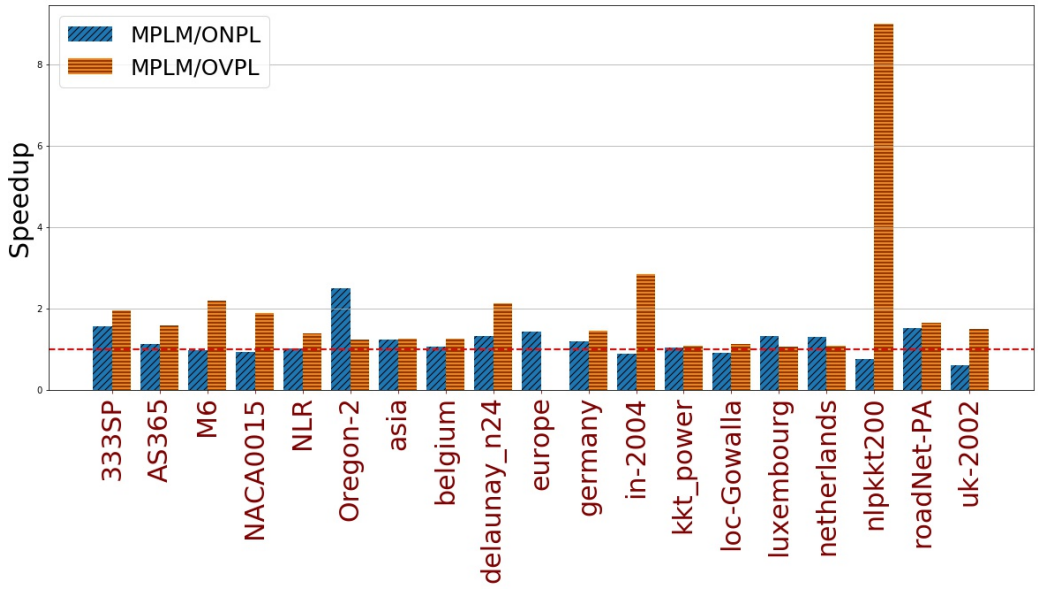


Figure 3.11: Speedup of ONPL and OVPL over MPLM on the Cascade Lake (48 threads)

Modularity

Since the algorithm has significant race conditions, any change of timings could affect the quality of the communities detected. Modularity is one of the standard metrics to evaluate the quality of the communities and is the metric optimized by MPLM. Figure 3.10b shows the modularity of the implementations of MPLM, ONPL, and OVPL on the *Cascade Lake* architecture using 48 threads. All methods achieve almost the same modularity which confirms the quality of the vectorized communities has not been significantly impacted.

ONPL

is a vectorized algorithm with the same memory consumption as the scalar algorithm MPLM and similar memory access patterns. Figure 3.11 shows the performance of ONPL compared to MPLM on the Cascade Lake for 48 threads: ONPL shows performance improvement for most of the selected graphs and at most a factor of 2.5 performance gain compared to *MPLM*. Figure 3.12 shows the *ONPL* performance in the NetworKit on the *SkylakeX* architecture. ONPL performs better than its scalar counterpart for almost all the graphs. The best performance of *ONPL* is recorded on the *SkylakeX* processor is around a factor of 1.8 compared to *MPLM*.

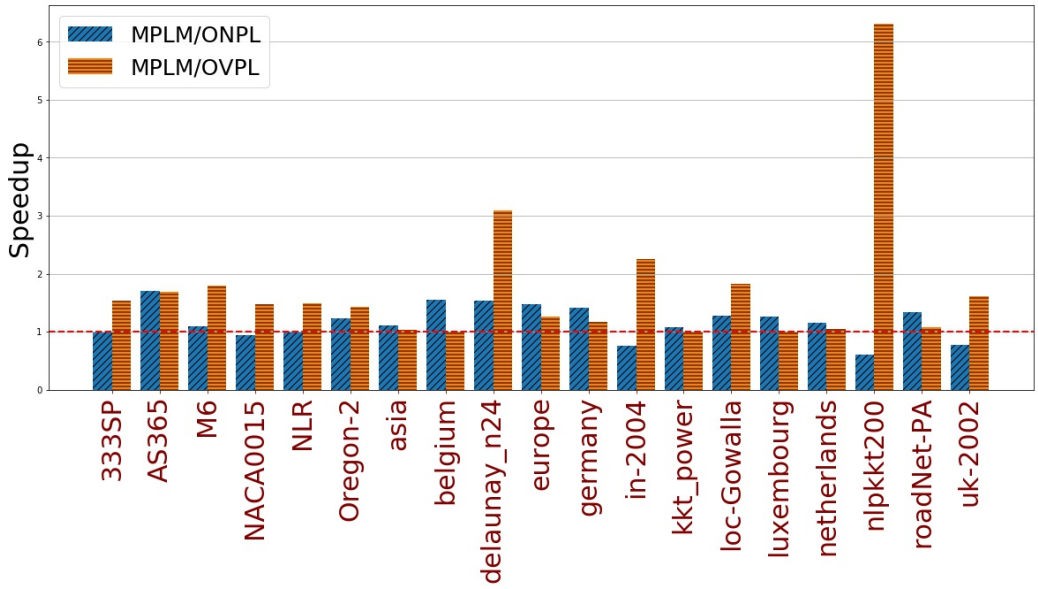


Figure 3.12: Speedup of ONPL and OVPL over MPLM on the SkyLakeX (36 threads)

OVPL

is an algorithm that consumes a lot more memory than the scalar algorithm due to having to store community affinity information for an entire block of vertices. Figure 3.11 presents the results of OVPL on the *Cascade Lake* architecture relative to the scalar implementation. For the graphs that were completed (some graphs ran out of memory), the performance derived is much better than the scalar implementation. Figure 3.12 shows the performance of OVPL on SkylakeX. We can see a factor of 9.0 and 6.5 performance gain for *OVPL* on the *Cascade Lake* and *SkylakeX* processors respectively compared to *MPLM*.

From the algorithm perspective, *OVPL* performs vectorization on a block of vertices, more specifically proper vectorization applies on the iteration only the minimum degree of vertices from the block. The rest of the iterations need more branching and also some lanes of the vectorization always remain unused. Our experimental results also reflect the scenario. Figure 3.13 shows only the performance of the selected graphs where most of the vertices have the same degree or very small variations. It shows a great performance gain. Graphs like `Delaunay`(average Degree 5) triangulations of random points or sparse matrix `nlpkkt`(average Degree 26) have most vertices with degrees close to the average. Every vertex in *OVPL*'s block is in sorted order and properly distributed by their degree, which also brings great load balancing.

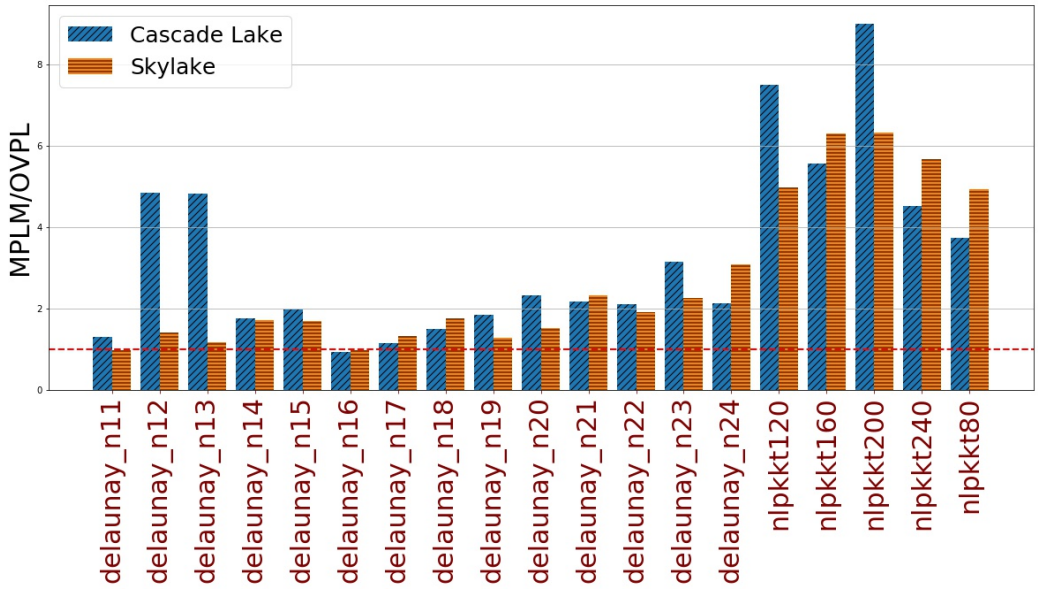
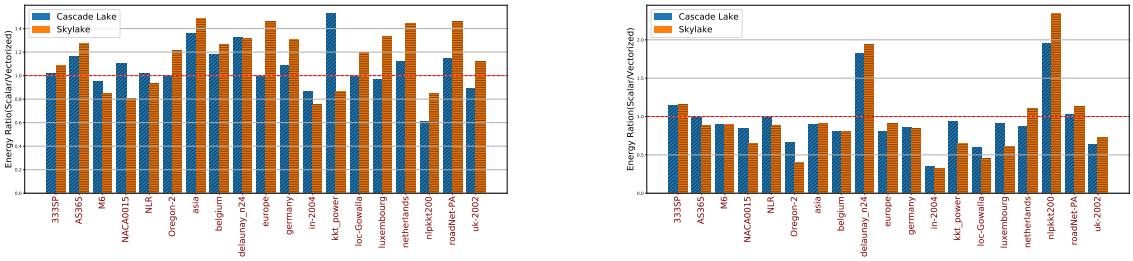


Figure 3.13: Speedup of *OVPL* over *MPLM* for the selected graphs where many vertices have degrees close to the average on both architectures.

Energy Consumption



(a) MPLM vs ONPL.

(b) MPLM vs OVPL.

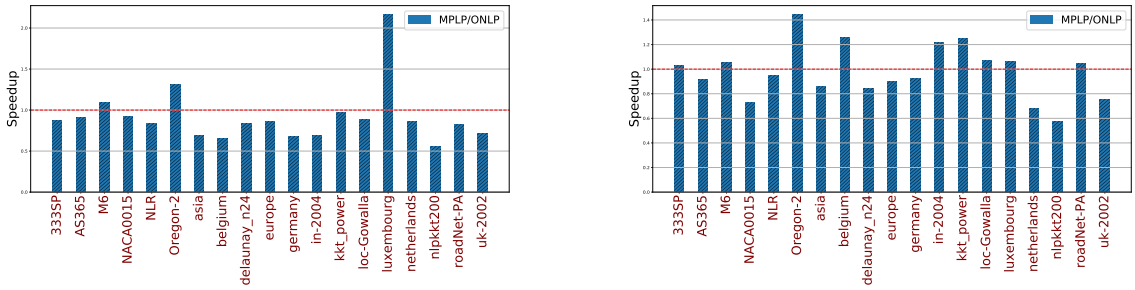
Figure 3.14: Performance and quality of the Modified PLM (MPLM) over PLM.

Figure 3.14 shows the overall energy consumption of different Louvain methods. The energy consumption by Louvain methods is calculated from RAPL (Running Average Power Limit) energy usages. Figures 3.14a and 3.14b show the energy consumption of ONPL and OVPL over MPLM. Any bar above 1 represents ONPL or OVPL consuming less energy than MPLM. Using SIMD operations helps to reduce the number of instructions in the execution. So, the expectation is that it can give better run time as well as better energy usage.

OVPL consumes more energy compared to ONPL and MPLM. Indeed, OVPL needs extra pre-processing, so it is adds work to enable vectorization. Also, because the vectorization pads the graph

representation to fit the vector lanes, there are cases where vector lanes are actively used to perform no useful computation, which raises energy consumption. Since OVPL adds work and wastes vector lane, it makes sense that it raises energy consumption.

ONPL shows decent energy efficiency for both architectures (Cascade Lake and Skylake). Most graph tested have a better energy consumption with ONPL than with MPLM. If we compare Figures **??** and 3.14, we can see that some graphs see better energy gains than speedup. For instance, *uk-2002* see a slowdown from ONPL but a factor of 1.2 of gain in energy efficiency. That means vectorization can help graph algorithms by not only making them faster but also energy efficient. We conjecture that while vector instruction consume more power, they decrease the number of instruction that need to be decoded which can translate in energy gains.



(a) On Cascade Lake Processor (48 threads).

(b) On SkyLakeX Processor (36 threads).

Figure 3.15: [Label Propagation] Speedup of vectorized Label Propagation (ONLP) over the parallel Label Propagation (MPLP).

Figure 3.15 shows the performance of label propagation(LP) on Cascade Lake and SkylakeX processors. The parallel and vectorized one neighbor per-lane label propagations represent by MPLP and ONLP. A couple of graphs get moderate performance gain for ONLP on Cascade Lake 3.15a processor; the highest performance gain is reported around 2.0 times over MPLP. Graphs on SkylakeX processor 3.15b also show moderate performance.

It is possible to vectorize the Label Propagation, but it has limited benefits. In the Louvain method, we vectorize the affinity calculation and modularity calculation code sections. Both of the code sections are required a good amount of instructions to assign vertices to their respective community. So while gather and scatter provide limited performance benefits, they enable the rest of the affinity and modularity calculation to be vectorized which improves performance. However, the vectorization of the Label Propagation does not lead to many more instructions to be vectorized.

3.7 Related Work

Label propagation is one of the most popular community detection algorithm proposed by Raghavan *et al.* [2]. The algorithm iteratively refines labeling of vertices to communities by finding for each vertex the label that most frequently appears in its neighborhood and migrating the vertex to that label. PLM [22] is the shared-memory parallelization of the Louvain Method [21] we use as a reference. Halappanavar *et al.* [56] presented community detection for static and dynamic networks using Grappolo.

Cheong *et al.* [64] proposed a parallel Louvain method for GPUs using three levels of parallelism for the single and multi-GPU architectures. Later, Naim and Manne *et al.* [65] proposed a highly scalable GPU algorithm for the Louvain method, which parallelizes the access to individual edges. There are other recent works like Sanders *et al.* [66] proposed Louvain method for the python; the main objective of their work is the simplicity to implement the algorithm in python language. Gheibi *et al.* [67] proposed a cache efficient Louvain method for Intel Knight Landing(KNL) and Haswell architecture.

Both GPUs and CPUs are SIMD systems, at least in spirit. Taking the analogy of a GPU warp as a core and a thread inside a warp as a lane, algorithms for GPUs can be re-envisioned as vectorized CPU algorithm. At a very high level, the distinction between vertex-based algorithms (such as OVPL) and edge-based algorithms (such as ONPL) appears in GPUs. However, there are still many differences between the architectures which cause engineering and algorithmic decisions for CPU and GPU systems very different.

3.8 Conclusion

We considered the impact of AVX-512 instructions on graph partitioning problems. We investigated, in particular, the *Cascade Lake* and the *SkylakeX* architectures and how to use them to perform speculative greedy graph coloring, the Louvain method, and Label Propagation.

Using different SIMD lanes for different vertices only makes sense for the Louvain Method as this vectorization requires a pre-processing overhead. The vectorization forces to process blocks of vertices with the same number of neighbors, which induces some work overhead. It proved to be particularly efficient for graphs with balanced degrees and high average degrees.

The vectorization strategy that processes multiple neighbors of a single vertex at once also shows performance improvement for many graphs. That strategy is only possible thanks to scatter instructions and other various new instructions in `AVX-512` that are critical to partitioning problems. The reduce and scatter pattern is critical in implementing these vectorizations.

Reduce-scatter as a concept can be implemented in multiple ways with vector instructions. We implemented both in our software environment by using intrinsic operations. In future works, we want to investigate compiler techniques to enable us to deploy these techniques on more graph partitioning kernels without requiring low-level programming expert.

CHAPTER 4

PERFORMANCE MODEL OF ITERATED SPMV FOR DISTRIBUTED SYSTEM.

Many applications rely on basic sparse linear algebra operations from numerical solvers to graph analysis algorithms. Yet, the performance of these operations is still not well understood. Users and practitioners rely on a rule of thumb understanding of what typically works best for some application domain.

This chapter aims at providing an overall framework to think about the performance of sparse applications for distributed systems. We use the sparse matrix-vector(SpMV) multiplication as the representative of the experiments. We model the performance of multiple SpMV implementations on the distributed system. As an end result, we represent a polynomial regression model for distributed systems that can provide the performance details of SpMV based on the structure of the matrix and system architecture.

4.1 Introduction

Sparse matrix-vector multiplication (SpMV) plays an important role in solving linear system. Performance of SpMV mainly depends on the size and structure of the matrices and the architecture of the system. The size of the matrices can be very large for scientific research: in high energy physics, the LHC project produces PBs of data; the climate science community relies upon access to the CMIP5 archive, which is several PBs in size; the multi-modal imagers used in biosciences can acquire 100GBs-TBs of data. That is why researchers like to go for parallel algorithms to perform SpMV. As a result, algorithms for distributed system become more popular for SpMV. The next important thing is the platform of the computer architectures.

To perform sparse matrix-vector multiplication (*SpMV*) on distributed systems, the most common mechanism is to partition the matrix into multiple parts and perform *SpMV* on each part individually in the different processors. Good partitioning can ensure better load balance and reduces the volume of communication between the underlying *MPI process*. Many partitioning algorithms have been proposed to ensure good load balance and to minimize *MPI communications* [68, 69, 70].

In this chapter, we explore two partitioning modes (Uniform 2D-Partitioning and 1D Row Partitioning) and the performance of the different *SpMV* representation based on these partitioning mechanisms on distributed systems. We develop a linear and a polynomial *support vector regression* (*SVR*) [18] performance models for SpMV operations on the distributed system for these different techniques. The models are accurate enough to predict the best configuration to execute SpMV given a matrix.

Performance modeling for any sparse matrix related algorithm is difficult. The structure of the sparse matrix varies in wide range. We first attempt to provide a linear model based on the size of the matrix. We find out if two matrices are same size in concept of rows then the performance of the SpMV shows little bit linearity against number of non-zero per rows. But it is still quite difficult to give accurate prediction. We build up a linear model that generate random matrices of different number of rows. For each matrix with a specific rows we change the number of non-zeros per row in the range $1, 2, 4, 8, 16, \ldots, 132$. Then build a linear regression model for particular row size matrix against the non-zeros per row. Our aim is to find out two near similar generated matrix A and B for given test matrix(M) that way the number rows in A and B immediate lower and higher than M respectively form the available matrices of the model. Then we predict the performance of the test matrix M based on this two model matrices. We describe in details in the below section 4.5.1.

Support vector regression (SVR) [18] is widely popular for performance modeling. In this paper, we present a polynomial SVR model to predict the performance of SpMV on distributed system. One of the main difficulties of the SVR model is to find out the right features for the model. Our proposed SVR model shows promising result that can be useful to model many other linear system as well. In section Section 4.5.2, we provide details of the polynomial SVR model.

4.2 Related Work

There have been lots of previous work have done on the *SpMV* performance model for the *CPU* and *GPU* architecture. In particular, Guo and Wang [71] *et.al.* have proposed a linear model for general-purpose *GPU* that can predict the runtime of the *SpMV* based on the strides size and nonzero per row. There are also some other *SpMV* models [72, 73] that exist for the *GPU* architectures. A performance model for *SpMV* on the *GPU* architectures is more common than a single or distributed

system of *CPU* architectures. Consistent and parallelisms of the *GPU* performance is the main reason behind all these models for the GPU.

4.3 Sparse Matrix Vector Multiplication (SpMV)

4.3.1 Distributed Memory Execution

To perform sparse matrix-vector multiplication (*SpMV*) on distributed systems, the most common mechanism is to split the matrix into multiple parts and perform *SpMV* on each part individually in the different processors. In this work, we explore two different type of graph partitioning and perform different SpMV algorithms based on the partition. A good partitioning can ensure better load balance and brings more freedom to each MPI process to work more independently.

Randomize 2D-Uniform Partitioning

In *2D-Uniform partitioning*, matrices are partitioned into both row-wise and column-wise. So, there is a choice to make the number of partitions for either way. In our experiments, we choose the same number of partitions for both way and that makes the perfect square number of partitions for the matrix. Each MPI process handles one of the portion of the matrix.

Now, load imbalance is one of the known issues for the scale-free graph partitioning. In particular, 2D-Uniform partitioning can balance the number of rows, but can have significant load imbalance in the non-zeros. This imbalance also impact on the load balance of the SpMV. Boman and Devine et al. mentioned in their graph partitioning work [74] that randomization can bring great load balance for SpMV for the 2D-Uniform partitioning. Each row (and corresponding vector entry) is assigned to a random process. Since the expected number of rows and non-zeros is uniform for all processes, this method generally achieves good load balance.

1D-Row Partitioning

In our *1D-Row partitioning*, matrices are only partitioned row-wise. So, it is like the row-wise K-way graph partitioning. We can define the ID-Row(K-way) partitioning for a graph $G = (V, E)$ with $|V| = n$, partition V into k subsets, $v_1, v_2, \ldots, v_k$ such that $v_i \cap v_j = \phi$ for $i \neq j$, $|v_i| = n/k$, and $\cup_i v_i = V$, and the numberof edges of E whose incident vertices belong to different subsets

is minimized. A K-way partition of V is commonly represented by a partition vector P of length n, such that for every vertex $v \in V$, $P[v]$ is an integer between 1 and k, indicating the partition at which vertex v belongs.

A K-way partition of the graph or matrix can be used to assign k tasks to k processors. So, we buildup our MPI performance model based on this partitioning, we choose METIS [69] K-way partitioning method. METIS graph partitioning is well known for the proper load balance for the sparse matrix [75].

4.4 Sparse Matrix-Vector Multiplication Algorithms

4.4.1 SpMV on the 2D-Uniform Partitioning

In here, we describe the mechanism to perform *SpMV* on the 2D-Uniform partitioning matrix. In our experiments, we always used perfect square($4,\ 9,\ 16,\ 36, \ldots,\ 225,\ 256\ etc.$) number of MPI processes for the experiment. So, if p^2 is the number of processor then a matrix need to row-wise split into p parts and then column wise p parts. That means a matrix will split into in total p^2 partition. Another important fact is that each processor will contains the same number $\lceil \frac{matrix\_size}{p} \rceil$ of rows and columns. If any partition contains less number of rows or column than $\lceil \frac{matrix\_size}{p} \rceil$ then we added extra dummy rows or columns with all zero elements to make it similar to the others. Figure 4.1 represents the mechanism of the 2D-Uniform partitioning on a system of 9 processors and 9×9 matrix. Then, the matrix splits row-wise $\sqrt{9} = 3$ times and column-wise also 3 times. Now, matrix has 9 rows and 9 columns that means each process will handle 3×3 sub-matrix. Figure 4.1 shows 9 different region by 9 different colors. Processes that contains the same rows are called same row rank processes and who contains the same columns are called same column rank processes.

To perform the *SpMV*, every processor needs the corresponding vector elements. From the Figure 4.1, we can see sub-matrices $1, 4, 7$ require the same vector elements. In our experiment, only diagonal sub-matrix will hold the corresponding column vector elements and beginning of the process it will share the vector elements with all the column rank processors by using MPI collective method `MPI_Bcast`. After perform the multiplication, all row rank processors need to reduce their values. The resulted reduce value is saved onto the corresponding diagonal sub-matrix by using

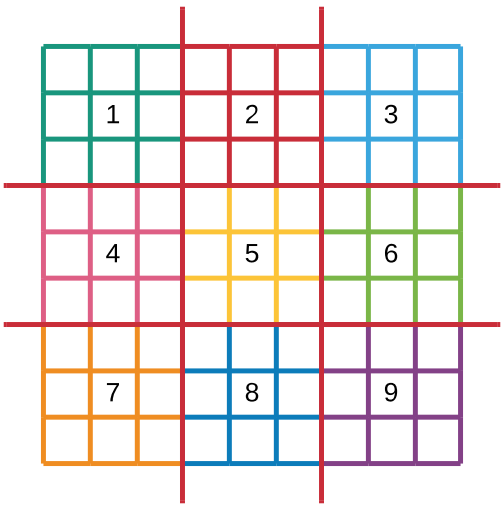


Figure 4.1: 2D-Uniform partitioning a symmetric matrix for 9 distributed processors.

MPI collective `MPI_Reduce` with `MPI_SUM` operation.

4.4.2 SpMV on the 1D-Row Partitioning

In 1D-Row partitioning, matrices are splitted in row-wise only, so if you assign a single part of the matrix to a MPI process then it can need any vector elements data to perform SpMV on the part of the matrix. So, there is a choice to make, either a MPI process can hold full vector information or only hold the vector portion corresponding to the rows. Based on this criteria, we perform two different SpMV algorithms on the 1D-Row partition.

Global 1D-Row SpMV(G1DR-SpMV)

In this algorithm, all the MPI processes hold the entire vector information, they perform matrix multiplication locally on the part of the matrix that belongs to them. After matrix multiplication, an `ALL_Gatherv` takes place to share the updated value of the vector. Because it performs the all-to-all MPI collective operation, so this algorithm is bounded by the MPI communication.

Local 1D-Row SpMV(L1DR-SpMV)

In this algorithm, all the MPI processes only keep a portion of the vector corresponding to the rows of their part of the matrix. Initially they perform local matrix multiplication on the non zero

elements whose column belongs to the local vector. For the rest of the non zero elements whose vector elements belongs to the other process, they request to the processes to send the information. So, it requires `MPI_Send` and `MPI_Recv` to transfer data among the process. But the good thing processes communicate independently each other and METIS K-way partition can give a better partition which reduces the inter-process communication.

4.5 Performance Model

4.5.1 Linear Model

We applied the linear model only for 2D-Uniform partitioning SpMV, because it only suitable for the model. In the model, we picked perfect square(p^2) number of process. We know from the 2D-Uniform partition that each matrix splits into ($p \times p$) parts that means each row of the partition contain p parts and each part handle by each process. In our model, every process receives a matrix with same size and same number of nonzero per row. But the distribution of the nonzero per row follows random distribution. Column-wise processors broadcast vector information and row wise processors reduce the results. Each process independently performs matrix-vector multiplication. We record average time of these processors. Figure 4.2 shows the performance of the model. It shows the linearity of the performance over nonzero per row for a particular matrix. It gives us the idea to build up a regression model that can predict run time of SpMV based on the nonzero per row. We train the model with different size random matrix. In the training data, we used 7 different variation of matrix($1, 2, 4, 8, 16, 32, 64$ nonzero per row). In real data, the matrix size can vary a wide range. So, it is not feasible to train all possible size matrix. Let assume the a subject matrix with *r* average row per process and *npr* nonzero per row that we need to predict the SpMV performance for the matrix. Now, it is not necessary that our system train with the matrix which has the same size as r_i. Our system will find out two row equation for row r_1 and r_2, where $r_1 < r < r_2$. It is important to note that, r_1 is the max row equation available in the model that is lower than r and r_2 is the minimum row equation that is greater than r.

$$y_1 = m_1 \times x + c_1 \qquad \text{for } r_1 \text{ and } x = npr$$

$$y_2 = m_2 \times x + c_2 \qquad \text{for } r_2 \text{ and } x = npr$$

Now, according to our model we expect the the performance(y) of the subject matrix with row r is between y_1 and y_2 ($y_1 \leq y \leq y_2$). Our system finally predict the execution run time of the subject matrix using following equation,

$$y = y_1 + \frac{(y_2 - y_1) \times (r - r_1)}{r_2 - r_1}$$

Figure 4.3 shows the example for the subject matrix with 250000 average row per process and 32

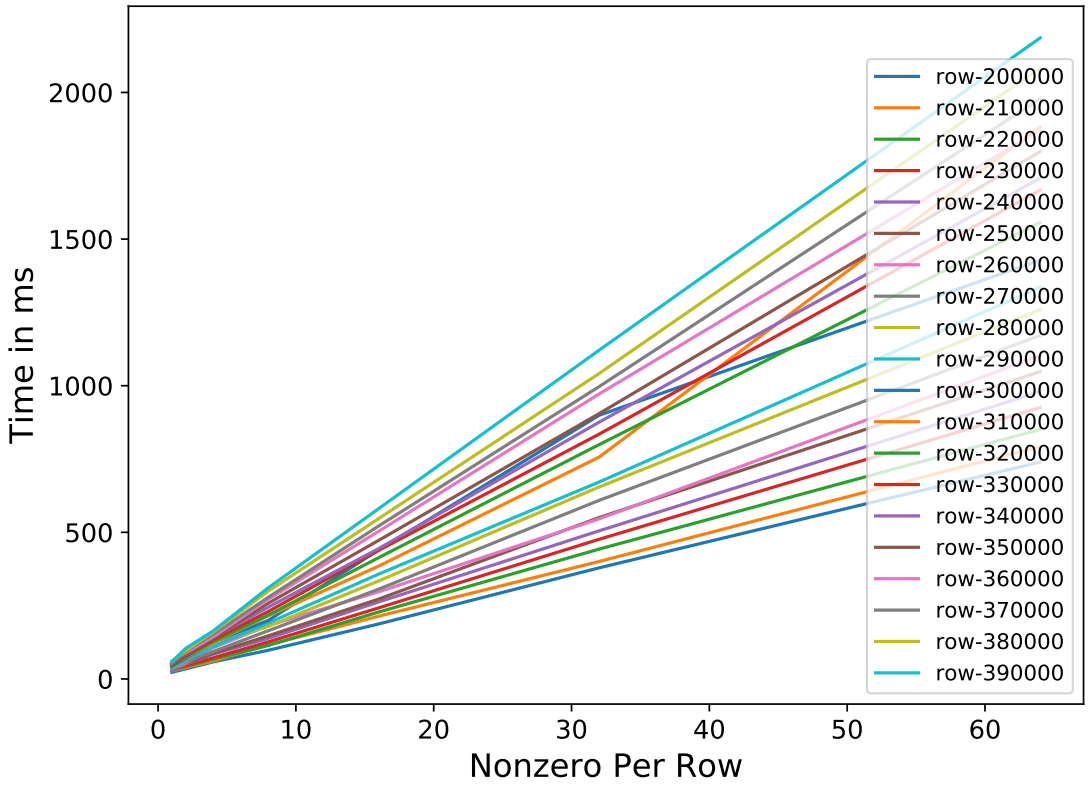


Figure 4.2: Performance model corresponding nonzero per row for a particular partition row.

nonzero per row. Here, possible r_1 and r_2 equations are available for rows 200000 and 310000. Figure reflect the prediction mechanism of our system.

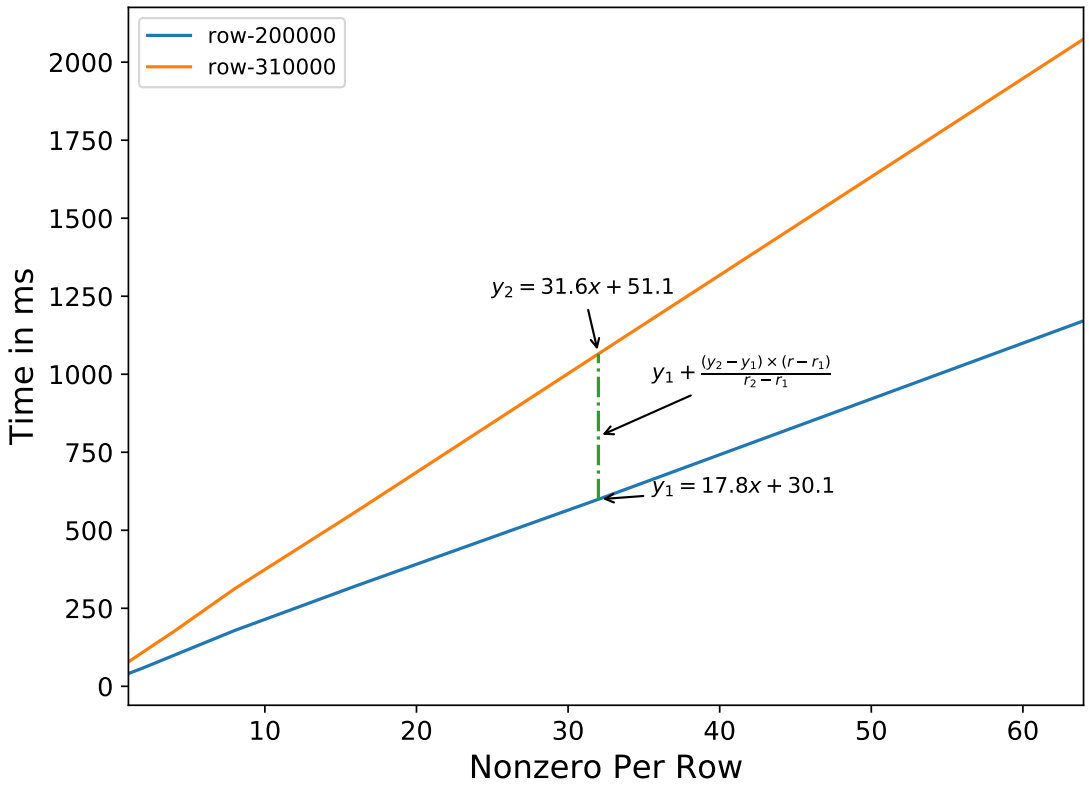


Figure 4.3: Predict SpMV Performance for a matrix that has avg nonzero per row 32 and avg row per process 250000.

4.5.2 Polynomial Support Vector Regression (SVR) Model

Support vector regression (SVR) is a variation of support vector machines (SVM), which solves the following problem for a given training vectors $x_i \in R^p, i = 1, \ldots, n$, and a target vector $y \in R^n$,

$$\min_{\omega, b\zeta, \zeta^*} \frac{1}{2}\omega^T\omega + C\sum_{i=1}^{n}(\zeta_i + \zeta_i^*)$$

$$\text{subject to } y_i - \omega^T\phi(x_i) - b \leq \varepsilon + \zeta_i,$$

$$\omega^T\phi(x_i) + b - y_i \leq \varepsilon + \zeta_i^*,$$

$$\zeta_i, \zeta_i^* \geq 0, i = 1, \ldots, n$$

According to [76], the dual problem for $\varepsilon - SVR$ under given regularization parameter C with given kernel K is

$$\begin{aligned}
&\min_{\alpha,\alpha^*} \tfrac{1}{2}(\alpha - \alpha^*)^T Q(\alpha - \alpha^*) + \varepsilon e^T(\alpha + \alpha^*) - y^T(\alpha - \alpha^*) \\
&\text{subject to } e^T(\alpha - \alpha^*) = 0 \\
&0 \le \alpha_i, \alpha_i^* \le C, i = 1, \ldots, n
\end{aligned}$$

Where e is the vector with all ones and $C > 0$ is the upper bound. Q is and positive semi-definite matrix, $Q_{ij} = K(x_i, x_j) = \phi(x_i)^T \phi(x_j)$ is the kernel. Here training vectors are implicitly mapped into a higher (maybe infinite) dimensional space by the function ϕ.

The decision function is:

$$\sum_{i=0}^{n} (\alpha_i, \alpha_i^*) K(x_i, x) + \rho$$

We applied polynomial SVR model for all three algorithms 2D-Uniform partitioning SpMV, G1DR-SpMV and L1DR-SpMV. One of the main task for the SVR model is to find out the appropriate attributes for the model and then find right values for the free parameters. The most common technique is to find the proper values of the free variables is the Cross-Validation and Grid-Search.

Cross-Validation and Grid-Search

Initially we split the matrices into train and test data set that test data can not participate in the training mechanism. Here, we choose polynomial("poly") kernel to perform all the machine learning regression. In this kernel, there are three variables that need to be selected (C, ε, γ). There are no fixed value for these variables, based on the application criteria it can vary. The most technique to choose the optimal value for these variables are to apply cross-validation in the grid-search. In our experiment, we set γ as "auto", so we only need to find the optimal C and ε. We select $5 - fold$ cross-validation that means we split the training set into 5 different parts, and among 5 parts we choose 4 as a training set and the remaining one as a test set. We pick different sets as a training and record the score. To do this experiment, we need to set the free variable. In the grid search, one need to select a set of variables for the free variable(C, γ), like $C = \{2^{-2}, 2^{-1}, \ldots, 2^4, 2^5\}$ and $\gamma = \{0.01, 0.02, \ldots, 0.2, 0.3\}$. Now, for each pair of variable we need to perform cross-validation and record the score. We need to select the pair that gives the best

performance in cross-validation. The next step is to train the model using best variable on the whole train set. So, our model is now ready to predict the performance. We test the performance on the test data set.

Feature selection

We build up polynomial support vector regression (SVR) models for three different SpMV algorithms on the two different graph formats (CSR, COO). The SVR model follows the same mechanism for all multiplication algorithm. The model predicts the average run time for a particular test matrix based on its attributes. So, attributes are the key to a good performance model. Although the local multiplication is the same for every algorithm, the communications are different for the different partitioning modes. The common attributes for all three algorithms are:

1. Average rows per process.
2. Average non zero per process.
3. Average non-zero per row.
4. Density of the matrix.
5. Standard deviation of the non zero per row.

But the communication among the MPI processes in the Local L1DR-SpMV (one-to-one) is different than the other two (one-to-all or all-to-all). That is why based on the sparsity of the matrices the communication can vary significantly. For that, we need the following extra attributes for the L1DR-SpMV which can be extracted after partitioning.

1. Average local non-zero elements.
2. Average global non-zero elements.
3. Average inter-process call.
4. Average data transfer.

4.5.3 SpMV Model from Micro-Benchmark

In this model, we predict the runtime of the SPMV based the performance of couple of micro-benchmark of a selected computer architecture. Initially, SpMV on a distributed system can be divided into two main parts,

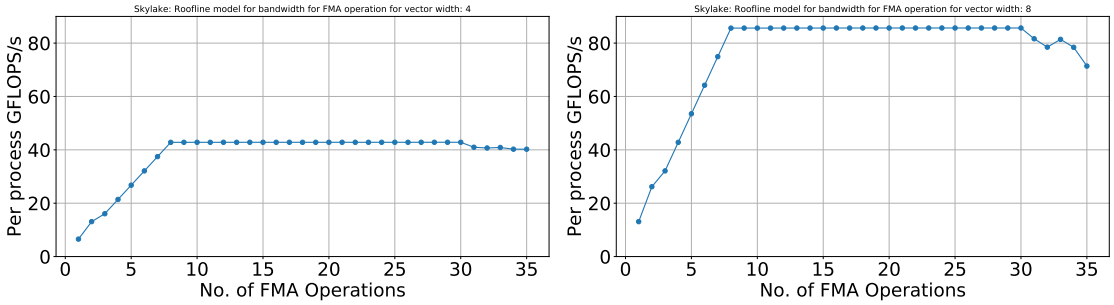
- Core Calculation: performance of the matrix-vector calculation in a MPI node.
- MPI Communication: communication runtime among MPI ranks(processes).

Instruction Cost

We can represent the basic SpMV by $y = y + Val * x$, which requires two floating point operation (multiply and addition). So, we can say we need to calculate NNZ (number of non-zeros) times FMA (fused multiply addition) to perform SpMV. But, first it needs to load the data and SpMV is bound by the memory bandwidth rather than instruction. So, we separated the *core calculation* into,

- Run time for FMA: $L_{FMA} \times NNZ$, where L_{FMA} is the latency of a single FMA and NNZ is number of non-zeros.
- Memory access latency: L_{RW}

Micro Benchmark for *FMA* The machines we will use are based on the SkylakeX architecture. The throughput of SkylakeX is 2 instructions per cycle and the latency of FMA is 4 cycles. So there is a potential of pipelining ($2 \times 4 = 8$) to get the optimal results. Now, SkylakeX has 512-bit register that can give the ability of the vectorization. For 64-bits floating point operation it can give vector width 8 and for 32-bits it can give at max vector width 16. To find the peak performance of the FMA and to avoid the read-write latency we need to setup the benchmark that datasets can be contained in the register. Now, SkylakeX has 32 registers. We can populate the pipeline by using sufficient amount of work. By varying the number of fused-multiply-addition calculation, we can find out the performance limitation. From the information of the processors, we can say that 4 cycles required for *FMA* and the throughput of the FMA is 2 instruction per cycle, that means we should at least use $4 \times 2 = 8$ instruction at a time to populate the pipeline.



(a) Vector Width: 4 (b) Vector Width: 8

Figure 4.4: Skylake: (MPI)Roofline model for bandwidth for FMA operation

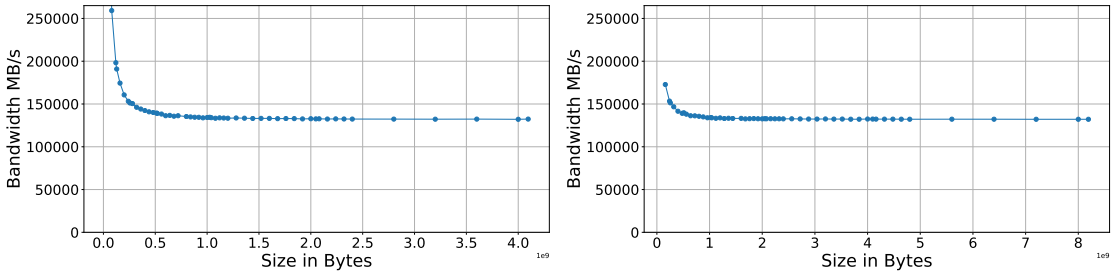
Figure 4.4 shows the roofline model of the *FMA* on SkylakeX processor. We can estimate the theoritical peak performance a single FMA by the following equation,

$$P = Base\_Clock\_Frequency \times Vector\_Width \times \frac{FLOPs}{Instruction}$$

The significance of the *FMA* latency is actually negligible compare to the memory access latency.

Memory Accesses Cost

Micro-Benchmark for Memory Access We use the *STREAM* [77] benchmark to find out the cost the memory accesses for a selected architecture.



(a) Single Precision. (b) Double Precision.

Figure 4.5: Single and double precision memory access bandwidth on Skylake processor.

Figure 4.5 shows the relation between data size and bandwidth of the SkylakeX processor. To pick the right bandwidth for a matrix-vector multiplication, we first calculate the size of the data. Based on the size of the data, we pick the average of the available immediate lower and higher point

of the benchmark. Note that in STREAM, all memory accesses are sequential.

Cache Access Patterns In SpMV some of the arrays are traverse sequentially and some have irregular pattern. Table 4.1 shows the memory access property for different matrix representation format. We can see the access to the vector `x` is irregular for both CSR and COO format. But access to the vector `y` is only irregular for the COO format.

One could model all irregular accesses as random accesses. However we know that most graphs actually exhbit significant locality [78]. We model the memory accesses by computing a cache friendliness metric which represent the fraction of access that are in cache and the fraction that are in memory.

This cache friendliness is computed for a matrix as follows. We assume the matrix is traversed sequentially and we model the access to the cache assuming the cache has the granularity of a cache line and that the cache replacement policy is LRU. And we assume that the cache is of the size of the L3 cache divided by the number of core on the processor. This model does not accurately capture many properties of the memory subsystem (such as associativity of caches, or cache sharing across multiple cores, the fact that there are multiple levels of caches, and concurrent processes); but it is a simple to compute estimation of how irregular the memory access actually are.

All the sequential data accesses and *hits* of irregular accesses are treated as *Sequential* data accesses. The cost of these data accesses is accounted based on the predicted bandwidth to the core out of the STREAM benchmark. And all cache *miss* in irregular accesses are treated to `Random` data accesses. The cost of these operations are accounted based on the latency of the memory access by the cores. The cost of these sequential and random access are summed.

Note that this model does not capture all the complexity of a modern system: for instance, a mix of latency bound and bandwidth bound memory operations can overlap especially when multiple cores access memory at the same time. But while one could certainly craft an example where the model is very inaccurate, we do not believe these extreme case would happen in practice.

Micro-Benchmark for MPI communication

MPI communications mostly depend on the size of the message and network topology. We build a benchmark based on the OSU-MPI-Benchmark. Because of the dynamics of the network topology

Table 4.1: Memory Access Property for 2D-Partitioning SpMV Model(RPP=rows per process, NNZ=non-zero elements).

Array	#Accesses		Data Type	Access Type	
	CSR	COO		CSR	COO
rowA	2×RPP	NNZ	Integer	Sequential	Sequential
colA	NNZ	NNZ	Integer	Sequential	Sequential
valA	NNZ	NNZ	Floating	Sequential	Sequential
x	NNZ	NNZ	Floating	Irregular	Irregular
y	RPP	NNZ	Floating	Sequential	Irregular

we build a polynomial model using an SVR based on the number of nodes, number of MPI ranks, message size, and MPI communication type.

Putting it together

Figure 4.6 shows the overall structure of the SPMV model from micro-benchmark. The three components, instructions, memory, and MPI communication are added together.

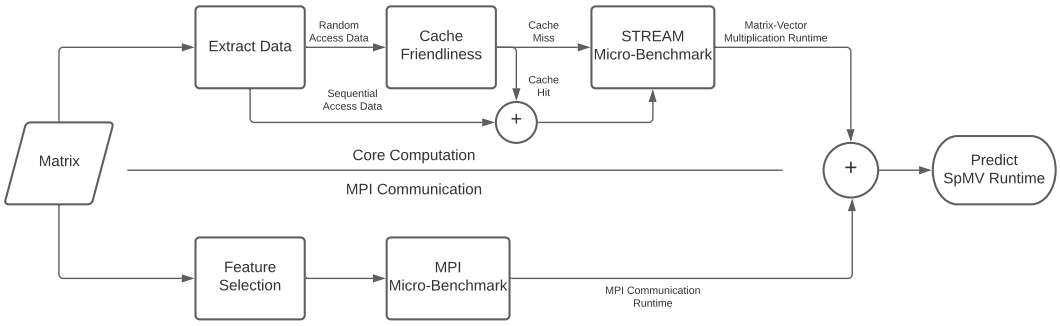


Figure 4.6: Structure of the SpMV model from micro-benchmark.

4.6 Experimental Settings

4.6.1 Hardware Platform and Operating System

All the nodes of the computing cluster come with Intel Xeon Gold 6154 processors (SkylakeX architecture) and 388GB of DDR4 memory. Each node has 36 cores in 2 sockets. Hyper-threading is disabled. The base frequency of each processor is 3.00 GHz. Each processor has 25 MB of L3 Cache. The nodes are connected by EDR Infiniband. The machine uses Linux `3.10.0`. To present

concise results all experiments are performed on 144, 169, 225 and 256 MPI processes allocated on 4, 5, 7 and 8 nodes respectively. The system support a maximum of 256 MPI processes.

4.6.2 Matrices

All the matrices we use come from the SuiteSparse Matrix Collection (previously known as the *Florida Sparse Matrix* collection) [79]. We used seven matrices for our tests. Their properties are described in Table 4.2.

Because the linear and SVR models require to be trained based on timings from real runs, we used an other 83 matrices in order to train these two models.

Table 4.2: Properties of the test matrices.

Name	Rows	Columns	Nonzero Elements	Avg Degree
333SP	3,712,815	3,712,815	22,217,266	6.0
AS365	3,799,275	3,799,275	22,736,152	6.0
M6	3,501,776	3,501,776	21,003,872	6.0
NLR	4,163,763	4,163,763	24,975,952	6.0
hugetrace-00010	12,057,441	12,057,441	36,164,358	3.0
road_central	14,081,816	14,081,816	33,866,826	2.4
road_usa	23,947,347	23,947,347	57,708,624	2.5

We also generated three larger matrices using the R-MAT model [63]. R-MAT can generate graphs with *power-law* degree distributions and small-world characteristics. We generated the R-MAT matrices using parameters $a = 0.33$, $b = 0.33$ and $c = 0.33$. In the name of the matrix the first numerical value in the name represent the total number of edges and the second value represent the number of rows.

4.6.3 Metrics

We will resent two types of results, runtimes on particular matrices and number of MPI processes for a particular execution mode of SpMV. We will also present relative errors for models calculated by:

$$error = \frac{|actual\ time - predicted\ time|}{actual\ time} \times 100$$

4.7 Experimental Results

4.7.1 Runtime of SpMV

Table 4.3 present the results of the different way of executing SpMV. At first, we want to look at the different 1D partitioning results. We can see local 1D-row SpMV models perform better than global 1D-row versions. In local 1D-row model, processes maintain local vector and communicate with other processes independently. On the other hand, global 1D-row models maintain global vector and use collective communications. So, global 1D-row models transfer same amount of data in the communication for a specific matrix size, it does not mater how sparse it is. On the other side, local-1D-row requests data when they require from other process; which depends on the particular non-zero elements of the matrix. Other than that, matrices in the testing set are mostly well partition-able. That means that once partitioned each node will not need many external values of the x vector to perform the multiplication. Because of that, the local variant of 1D partitioning which does custom messages is particularly effective because it minimizes the total amount of value exchange communications.

While local 1D partitioning performs better than the global execution for all the matrices we picked and are listed table 4.3, it is because the global communication of global 1D shares data that is not used. Indeed the matrices of Table 4.3 are well partition-able. But we expect that matrices that can not be well partitioned would not see such poor performance when using the global 1D method. The performance of Global1D execution on R-MAT matrices, which can not be well partitioned, (Table 4.4) is about the same as local 1D.

Similar reasons explains the performance of 2D partitioning methods. The performance of SpMV on R-MAT is better on 2D uniform partitioning than using 1D partitioning. Indeed the matrix is not well partitionable, so METIS partitioning can not minimize the communication of a 1D partitioning and suffers from load imbalance. On the other hand, 2D partitioning will balance the load thanks to its random row and column permutation. And the 2D decomposition of the matrix provides more efficient communication patterns.

On the matrices of Table 4.3, the random permutation of rows and columns breaks all locality in the matrix and cause communications that were not necessary in the 1D local execution.

4.7.2 Accuracy of performance models

Table 4.5 presents the relative error in prediction made by the different models. The linear model performs erratically on most of the testing data set. And we only present the data for the 2D partitioned methods, the model often exhibits an error higher than 15%. The linear model can not capture the behavior of the methods based on 1D partitioning and is often predicts an order of magnitude away from the actual runtime (error not shown).

The SVR model captures pretty well the performance of 2D partitioning methods with errors of prediction below 15% on most instances and usually below 10%. 2D partitioning with uniform permutation of rows and columns provides the regularity in the dataset that SVRs can easily capture. The geometric mean error on 2D partitioning is below 5% on both matrix representation.

Though the SVR model has moderate accuracy on 1D partitioning model with a geometric mean error of 9%. This stems from the fact that the features of the SVR model do not capture well the quality of the partitioning and the complexity of the communication patterns.

The model based off micro benchmark overall performs the best. The method using 1D partitioning with custom messages, Local 1D, is fairly well modeled. All the instances are modeled within a 15% of error and often within 10%. The geometric mean error are 5.3% and 3.7% on CSR and COO matrix representations. The model is accurate because the execution of SpMV is carefully modeled and the communication pattern are also known to the model.

The 2D partitioning execution are also well modeled with error usually below 10% and a geometric mean error of 3% and 6%. The error on the Global 1D execution is much higher (geometric mean of 9% and 22%). This is due to the micro benchmark based model to not accurately predict all gather collective. This MPI collective operation is fairly hard to model accurately as the underlying communication algorithms can configure very differently way depending on the system state [80]. The 2D partitioning also uses collective but in smaller communicators which makes them easier to model.

The accuracy on R-MAT matrices are given in Table 4.6. The SVR model provides good accuracy in its prediction of the R-MAT SpMV performance with geometric means ranging from 3% to 9%. The accuracy of the microbenchmark based model does not perform as well, ranging from 9% to 17% and reaching 50% on the COO L1DR.

Table 4.3: Actual Run Time of All SpMV execution.

Matrices	Nodes	Prcs	Actual Time(s)					
			CSR 2DU	COO 2DU	CSR G1DR	COO G1DR	CSR L1DR	COO L1DR
AS365	4	144	7.4E-03	5.5E-03	2.7E-02	3.3E-02	1.3E-03	1.4E-03
	5	169	5.8E-03	4.3E-03	2.1E-02	2.7E-02	1.1E-03	1.2E-03
	7	225	4.7E-03	3.6E-03	2.0E-02	2.5E-02	9.2E-04	9.3E-04
	8	256	4.4E-03	3.6E-03	2.4E-02	2.8E-02	8.5E-04	8.4E-04
road_central	4	144	2.5E-02	1.9E-02	9.6E-02	1.1E-01	2.6E-03	2.5E-03
	5	169	2.2E-02	1.5E-02	7.1E-02	7.1E-02	2.4E-03	2.1E-03
	7	225	1.9E-02	1.3E-02	7.4E-02	8.0E-02	1.9E-03	1.7E-03
	8	256	1.5E-02	1.2E-02	8.6E-02	9.1E-02	1.8E-03	1.6E-03
NLR	4	144	8.2E-03	6.1E-03	2.9E-02	3.6E-02	1.4E-03	1.4E-03
	5	169	6.5E-03	4.8E-03	2.2E-02	3.0E-02	1.3E-03	1.3E-03
	7	225	5.4E-03	4.0E-03	2.1E-02	2.7E-02	9.9E-04	1.0E-03
	8	256	4.8E-03	3.9E-03	2.6E-02	3.0E-02	9.5E-04	9.7E-04
hugetrace-00010	4	144	2.3E-02	1.7E-02	8.2E-02	9.3E-02	2.5E-03	2.6E-03
	5	169	2.0E-02	1.3E-02	6.2E-02	6.7E-02	2.2E-03	2.2E-03
	7	225	1.7E-02	1.1E-02	6.4E-02	7.0E-02	1.8E-03	1.8E-03
	8	256	1.4E-02	1.0E-02	7.3E-02	7.9E-02	1.6E-03	1.6E-03
333SP	4	144	7.2E-03	5.4E-03	2.6E-02	3.3E-02	1.2E-03	1.2E-03
	5	169	5.7E-03	4.2E-03	2.0E-02	2.7E-02	1.1E-03	1.1E-03
	7	225	4.7E-03	3.5E-03	1.9E-02	2.4E-02	8.4E-04	8.9E-04
	8	256	4.2E-03	3.4E-03	2.4E-02	2.7E-02	8.2E-04	8.4E-04
M6	4	144	6.8E-03	5.0E-03	2.5E-02	3.1E-02	1.3E-03	1.3E-03
	5	169	5.3E-03	3.9E-03	1.9E-02	2.5E-02	1.1E-03	1.1E-03
	7	225	4.4E-03	3.4E-03	1.8E-02	2.3E-02	8.7E-04	9.2E-04
	8	256	4.0E-03	3.3E-03	2.2E-02	2.6E-02	8.3E-04	8.2E-04
road_usa	4	144	3.6E-02	2.9E-02	1.6E-01	1.8E-01	4.1E-03	3.8E-03
	5	169	3.6E-02	3.2E-02	9.9E-02	1.5E-01	3.4E-03	3.2E-03
	7	225	3.1E-02	2.3E-02	9.8E-02	1.4E-01	2.8E-03	2.6E-03
	8	256	2.7E-02	2.0E-02	1.5E-01	1.5E-01	2.6E-03	2.4E-03

The average error of the linear model for the R-MAT data is around 32%. The main reason behind this error is the density of the matrices. As we can see All R-MAT matrices are highly dense compare to the real-world test matrices. When the non-zero per row increases, the angle between two candidate rows for the model is becoming very large. That leads to a higher error of the model, which means the linear model is more suitable for the highly sparse matrices.

4.8 Discussion

The micro-benchmarking based model and the SVR model are accurate enough to predict the correct configuration of the system for a particular matrix in most cases. When an error occurs, the loss of performance for not picking the right model is usually under 10%.

We believe there is more potential in the fine modeling of performance of the micro-benchmark based model over the SVR model. The SVR model is fairly expensive to train. It requires to build a different model of the problem for each hardware configuration. The trained model is different

Table 4.4: Actual Run Time of All SpMV execution on R-MAT graph.

Matrices	Nodes	Prcs	Actual Time(s)					
			CSR 2DU	COO 2DU	CSR G1DR	COO G1DR	CSR L1DR	COO L1DR
rmat_620M2M	4	144	3.8E-02	4.9E-02	2.3E-01	6.3E-01	1.6E-01	5.4E-01
	5	169	3.1E-02	4.5E-02	2.4E-01	5.3E-01	1.0E-01	5.6E-01
	7	225	1.9E-02	2.8E-02	1.8E-01	2.6E-01	1.3E-01	6.6E-01
	8	256	2.0E-02	1.6E-02	1.9E-01	2.4E-01	1.9E-01	8.3E-01
rmat_680M3M	4	144	5.8E-02	6.7E-02	2.0E-01	9.9E-01	2.0E-01	3.3E-01
	5	169	4.2E-02	6.0E-02	2.0E-01	7.8E-01	1.3E-01	2.9E-01
	7	225	3.3E-02	4.4E-02	1.5E-01	4.3E-01	1.3E-01	2.9E-01
	8	256	3.0E-02	2.7E-02	1.5E-01	3.5E-01	1.5E-01	3.5E-01
rmat_690M2M	4	144	3.9E-02	4.7E-02	2.7E-01	6.3E-01	1.9E-01	6.0E-01
	5	169	3.1E-02	4.3E-02	2.2E-01	4.3E-01	1.3E-01	6.7E-01
	7	225	2.1E-02	3.2E-02	1.9E-01	2.9E-01	1.6E-01	6.6E-01
	8	256	1.6E-02	1.9E-02	1.7E-01	2.1E-01	1.4E-01	7.0E-01

Table 4.5: All SpMV prediction model performance.

Matrices	Nodes	Prcs	Error													
			Benchmark Model						SVR Model						Linear Model	
			CSR 2DU	COO 2DU	CSR G1DR	COO G1DR	CSR L1DR	COO L1DR	CSR 2DU	COO 2DU	CSR G1DR	COO G1DR	CSR L1DR	COO L1DR	CSR 2DU	COO 2DU
AS365	4	144	1.2%	4.0%	19.2%	34.4%	3.9%	1.2%	0.8%	5.9%	14.0%	8.0%	20.2%	19.1%	23.1%	13.4%
	5	169	9.8%	19.7%	2.7%	26.1%	3.4%	1.9%	16.9%	9.3%	0.9%	1.0%	19.7%	17.7%	19.9%	17.1%
	7	225	8.3%	13.6%	4.8%	24.3%	7.5%	1.0%	20.9%	6.4%	4.9%	6.7%	23.2%	20.0%	16.0%	2.7%
	8	256	1.4%	0.1%	21.2%	31.0%	8.5%	1.1%	13.5%	1.9%	20.9%	16.8%	22.8%	18.1%	20.0%	4.0%
road_central	4	144	7.4%	5.6%	20.2%	27.1%	10.5%	14.3%	8.9%	12.3%	14.8%	6.4%	3.0%	6.0%	11.9%	17.6%
	5	169	3.5%	12.4%	0.6%	1.7%	9.0%	14.3%	1.4%	3.5%	4.5%	24.9%	1.9%	4.5%	3.8%	30.3%
	7	225	9.4%	8.7%	10.3%	16.6%	6.6%	11.4%	5.8%	0.1%	11.8%	6.3%	9.2%	0.6%	17.8%	5.2%
	8	256	2.4%	8.1%	23.4%	27.9%	6.0%	8.4%	4.6%	1.4%	26.3%	21.9%	11.0%	3.7%	4.0%	11.0%
NLR	4	144	0.1%	4.9%	19.1%	34.5%	2.2%	3.7%	2.0%	6.4%	14.0%	8.0%	18.2%	15.0%	14.0%	19.4%
	5	169	11.4%	19.6%	2.3%	26.1%	5.4%	0.1%	15.5%	7.5%	1.3%	0.9%	21.2%	18.9%	18.4%	0.5%
	7	225	5.6%	15.2%	4.1%	24.1%	6.0%	0.6%	14.7%	6.2%	4.3%	6.6%	22.1%	19.8%	14.0%	29.2%
	8	256	3.9%	2.8%	21.3%	31.1%	10.4%	6.4%	13.0%	1.0%	21.3%	17.1%	24.6%	22.8%	14.4%	2.6%
hugetrace-00010	4	144	8.2%	4.2%	20.0%	28.6%	8.2%	9.0%	12.3%	13.3%	14.6%	6.8%	0.6%	3.1%	12.3%	24.4%
	5	169	6.7%	13.7%	0.7%	7.6%	10.8%	11.6%	7.2%	2.0%	3.2%	15.3%	0.4%	2.4%	0.1%	13.8%
	7	225	10.1%	9.5%	10.2%	18.3%	7.8%	9.1%	8.7%	1.5%	11.5%	6.8%	6.0%	6.3%	5.6%	5.7%
	8	256	3.4%	9.2%	22.7%	28.2%	6.4%	7.0%	3.4%	0.4%	25.3%	21.0%	8.0%	7.7%	3.2%	7.8%
333SP	4	144	1.7%	3.7%	19.2%	34.4%	2.3%	2.9%	0.7%	5.8%	14.1%	8.0%	15.8%	15.5%	18.0%	2.3%
	5	169	9.2%	19.6%	2.2%	26.0%	0.2%	0.4%	17.1%	9.7%	1.4%	1.0%	17.6%	18.7%	20.1%	10.9%
	7	225	3.9%	15.5%	5.6%	23.9%	3.8%	1.8%	16.8%	8.7%	5.6%	6.1%	20.3%	20.9%	18.7%	0.8%
	8	256	1.5%	4.3%	21.2%	30.9%	10.7%	6.2%	14.4%	2.8%	20.8%	16.5%	24.6%	22.6%	13.0%	3.8%
M6	4	144	2.8%	3.5%	19.2%	35.2%	4.9%	0.4%	0.0%	5.0%	14.1%	9.1%	20.7%	18.8%	19.9%	12.5%
	5	169	9.3%	18.7%	3.2%	26.1%	2.9%	1.4%	19.5%	10.1%	0.3%	1.0%	19.0%	20.4%	19.9%	8.6%
	7	225	4.7%	9.7%	4.5%	23.9%	8.8%	6.2%	19.9%	4.4%	4.5%	6.1%	23.9%	23.8%	19.4%	3.9%
	8	256	0.3%	0.8%	21.1%	30.9%	12.0%	5.2%	15.3%	1.0%	20.4%	16.3%	25.1%	21.1%	12.0%	6.3%
road_usa	4	144	9.0%	3.7%	19.3%	26.1%	1.3%	12.7%	7.4%	1.5%	14.6%	5.6%	3.9%	7.1%	6.1%	5.3%
	5	169	0.1%	12.6%	23.8%	16.0%	6.1%	17.4%	1.8%	16.1%	27.8%	2.8%	5.8%	9.4%	24.9%	18.3%
	7	225	3.1%	5.4%	16.5%	20.1%	6.8%	15.3%	0.0%	0.3%	13.7%	10.6%	3.3%	3.9%	13.4%	17.2%
	8	256	1.7%	7.6%	23.0%	27.4%	9.7%	14.6%	2.8%	2.9%	26.7%	22.0%	4.2%	2.1%	8.7%	28.2%
Geometric Mean			3.1	6.4	9.2	22.8	5.3	3.7	4.5	3.1	8.5	7.1	9.6	9.7	10.8	7.7

Table 4.6: Performance of the SVR SpMV model on RMAT matrices.

Matrices	Nodes	Prcs	Error													
			Benchmark Model						SVR Model						Linear Model	
			CSR 2DU	COO 2DU	CSR G1DR	COO G1DR	CSR L1DR	COO L1DR	CSR 2DU	COO 2DU	CSR G1DR	COO G1DR	CSR L1DR	COO L1DR	CSR 2DU	COO 2DU
rmat_620M2M	4	144	25.5%	5.4%	1.5%	19.1%	11.6%	5.9%	0.3%	10.0%	4.4%	0.0%	0.1%	8.0%	45.3%	19.9%
	5	169	22.7%	3.5%	13.6%	23.2%	24.9%	13.2%	2.3%	2.6%	13.0%	14.9%	25.4%	8.3%	44.3%	35.7%
	7	225	6.8%	12.1%	18.5%	17.0%	4.4%	47.6%	7.7%	0.9%	1.2%	5.4%	0.7%	7.2%	33.3%	31.5%
	8	256	21.1%	35.1%	19.3%	24.1%	11.8%	44.3%	12.9%	24.5%	6.8%	6.6%	25.4%	13.6%	46.6%	3.5%
rmat_680M3M	4	144	27.4%	48.6%	22.2%	22.8%	2.6%	4.3%	5.6%	4.4%	7.3%	3.9%	0.0%	0.2%	45.6%	27.6%
	5	169	27.2%	26.2%	8.2%	20.7%	7.0%	15.0%	6.7%	4.4%	7.5%	2.9%	4.5%	5.4%	36.1%	34.5%
	7	225	15.7%	3.0%	27.9%	0.8%	6.5%	7.4%	0.8%	13.3%	4.8%	1.3%	7.2%	2.0%	46.6%	40.7%
	8	256	31.5%	11.5%	32.0%	5.9%	10.9%	23.1%	1.6%	8.4%	2.1%	3.2%	9.9%	2.7%	49.4%	15.4%
rmat_690M2M	4	144	19.2%	21.0%	4.5%	8.9%	8.9%	17.7%	2.4%	25.9%	6.0%	8.3%	0.8%	5.5%	41.9%	12.0%
	5	169	13.7%	11.7%	10.8%	8.1%	23.3%	13.2%	2.7%	12.2%	8.2%	10.1%	17.7%	5.0%	38.3%	25.0%
	7	225	9.8%	6.5%	23.4%	14.7%	3.2%	38.6%	2.5%	3.5%	4.5%	13.2%	5.0%	16.0%	35.3%	34.6%
	8	256	6.9%	32.8%	41.5%	48.9%	31.4%	24.2%	10.4%	18.6%	16.7%	12.1%	21.6%	31.7%	27.1%	10.1%
Geometric Mean			16.9	12.7	14.0	12.9	9.3	16.4	3.0	7.4	5.6	4.0	2.9	5.3	40.2	20.3

or 256 MPI processes than it is 225 processes. That means each new hardware configuration needs to reexecute SpMV for all the training matrices across all execution configurations. This is not a scalable way to predict performance.

On the other hand, the micro-benchmark based model only requires to run classical benchmark for the platform which are not specific to SpMV. In other words, the cost of this model does not scale with the number of hardware configuration or execution algorithms for SpMV.

Also, the SVR based model is hard to interpret. Once the model is trained, even if it is accurate, the only thing we get is a third degree polynomial. One can not easily pinpoint from the model why the execution get the time it gets.

However, the finer model based on micro-benchmarks provides a direct explanation of the runtime of the algorithm. One can easily understand from the model how the runtime is formed. This can tell us for a particular matrix whether the bottleneck is in the memory subsystem, or in the MPI communication. This type of model is easier to explain.

4.9 Conclusion

In this chapter, we provide three SpMV models to predict the run time of the SpMV on using two matrix reprensentation and three distributed memory strategoes. Two of the performance models can predict the run time accurately enough to identify the optimal strategy to compute SpMV. While the SVR model provides usually better accuracy in the prediction of the runtime, its training is computationally expensive and the model does not provide any insight of why the performance is the way it is. On the other hand, the microbenchmark based model provdes explainable prediction, that only require classic performance benchmark of the architecture.

CHAPTER 5

POSTMORTEM GRAPH ANALYSIS ON THE TEMPORAL GRAPH

Temporal graphs capture changes in relational data over time and have been of increasing interest to data analysts. Most research focuses on *streaming* algorithms that incrementally update an analysis to account for the changes in the graph. However, one can also be interested in understanding the nature of changes in the graph over time. In such a case, they perform a *postmortem* analysis on different points in time where all the data known in advance

We study in this paper a *postmortem* analysis of *Pagerank* over-time on graphs that are defined by temporal relational event databases. A relation between two entities at a particular point in time will form an edge between these two entities and that will remain in the graph for a fixed period of time.

While one can reuse a streaming algorithm for that purpose, leveraging the availability of all the data from the beginning can be beneficial. Postmortem analysis enables encoding the temporal graph with a more efficient graph representation. Also, it provides an additional level of parallelism since one can not only parallelize within a particular timestamp but also across different timestamps. We will show that depending on the properties of the temporal data, either parallelization can be better, and in some cases, a combination of both approaches is preferable.

We experimentally show across 7 databases and across different temporal derivations of the graph that postmortem analysis can be between 50 times and 880 times faster than streaming analysis.

5.1 Introduction

Graphs have been used to model various natural, social, and constructed objects and phenomena such as the brain, friendship relations, and the physical road infrastructures. Such models help understanding more deeply the objects we study. They have been used to identify terrorists [81, 82, 83], understand the link between traffic and economic activity [84, 85, 86], or identify keywords in text [87, 88]. There are numerous analyses conducted on these graphs for a different type of us-

age, including Pagerank [89], betweenness and closeness centrality [90, 91], modularity-optimizing community detection [92, 52], k-core decomposition [93, 94].

These graphs are often analyzed as static graphs, but fundamentally the objects they model evolve over time: Roads are constructed and blocked off; Humans form new relations while others fade away. A more accurate model would be to define a *temporal graph* [95] that has vertices and edges that only exist for some periods of time. A common type of analysis on these temporal graphs is *streaming analysis* where an analysis is performed on the most up-to-date version of the graph. Obviously recomputing the analysis from scratch would be expensive and in many cases, it is possible to perform an incremental update on the analysis by starting from the results of recent analyses and accounting for only the latest changes in the graph. This has been done on many analyses including streaming Pagerank [96, 97], streaming Closeness Centrality and Betweenness Centrality [98, 99], streaming k-core [93, 94], and many others.

We are interested in this paper in a different form of analysis that sees the graph as a time series. In this analysis, we assume that we know the entire temporal graph at the beginning of the analysis; we refer to the analysis as being *postmortem*. (Some people may refer to that sort of analysis as being offline; but we chose not to refer to it this way to avoid confusions.) This is in contrast with a streaming analysis which discovers the graph during the analysis. Various problems on temporal graph have been investigated, including diameter change [19], and rank of web pages change [20] on the web.

We assume that the analysis is conducted at regular interval in time. Also the temporal graphs are defined by edges that appear at a particular point in time and remain in the graph for a constant amount of time. As such, the temporal graph can model edge addition and deletion, as well as vertex addition and deletion.

We will also restrict our analysis to computing Pagerank [89]. It is a simple analysis that is well understood, with known streaming algorithms [96, 97]. And it applies to a wide variety of applications.

In this paper, we show how to perform a postmortem temporal analysis of a graph using Pagerank on a shared-memory parallel system. We show that postmortem analysis is much faster than an equivalent streaming analysis and static (offline) analysis. In particular, we show that postmortem analysis provides benefits over static and streaming execution model. The challenges and contribu-

tions of this paper include:

Data Representation: Streaming and static have a fairly well set representation that have their own pros and cons. But in a postmortem analysis, representing the temporal data offers tradeoff between volume of memory and performance of the analysis. We present our data representation in Section 5.4.1. We investigate and evaluate the tradeoffs.

Leveraging incremental methods: There are several methods to reduce the amount of work when computing Pagerank in a streaming mode. Upon some update, the graph is still quite the same as it was, the values of Pagerank are going to be related, and incremental methods have been developed for Pagerank. Based on existing methods (described in Section 5.3.3), we develop an incremental method appropriate for this particular use case in Section 5.4.2.

Different Level of Parallelization: In Postmortem analysis, one can compute Pagerank on each graph simultaneously. Of course, the calculation of Pagerank on a particular graph is also a fundamentally parallel computation. Questions of load balance, incompatibility with incremental optimization, and scheduling need to address to benefit the most from modern platforms. We investigate these questions in Section 5.4.3.

SpMV-style vs SpMM-inspired Postmortem Pagerank: Pagerank is fundamentally similar to a sparse matrix-vector multiplication (SpMV) operation. However, we know that sparse matrix-matrix multiplication (SpMM) can obtain higher performance. We discuss how we take inspiration from SpMM and rephrase the calculation of Pagerank on a temporal graph to obtain the benefits of an SpMM formulation without compromising other optimizations in Section 5.4.4.

Demonstrate the efficiency of Postmortem analysis: It makes intuitive sense that postmortem analysis offers more avenues for optimization than both offline and streaming analysis. But to what extent is postmortem preferable. We evaluate experimentally the question in Section 5.6 and show that in our benchmark postmortem analysis can be between 50 times to 400 times faster than streaming analysis.

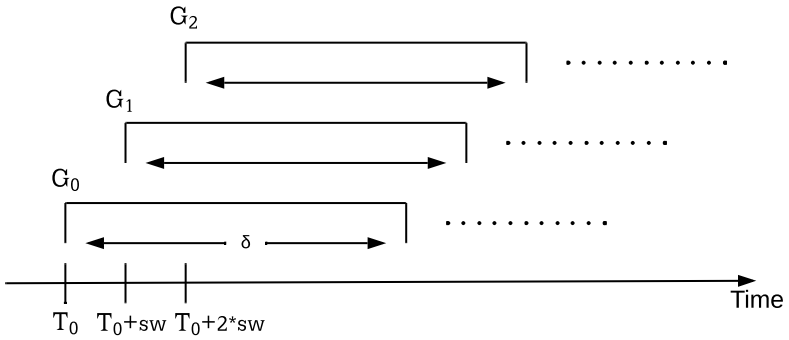


Figure 5.1: Sliding Window Model

5.2 Problem Statement

5.2.1 Temporal Graph from Temporal Events

Temporal Edge Set: We assume our input is a set of edges of the form $Events = \langle u, v, t \rangle$, where u, v are vertices from some vertex set V (the elements of V known because of offline behavior), and t is an integer timestamp. Without loss of generality, we can assume that entries are listed in increasing timestamp order. We call the entire sequence of such triples a temporal edge set, and each triple is an event.

Note that a streaming model assumes that the elements of the set are disclosed, monotonously in time, over the execution of the application. But in a postmortem model, all the temporal edges are known at the beginning of the application.

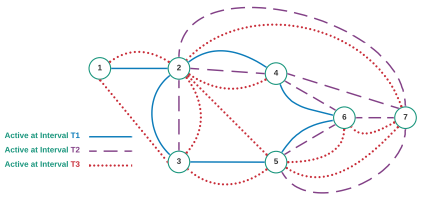
Sliding Window Model: We define $G(T_s, T_e)$ as the graph induced by the events that occured between T_s and T_e. That is to say, $G(T_s, T_e) = (V, E)$ where $\{e = (u, v) \in E | \exists (u, v, t) \in Event, T_s \leq t \leq T_e\}$.

In this paper, we are interested in analyzing the sequence of graph $(G_0 = G(T_0,\ T_0 + \delta), G_1 = G(T_1,\ T_1 + \delta), G_2 = G(T_2,\ T_2 + \delta), \ldots, G_m = G(T_m,\ T_m + \delta))$ with $T_i = T_{i-1} + sw\}$ and T_0 is set by the beginning of the dataset. In other words, the temporal graph is defined by a sequence of graphs generated by sliding a window over the time period. The window is of fixed size δ and each window slide by a sliding offset of sw time-units compared to the previous one. This sliding window model is illustrated in Figure 5.1.

Figure 5.2a presents an example of a list of temporal edges for a graph. The edges arrive

Edges		Edge Arrival Time	Time Interval		
v_1	v_2		T1	T2	T3
1	2	06/21/2021	✓	×	×
3	5	06/25/2021	✓	×	×
4	6	07/11/2021	✓	✓	×
2	3	08/01/2021	✓	✓	✓
2	4	08/11/2021	✓	✓	✓
5	6	09/13/2021	✓	✓	✓
2	7	10/02/2021	×	✓	✓
4	7	10/05/2021	×	✓	✓
5	7	10/06/2021	×	✓	✓
6	7	10/09/2021	×	✓	✓
1	2	11/05/2021	×	×	✓
1	3	11/06/2021	×	×	✓
2	5	11/09/2021	×	×	✓
3	5	11/12/2021	×	×	✓

(a) Temporal edge list[Time interval T1 = (6/1/2021-9/15/2021), T2 = (7/1/2021-10/15/2021) and T3 = (8/1/2021-1/15/2022)]



(b) Temporal Graph

Figure 5.2: Edgelist and temporal graph.

between 06/21/2021 and 11/12/2021. Maybe the analyst is interested in analyzing phenomena that take some time to unfold and select a window of size $\delta = 3\frac{1}{2}$ months. The first graph G_0 includes edges arriving after 6/1/2021 and until 9/15/2021. After that it will move forward the starting time of the second graph G_1 by $sw = 1$ month and the time interval for G_1 will be 7/1/2021-10/15/2021). Figure 5.2b shows the active edges for the first 3 graphs of the sequence of the temporal graph.

5.2.2 Postmortem Graph Analysis for Pagerank

Pagerank is a metric of the importance of vertices in a graph, originally used on webpages modeled as a directed graph [89]. Let v be a vertex, $\Gamma^+(v)$ be the set of vertices v points to, and $\Gamma^-(v)$ be the set of vertices that point to v. For a teleportation probability α, the Pagerank(PR) [89] equation for v is recursively defined as:

$$PR(v) = \frac{\alpha}{|V|} + (1-\alpha) \sum_{u \in \Gamma^-(v)} \frac{PR(u)}{|\Gamma^+(u)|} \tag{5.1}$$

While Pagerank values for each node of the graph could be obtained by solving the system of equations, it is more common to compute Pagerank iteratively. The Pagerank equation is evaluated from previous values of Pagerank. This involves performing one Sparse Matrix-Vector multiplica-

Algorithm 6 Pagerank on Temporal Graph

Input: $Events, sw, \delta, T_0, m$

1: $i \leftarrow 0$
2: **while** $i \leq m$ **do**
3: $\quad PAGERANK_i \leftarrow$ PagerankAlgorithm($G(T_i,\ T_i + \delta)$)
4: $\quad i \leftarrow i + 1$
5: $\quad T_i \leftarrow T_{i-1} + sw$
6: **end while**

tion (SpMV). After some iterations, the values converge to the solution of the equation. Implementations usually numerically check for convergence after each iteration and execute a fixed number of iterations at most. Beamer and Scott *et al.* [100] presented how to reduce Pagerank communication via propagation blocking; and although this paper does not leverage that particular technique, we believe it is compatible.

The problem we are trying to solve is to compute Pagerank on all graphs in the sequence. Sequentially one could solve the problem with the simple method given in Algorithm 6. But one does not have to compute the different Pagerank vectors in-order. They could compute in different orders. Of course, applications will have a downstream analysis that will depend on these vectors.

5.3 Background and Related Works

5.3.1 Applications of the Sliding Window Model

The formulation of the temporal graph based on sliding windows from an event database is appropriate for many applications. Parameters $delta$ and sw are application parameter. They enable the analyst to explore a dataset at different time scales and resolutions.

For instance, consider the analysis of academic collaboration networks. One can define events based on papers, if authors a_1 and a_2 co-wrote a paper on day d, you insert a tuple (a_1, a_2, d) in $Events$.

Setting a larger value of $\delta = 10$ years will enable the analyst to think of the important of authors in a scientific era. Meanwhile, setting a smaller value of $\delta = 1$ year will enable to study current collaborator dynamic. Neither value for the parameter is inherently better, but they enable to study different social phenomenon. The sw parameter is essentially a resolution parameter. It enables to

provide fewer or more points in the generated time series.

The temporal graph constructed this way could be analyzed in various way. While we focus on Pagerank in this paper, different analysis could be done using other kernels like closeness and betweenness centrality, connecting component, k-core, etc.

5.3.2 Temporal Graph Analysis

We are not the first to analyze graphs temporally from event data. Hossain, Murshed *et al.* analyzed communication network dynamics during organizational crisis [101]. They showed that some actors of an organization that are prominent or more active will become central during the organizational crisis. Now, analyzing this kind of problem requires insight into periodic changes in the dynamic communication graph. Time interval-wise analyses show the impact of actor's changes on the organization and one can find how the role of an actor evolves during a crisis and understand the underlying cause.

Stolman and Matulef [102] proposed a *HyperHeadTail* streaming algorithm which can estimate the degree distribution of a dynamic graphs. The dynamicity is represented as a multigraphs where two identical vertices can hold multiple edges for different times. In their work, the divided the multigraph into multiple window and perform degree distribution on different window graph. The work is formulated under the streaming paradigm where a batch of edges will arrive the system and gradually perform algorithm. Han and Sethu [103] have proposed an edge sampling algorithm for triangle counting of dynamic graphs.

Chen and Lui proposed a unified framework [104] to estimate the graphlet (small connected subgraph pattern) counts of the whole graph as well as the graphlet counts of individual nodes under the streaming graph model. To understand the structure of graph, Gabert *et al.* provided postmortem analysis to dense region in a dynamic graph using k-cores decomposition [105]. Previous streaming algorithms for k-core were designed [93].

Many centrality metrics can be used to find the important vertices in the graph, and multiple have been considered on dynamic graphs. Nathan and Bader [106] proposed a dynamic algorithm for updating Katz centrality in graphs under the streaming model. Under a streaming model, incrementally updating closeness centrality [107] and betweeness centrality [98] have also been studied.

5.3.3 Execution Model

There are three main ways to compute the many Pagerank values in the temporal model.

Offline Pagerank Model

One can build independently a graph for each window and perform Pagerank. It requires reconstructing a correct graph from the $Event$ data many times. The cost of the application will be driven by the cost of building the graphs, but the application becomes massively parallel since each time window can be computed independently. As such, this is an execution model that is appropriate for a massively distributed system such as a cloud platform.

Streaming Pagerank Model

In the streaming model, the application maintain only a single copy of the graph. The version of the graph that is stored is meant to represent the graph as it is "now". Updates to the graph come as an edge stream. The streaming system needs to adjust the representation of the graph to account for the new edges and recompute the analysis accordingly. Middlewares have been built to support streaming graphs like STINGER [97] and ElGA [108]. These middlewares spend significant effort in maintaining a valid representation of the graph upon updates made to the graph by using advanced datastructures that minimize modification cost.

One of the benefit of streaming analysis is that when the calculation on the updated graph is made, the system has access to the result of the analysis on the previous version of the graph. This can lead to incremental algorithms which require less computation than recomputing the analysis from scratch [106, 107, 98, 104, 93].

We present now one way to incrementally update Pagerank values. A directed graph $G(V, E)$ with vertex and edge set V and E can be represented by a sparse matrix A where an edge($i \to j$) is represented by $a_{ij} = 1$. If we represent the out degree of the graph by a diagonal matrix D then, *Pagerank* can be defined by the linear system [109, 110],

$$(I - \alpha A^T D^{-1})x = (1 - \alpha)v \tag{5.2}$$

Where α is the "teleportation" constant, v is the initial *Pagerank* vector usually filled by $1/|v|$ and x is the *Pagerank* vector. Jason presented [97] an approximation version of *Pagerank* for the streaming graph,

$$\Delta x^{k+1} = \alpha A_{\Delta}^{T} D_{\Delta}^{-1} \Delta X^{k} + \alpha (A_{\Delta}^{T} D_{\Delta}^{-1} - A^{T} D^{-1}) x + r \tag{5.3}$$

where modifications of the streaming graph by edge addition or deletion are represented by Δ and k is represent the previous iteration. Here r is the residual error, $r = (1-\alpha)v - (I - \alpha A^{T} D^{-1})x$.

The streaming execution model reduces the graph building time from offline execution. But they introduce more complex data structures to efficiently support insert and remove operations. The streaming model also enables to leverage incremental algorithms to decrease the total amount of computation. But it suffers from an inherent lack of parallelism. Since only one version of the graph is stored, the only available parallelism comes from the Pagerank computation itself and the graph updating procedure.

Postmortem Pagerank Model

We argue in this paper that in a postmortem model, we can produce analysis much faster than both the offline and streaming execution model.

Both offline and streaming models have significant graph construction cost, even though they are structured differently. In a postmortem model, we can build the graph representation in a single operation in a way that enable to access all the time windows.

The offline model benefits from high parallelism as it supports parallelism across different time-window and inside the kernel. The streaming model does not enable parallelism across time-window. But the postmortem model can support both levels of parallelism.

The streaming model leverages incremental updates to the Pagerank computation. Even if a postmortem execution leverages parallelism over different time-window, it can still arrange its calculation to leverage knowledge from the previous time-window if that information is known.

rowA = [0, 3, 9, 12, 16, 21, 24, 28]
colA = [2, 2, 3, 1, 1, 3, 4, 5, 7, 1, 2, 5, 5, 2, 6, 7, 2, 3, 3, 6, 7, 4, 5, 7, 2, 4, 5, 6]
timeA = [06/21/2021, 11/05/2021, 11/06/2021, 06/21/2021, 11/05/2021, 08/01/2021, 08/11/2021, 11/09/2021, 10/02/2021, 11/06/2021, 08/01/2021, 06/25/2021, 11/12/2021, 08/11/2021, 07/11/2021, 10/05/2021, 11/09/2021, 06/25/2021, 11/12/2021, 09/13/2021, 10/06/2021, 07/11/2021, 09/13/2021, 10/09/2021, 10/02/2021, 10/05/2021, 10/06/2021, 10/09/2021]

Figure 5.3: Temporal CSR Representation

5.4 Postmortem Graph Analysis

5.4.1 Data Representation

The performance of graph analyses vastly depends on the graph storage system. The offline and streaming model of computing Pagerank on a temporal graph suffer from data representation problem that can be addressed in a postmortem case. The CSR storage format is widely popular for the sparse matrices which is a fundamental attribute for Pagerank calculation using sparse-matrix vector multiplication (SpMV). We use a format that is similar derived from the CSR format.

Figure 5.3 shows a temporal CSR format for the graph presented in Figure 5.2b. Usually CSR requires two vectors, `rowA` and `colA`, to represent a graph. The `colA` vector is a concatenation of the adjacency list of the graph, while `rowA` indicates where the adjacency of each vertex starts. In other words, the first vertex neighbors are listed in `colA` between indices `rowA[0]` and `rowA[1]`. There are $V + 1$ entries in $rowA$ and E entries in `colA`.

But for postmortem analysis we keep an additional vector which tracks timestamps for each edge, `timeA`, which will have the same size as the `colA` vector. There are duplicate entries in `colA`, because two vertices may appear multiple times in $Events$ for different times. We store the neighbors of a vertex sorted by neighbors, and then by timestamp.

In this representation, we can iterate through the neighbors of vertices of a particular graph. For a particular vertex v of G_0 (for instance), the edges are all stored between `rowA[v]` and `rowA[v+1]`, but some of them do not exist for graph G_0. For a possible neighbor, the different times at which an event occured are stored consecutively in the temporal CSR representation. So as long as one of the edges has a timestamp between T_0 and $T_0 + \delta$, then it exists in G_0.

This basic representation requires one vector of size $V + 1$, and two vectors of size $|Events|$. One iteration of a Pagerank calculation requires performing one SpMV. This involves traversing the

neighbors of every vertex and has a complexity of $\Theta(|Events|)$.

When the span of time increases or when δ decreases, the total number of events is not related to the total number of edges in one particular graph. Since $|Events|$ could be arbitrarily larger than the number of edges in any particular graph, the complexity of calculating a single SpMV can be arbitrarily larger than it should be.

To remedy this, we partition the representation in many multi-window graphs. Each multi-window graph represents a contiguous number of graphs and only stores the edges that are relevant to these graphs. We distribute the graphs uniformly to the multi-window graphs. So if the analysis involve X graphs and we represent the data with Y multi-window graph, each multi-window graph will contain $\frac{Y}{X}$ graphs.

A multi-window graph w has a vertex set V_w and an edge set E_w. Note that for a particular multi-window graph, V_w is typically smaller than the set of all vertices V since a vertex may not appear in that multi-window. Also, note that some edges may appear in two (or more) multi-window graph since an edge can appear in different consecutive graphs which could be in different multi-window graphs. In other words, this representation consumes more memory since $\sum_w |E_w| \geq |Events|$.

In this representation, performing SpMV for a graph only requires traversing the edges in the multi-window graph that contain the graph. And therefore computing SpMV for a graph in multi-window w has a complexity of $\Theta(|E_w|)$ which is closer to the number of edges in that graph than $\Theta(|Events|)$ is.

The question of how many multi-window graph remains to be investigated. We propose that a window graph should be accomodate by the system memory when computing Pagerank. The total memory cost of the representation is $encoding * (\sum_w |V_w| + 2 * |E_w|)$ where $encoding$ accounts for the size of the number encoding (we use 64-bit for all data). And we need to retain memory available to store the intermediate data of Pagerank.

5.4.2 Partial Initialization

To calculate Pagerank, one needs to initialize the Pagerank values and the most common initialization value is $\frac{1}{|V|}$ where $|V|$ is the number of vertices in the graph. For us, the default would be $\frac{1}{|V|}$.

Now, the postmortem analysis is a sliding window process and two consecutive graphs share

most of their vertices and in many case they share most of their edges. So the Pagerank values should be similar. And since Pagerank is a converging iterative process, having a better initial guess for the values should decrease the number of iterations to converge.

We build on out previous work [111] and propose a *partial initialization* for graph G_i that is a successor of window interval G_{i-1}. We denote by V_i all the vertices in graph G_i. We initialize the Pagerank of a vertex G_i simply based on the Pagerank of its neighbors that were present in G_{i-1} normalized to account for missing vertices. More specifically,:

$$PR_i[u] = \frac{|V_i \cap V_{i-1}|}{|V_i|} * \frac{PR_{i-1}[u]}{\sum_{v \in V_i \cap V_{i-1}} PR_{i-1}[v]} \tag{5.4}$$

Because the set of vertices encoded in a multi-window graph can be very different from the set encoded in the next multi-window graph, computing the indexing can be tedious. So we do not perform partial initialization across different multi-window graph. But since there are likely only few multi-window graph, the loss will be small.

We will experimentally validate the impact of partial initialization on convergence time.

5.4.3 Different Level Parallelization on Pagerank

We can utilize parallel computing at two different levels. We can parallelize over different time-window since they are all available in the postmortem representation, we call this *window-level parallelization*. We can also use parallelism inside the application kernel, here Pagerank, and we call this *application-level parallelization*. We can also leverage both at the same time which we call *nested parallelization*.

Window-Level Parallelization

Window-level parallelization is good for a well balanced graph and large number of time-window. If some graph are much larger than other ones, then window-level parallelization could lead to load imbalance. Also, if we have a limited number of time-window graph then we will only have a small amount of parallelism available.

Partial initialization may also be difficult to leverage in window-level parallelization. When

starting to process graph G_i, one can only perform partial initialization if the Pagerank values of G_{i-1} are known by the thread. In practice, we implement the algorithm so if the same thread processes G_{i-1} and G_i, then partial initialization occurs.

Because of these two effects, a classic work scheduler is unlikely to be satisfactory. Think of OpenMP's classic dynamic scheduler. With a granularity of 1, it would likely allocate G_i and G_{i-1} always to different threads. This would result in the benefits of partial initialization being negated. A larger granularity would reduce the amount of parallelism available and takes the chance of having a single chunk of work contain graphs that are significantly larger than the rest of the chunks. And this would lead to load imbalance.

To remedy this problem, we opt for the work-stealing scheduler of Intel TBB. With a work-stealing scheduler, the threads will be originally allocated a chunk of contiguous work. That contiguous chunk will only be broken when the other threads are running out of work.

Application-Level Parallelization

In this model graphs are processed one at a time, in order from the first graph to the last graph. And all the parallelism happens inside the call to Pagerank for that particular graph. In this case, the parallelization is over the vertices of the graph.

In this model, we can use partial parallelization for every graph except first one of each multi-window graph.

This model will perform well if the workload in each graph significant compared to the total amount of work. In other words, we recommend using application-level parallelization for an instance with low number of graphs or where a few graphs carry most of the load of the analysis.

Nested Parallelization

Nested parallelism mixes both window-level and application-level parallelism. In other words, different graphs are performed in parallel and each Pagerank calculation is also performed in parallel. This mode of operation offers the most parallelism and is likely to provide benefits of both modes of operations.

This nested form of parallelism can be challenging for some parallel computing middleware. We use TBB and its workstealing scheduler to orchestrate the execution. This model will perform better

in a large temporal graph with moderate number of graphs or well balanced window-application workload.

5.4.4 SpMM-inspired Postmortem Pagerank

When dealing with sparse matrices vector multiplication the primary bottleneck of executing the algorithm tend to come from moving the matrix from DRAM to the core, and from accessing the input vector in a random pattern. If the application supports it, it can be beneficial to execute multiple SpMVs simultaneously on different vectors and on the same matrix. One can perform multiple multiplication by reading the matrix only once. And interleaving the input vectors can transform the access patterns from mostly random to mostly regular. This is a common optimization in linear algebra: for instance LOBPCG tend to achieve higher performance than Lanczos to extract eigenvectors [112], and computing simultaneously multiple derivatives of radial basis functions [113].

Here, we have a similar structure. We compute Pagerank on multiple graphs but if the graphs are in the same multi-window graph, then the representation of the two graphs in memory is actually the same multi-window graph. By keeping in memory the intermediate value of multiple graphs' Pagerank calculation, we can perform one iteration of many Pageranks by accessing the multi-window graph only once. Also, since the graphs are likely sharing many common edges, the access pattern to the Pagerank vectors become also more regular.

We will abuse the name and refer to this method as an SpMM method. Even though technically, a different matrix is being used for the different Pageranks. We will also refer to the numbers of Pagerank being computed simultaneously as vector length by analogy to vector processing which plays a major role in SpMM implementation. Even though, the code may not actually use vectorization in practice.

Now, if we process consecutive graphs, say $G_0, G_1, \ldots, G_7$, then we are going to lose partial initialization. Indeed, the result of G_0 is needed to perform partial initialization on G_1. So we divided the multi-window graph into vector-length (e.g., 8) regions and picked first the graph from each region. This will perform first for instance $G_0, G_{10}, G_{20}, ...G_{70}$. These 8 Pageranks will not benefit from partial initialization. However, the next batch of graph processed will be $G_1, G_{11}, G_{21}, ...G_{71}$ which will all benefit from partial initialization.

We will investigate experimentally the impact of this SpMM-inspired optimization.

Table 5.1: Graphs and Parameters

Name(Events)	Sliding Offset	Window Size
ca-cit-HepTh (2,673,133)	12 hours, 1, 2 days	10, 15, 90, 180, 730, 1460 days
stackoverflow (47,903,266)	12 hours, 1 day	10, 15, 90, 180, 730 days
askubuntu (726,661)		90, 180 days
Youtube-Growth (12,223,774)		60, 90 days
epinions-user-ratings (13,668,281)		
ia-enron-email (1,134,990)	12 hours, 2 days	2, 4 days
wiki-talk (6,100,538)	12 hours, 1, 2, 4 days	10, 15, 90, 180 days

5.5 Experimental Settings

5.5.1 Execution environment

All the experiments are performed on a node which is equipped with two Intel Xeon Gold model 6248R (Cascade Lake architecture, 24 cores per processor, no hyperthreading, 36MB L3 Cache) and 384GB GB of DDR4 memory. The operating system used in the machine is Linux `3.10.0`.

All the codes are writen in C++ and compiled by the Intel C++ compiler `icpc` version `19.1.3.304`. Codes are compiled with optimization flag `-O3` and `xCORE-AVX512` flags, so the compiler generates a binary optimized for the architecture.

All the streaming version of *Pagerank* are performed on the `STINGER` [97] framework. `STINGER` is a package designed to support streaming graph analytics by using in-memory parallel computation to accelerate the computation. `STINGER` supports Pagerank with an incremental algorithm. The only modifications to `STINGER` that we performed are to the edge event injection logic so as to updates in batches equivalent to the postmortem code. This makes the code bases produce the same results and makes the comparison fair.

5.5.2 Graphs

We perform our experiments on real-world data sets to avoid the bias introduced by random graph generator. We select graphs from the *Stanford Large Network Dataset Collection* (SNAP) [59], network repository, and *DIMACS* [60, 61] data sets that are frequently used in graph algorithm research.

Table 5.1 presents the temporal graphs and provide details of the application parameters (window size, and window offset) that we set. We picked parameters that would look at the data at different scale and resolution. Still we choose to have the time-windows overlap (all the graph share some edges from its previous graph) since it seems likely analysis would always want that property. We assume all the edges of the graphs are sorted in non-decreasing order of their arrival time.

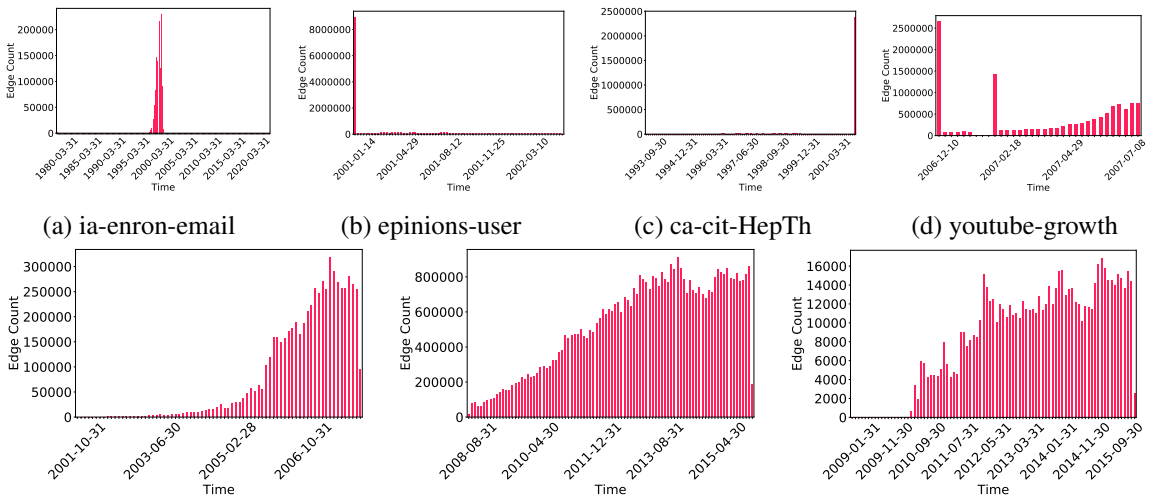
5.6 Results

5.6.1 Edge Distribution of Temporal Graph

Figure 5.4 presents the edge distribution for all the graphs over time. We can see the patterns of temporal edges are different for different graphs. This provides a diversity of instances to test our methods.

Figure 5.4a shows the the email communication of the *Enron Corpus* where we can see some big spike around 2001. These spikes represents the period of time when Enron scandal happened which is the period of time mostly captured by the dataset. Figure 5.4b shows the user review ratings collected by Epinions. Epinions was established in 1999 and peaked around 2001 and later they acquired by eBay. It is a bipartite graph, an edge represents a user reviewing a product. We can see around 2001 user reviews shows a huge spikes which is the reason the company was acquired. Citation graph `ca-cit-HepTh`5.4c also shows an irregular distribution pattern of temporal edges. On these networks, the Pagerank calculation bottleneck will be on a few graphs since few time-window cover most the edges in the dataset. We will see that this distribution of work will make application-level parallelism more efficient than window-level parallelization.

The temporal edge distribution for `wiki-talk` (Figure 5.4e), `askubuntu` (Figure 5.4g), and `stackoverflow` (Figure 5.4f) show increasing amount of streaming edges over time. But the number of edges that come in is relatively smooth. Graphs with balanced high-volume edges with



(a) ia-enron-email (b) epinions-user (c) ca-cit-HepTh (d) youtube-growth

(e) wigki-talk (f) stackoverflow (g) askubuntu

Figure 5.4: Temporal graph edge distribution over the time period.

large number of windows are well suited for nested parallelism.

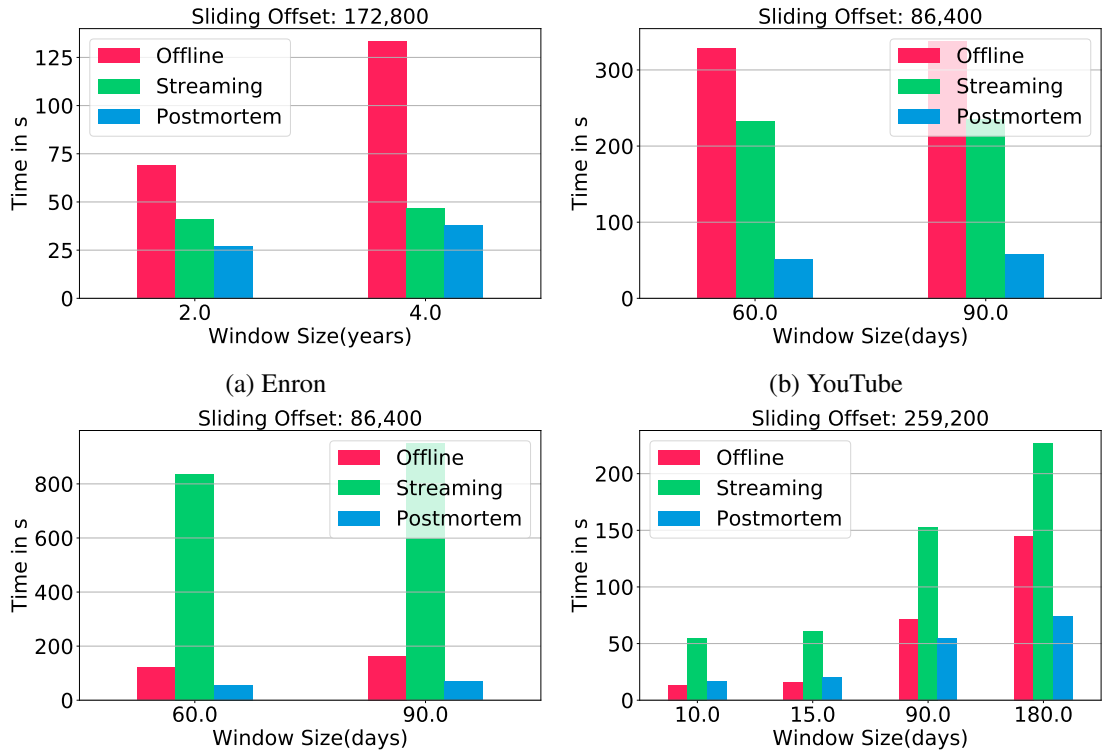
`youtube-growth` 5.4d shows a pattern that is both bursty by moment but steady in general.

5.6.2 Postmortem is usually faster than Offline and Streaming

We compare the performance of the three execution models: Offline, Streaming and Postmortem. Postmortem here uses partial initialization and each temporal graph is partitioning into 6 multi-window graphs. Postmortem uses only an application-level parallelism with a static scheduler. In other words, this is a bare-bone postmortem computation where the execution parameters have not been tuned.

Figure 5.5 shows the comparison among Naive, Streaming and Postmortem Pagerank for some of the temporal graphs from four of our temporal datasets. The performance on `enron-email` is reported in Figure 5.5a. The Streaming version is faster than the offline version. But Postmortem outperforms both of them. Figure 5.5b shows the performance for the youtube dataset. On this graph as well, streaming is faster than offline; and postmortem is faster than both. The postmortem version outperforms Streaming by more than 3 times on that dataset.

Figure 5.5c shows the performance for the epinions dataset. On that dataset, streaming is much slower than both offline and postmortem. Postmortem is faster than both and about more than 40 times faster than streaming. Figure 5.5d shows results on the `wikitalk` dataset. Here streaming



(a) Enron (b) YouTube

(c) Epinion (d) Wikitalk

Figure 5.5: Performance of Naive, Streaming and Postmortem Pagerank

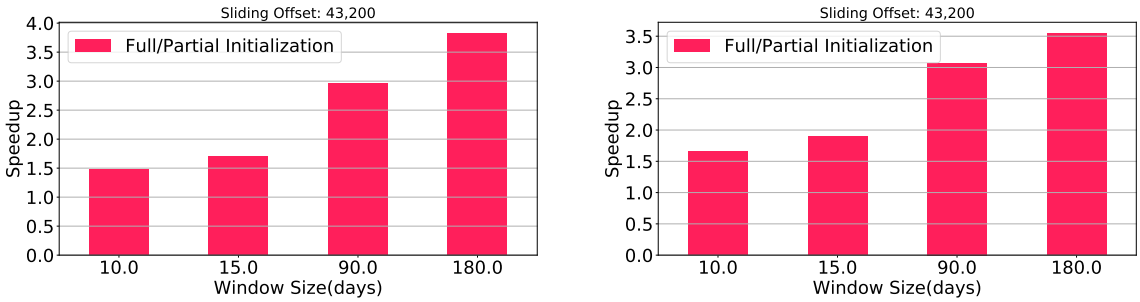
is also slower than both other methods. Postmortem is slightly slower than offline on small window size and better on larger ones.

5.6.3 Postmortem Detailed Results

Impact of Partial initialization

Figure 5.6 presents the impact of partial initialization on `stackoverflow` and `wiki-talk` temporal graph. It shows a performance gain that correlates with the size of the window and ranging from being 1.5 times faster to 3.5 times faster. It makes intuitive sense that the smart initialization improves the performance more on larger windows since the successive graphs become more similar.

We found similar speedup for other experimental graphs also (not shown). And from now, we will show results with partial initalization only rather than full initialization.



(a) Stackoverflow (b) Wiki-talk

Figure 5.6: Impact of partial initialization on postmortem graph analysis.

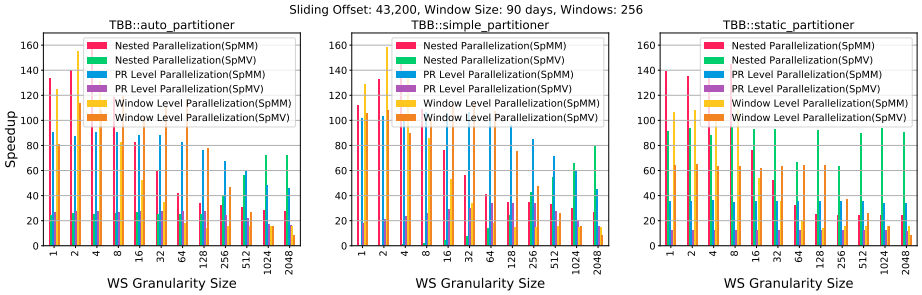


Figure 5.7: Postmortem Pagerank comparison over streaming on `wiki-talk` graph (SpMM load 16 Pagerank vectors).

Partitioner and Granularity

We saw that there is imbalance in the distribution of edges over time. But we also know that social graphs have power law edge distribution which makes the degree of the graph very unbalanced. As a result the bottleneck of the application is often the time-window graph that has many more edges than the other ones, and the block of vertices in the graph with extremely high degree.

In our experiment, we choose Intel's Thread Building Block(TBB) mechanism to parallelize the postmortem Pagerank to benefit from its scheduler. Now, TBB provide multiple `partitioners` and support different granularity. The `auto_partitioner` is the default workstealing scheduler while `simple_partitioner` is a variant of it. TBB also provides a `static_partitioner` which does not benefit from workstealing.

Choosing granularity requires experimental analysis. It depends on the partitioners, system cache memory, problem size, *etc.* We perform our experiments using a variety of granularity sizes to figure out the behavior of the results for certain attributes.

Figure 5.7 presents the performance of Pagerank on `wiki-talk` for different partitioner and

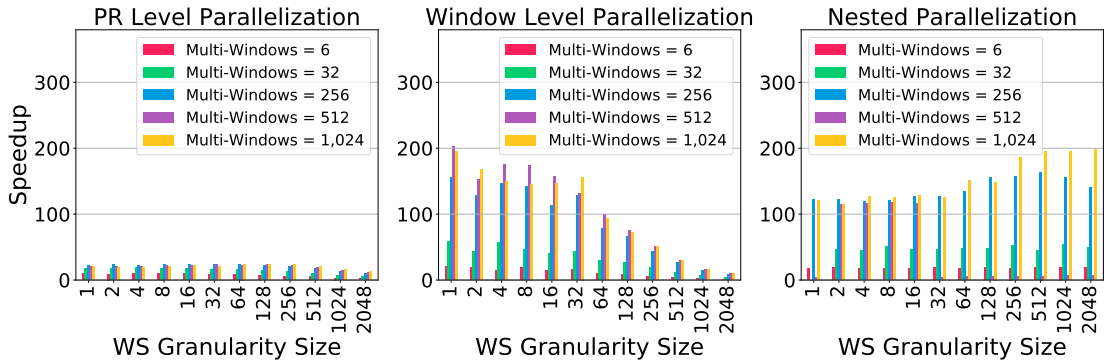


Figure 5.8: Postmortem Pagerank performance using TBB `auto_partitioner` for wiki-talk network for different number of multi-window.

granularity size for a certain sliding window and window size. The window size of the graph is 256 that means we can split the window-level parallelization at maximum by 256 where each worker thread will receive a single window. And we can see a performance drop after 128 for window-level parallelization because it lacks of parallelism. Nested and Pagerank-level parallelization show better result than window-level but they also lost some performance gain. The main reason also high granularity size assign large number of windows to each worker thread and make it imbalanced.

Overall, the performance of the `static_partitioner` seems worse than that of the other two partitioners. And the auto and simple partitioner are fairly comparable in performance.

Impact of the number of Multi-Window Graphs

The number of multi-window is an important parameter. If the number is too low, there runtime overhead due to traversing edges out of the graph the algorithm is considering will be high. If the number is too high, the system wastes memory and the impact of partial initialization will be lower.

The results presented in Figure 5.8 show that one the number of multi-window is “large enough”, the performance no longer varies.

Comparing SpMV to SpMM

The main difference between the *SpMV* and *SpMM* versions of postmortem Pagerank is that the SpMM version computes multiple Pagerank vector at once in a multi-window graph and treat them as a matrix. We choose a number of vector of either 8 or 16. Choosing a high number of vector in SpMM will reduce benefit of the partial initalization because all the initial Pagerank vectors will do

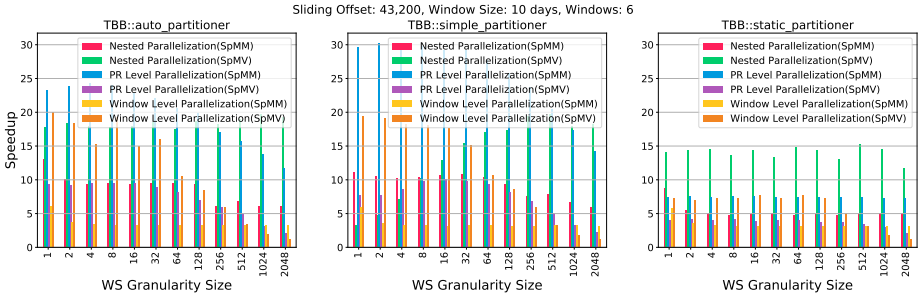


Figure 5.9: Postmortem Pagerank comparison over streaming on `wiki-talk` graph (SpMM load 16 Pagerank vectors).

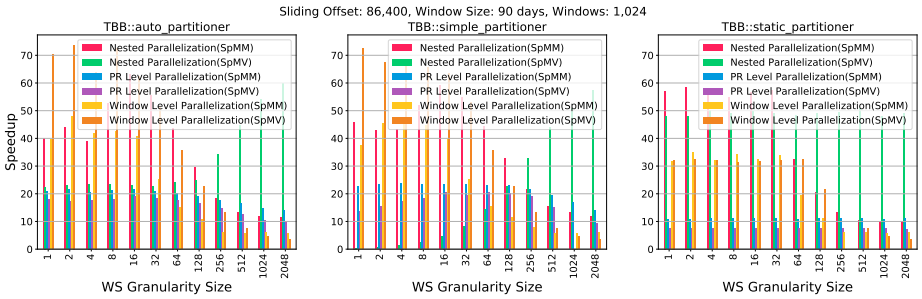


Figure 5.10: Postmortem Pagerank comparison over streaming on `wiki-talk` graph (SpMM load 16 Pagerank vectors).

full initialization.

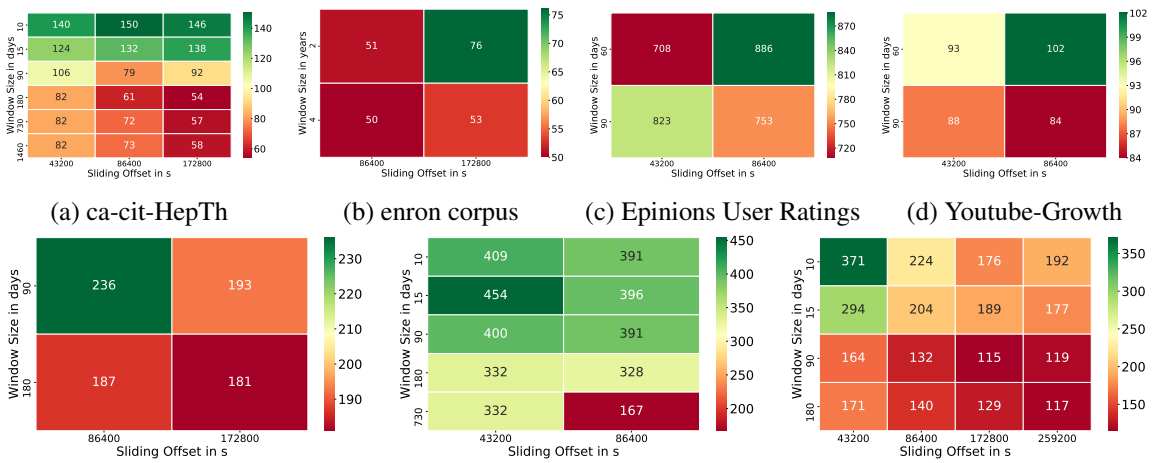
Figure 5.7 shows postmortem Pagerank performance on `wiki-talk`, where we can see the number of windows is 256. Our experimental results show that SpMM is usually much faster than SpMV.

Which level of parallelization?

Figure 5.9 shows better performance for Pagerank-level and nested but shows lack performance of the window-level parallelism. The main reason is the number of windows is only 6 where we have 48 available processors which stiffles the performance of window-level parallelism.

Figure 5.10 shows godd performance for window-level paralleization because of large number of windows. On the other hand at Figure 5.7 show better performance for nested parallelization.

Application-level parallelization is well suited for the well balanced windows with large window size graph. On the other hand window-level parallelization can out-perform other on the occasion where number of windows is large but the number of window size is smaller. That means less work in application level. Nested always show optimal or near optimal performance because it can adapt



(a) ca-cit-HepTh (b) enron corpus (c) Epinions User Ratings (d) Youtube-Growth

(e) Askubuntu (f) Stackoverflow (g) wiki-talk

Figure 5.11: Best performance gain by postmortem Pagerank over streaming version.

to both form of available parallelism.

Best Mechanism and suggest parameters

Figure 5.11 shows the overall best performance by the postmortem Pagerank relatively to the streaming model over the different configurations we tested. The Postmortem model proved to be between 50 and 800 times fater than the streaming model.

However, a user may not know how to set parameters. We provide a simple rules to set them that should lead to decent performance. Our experiments show that SpMM is never a bad choice. For partitioner, `auto_partitioner` with granularity size under 4 usually provides good results. To chose the type of parallelism, one need to look at the load balance in edges of different time windows. Unless the workload is dominated by couple of windows or very small number of multi-window, nested parallelization is the good fit for almost every graph.

We generated the performance of following this guidelines on `wiki-talk` across different sliding offset and window size and reported the results in Figure 5.12. The configuration does not report the best performance but reports very honorable performance at little tuning cost.

5.7 Conclusion

The study of performance of temporal graph analysis is often considered mostly in the streaming model where one wants to maintain the analysis current with the most recent data. However an

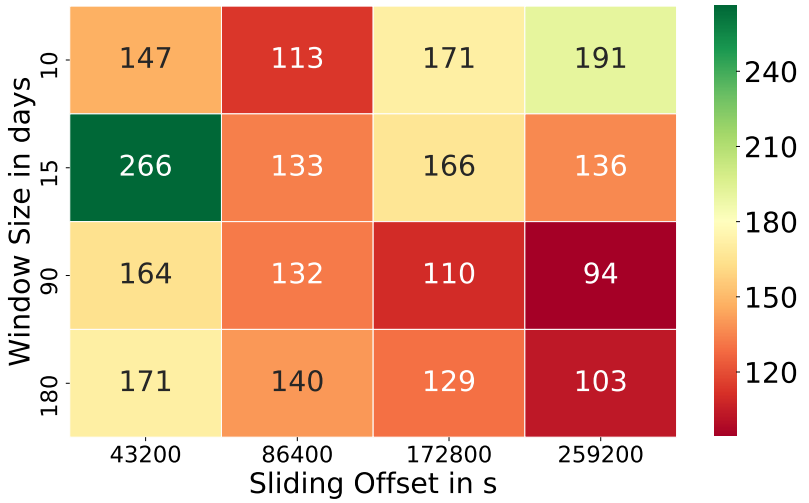


Figure 5.12: Postmortem performance with suggested parameter on wikitalk.

other common use case is to analyse a temporal data postmortem once all the data is known. We showed in this paper how to perform Pagerank efficiently on modern parallel systems by leveraging data representation, incremental algorithms, and different types of parallelism. When using these techniques, a postmortem analysis can be conducted from 50 to 800 times faster than a streaming analysis.

The methods we presented can still be refined: multiple questions remain. We partitioned the temporal data in multi-windows with equal number of graphs, but this may not be the decomposition that minimize memory and work overheads. We only considered Pagerank, but other analysis, like centralities for instance, behave in less regular way when small changes impact the graph. Nowadays, much graph analysis is performed on GPU-enabled system or on distributed memory systems; and extending our techniques to such systems would make temporal analysis more practical.

CHAPTER 6
CONCLUSION

In this dissertation, we addressed how modern computer architecture and application-specific optimization can bring greater performance gain for important graph analysis. Our work shows evidence that graph partitioning and clustering kernel can take the advantage of modern AVX-512 instruction and outperform state-of-the-art algorithms. On the other hand, researchers and data scientists want more than run-time enhancement of the application, they want to find the best configurations and architecture for an application before running the application on the system. To support this significant necessity, one needs to build a performance model that can predict the run time of an application for specific computer architecture. Our dissertation can be done it accurately for iterative sparse matrix-vector multiplication (SpMV). SpMV is one of the most frequently used kernel in modern algebra. As an example, Pagerank calculation in the social web is mainly iterative SpMV between vector and a stochastic matrix that holds the incoming links of vertices. Unlike chapter 3 where we discussed impact of AVX-512 instruction on shared memory architecture, we build SpMV performance model for distributed systems. Predicting run time on the CPU architecture is difficult and especially for the sparse kernel. We provided multiple models that can accurately predict run time for different configurations.

Chapters 3 and 4 only handle static graphs for shared-memory and distributed systems. But graphs like social networks are more likely to show temporal behavior. In chapter 5, we discussed postmortem graph analysis for offline temporal graphs. In our works, we choose Pagerank as a candidate application to show the effectiveness of the postmortem graph analysis. But, we showed postmortem graph analysis can provide a performance boost for applications like Pagerank for the offline temporal graphs compared to the state-of-art naive and streaming version. In particular, we believe applications like betweenness and closeness centrality can also show the similar improvement for postmortem graph analysis. Our works brought out a blueprint for modern graph analyses; how to take advantage of computer architectures and improve the implementation of the applications. This dissertation provides insight to analyze a graph for a given HPC architecture and enhance

its performance.

REFERENCES

[1] S. Porta et al. “Street centrality and densities of retail and services in Bologna, Italy”. In: *Environment and Planning B: Planning and Design* 36.3 (2009), pp. 450–465.

[2] 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.

[3] Ullas Gargi et al. “Large-Scale Community Detection on YouTube for Topic Discovery and Exploration.” In: *ICWSM*. 2011.

[4] Michelle Girvan and Mark EJ Newman. “Community structure in social and biological networks”. In: *PNAS* 99.12 (2002), pp. 7821–7826.

[5] Pall F Jonsson et al. “Cluster analysis of networks generated through homology: automatic identification of important protein communities involved in cancer metastasis”. In: *BMC bioinformatics* 7.1 (2006), p. 2.

[6] Christian Staudt et al. “Static and dynamic aspects of scientific collaboration networks”. In: *Proc. of ASONAM*. 2012, pp. 522–526.

[7] V. Krebs. “Mapping Networks of Terrorist Cells”. In: *Connections* 24 (3 2002).

[8] Joseph E Gonzalez et al. “Powergraph: Distributed graph-parallel computation on natural graphs”. In: *10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*. 2012, pp. 17–30.

[9] David Ediger et al. “Massive streaming data analytics: A case study with clustering coefficients”. In: *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*. IEEE. 2010, pp. 1–8.

[10] Robert Alverson et al. “The Tera computer system”. In: *Proceedings of the 4th International Conference on Supercomputing*. 1990, pp. 1–6.

[11] David A Bader and Kamesh Madduri. “Parallel algorithms for evaluating centrality indices in real-world networks”. In: *2006 International Conference on Parallel Processing (ICPP’06)*. IEEE. 2006, pp. 539–550.

[12] Jacob Nelson et al. “Crunching Large Graphs with Commodity Processors.” In: *HotPar* 11 (2011), pp. 10–10.

[13] Norman E Gibbs, William G Poole Jr, and Paul K Stockmeyer. “An algorithm for reducing the bandwidth and profile of a sparse matrix”. In: *SIAM Journal on Numerical Analysis* 13.2 (1976), pp. 236–250.

[14] Chun Yew Cheong et al. “Hierarchical Parallel Algorithm for Modularity-Based Community Detection Using GPUs”. In: *Proc EuroPar*. 2013, pp. 775–787. ISBN: 978-3-642-40047-6.

[15] David F Gleich. “PageRank beyond the Web”. In: *siam REVIEW* 57.3 (2015), pp. 321–363.

[16] Yousef Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.

[17] Tomas Dytrych et al. “Efficacy of the SU (3) scheme for ab initio large-scale calculations beyond the lightest nuclei”. In: *Computer Physics Communications* 207 (2016), pp. 202–210.

[18] Mariette Awad and Rahul Khanna. “Support vector regression”. In: *Efficient learning machines*. Springer, 2015, pp. 67–80.

[19] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. “Graphs over time: densification laws, shrinking diameters and possible explanations”. In: *Proc. of SIGKDD*. 2005, pp. 177–187.

[20] Lei Yang et al. “Link analysis using time series of web graphs”. In: *Proc. of CIKM*. 2007, pp. 1011–1014.

[21] Vincent D Blondel et al. “Fast unfolding of communities in large networks”. In: *Journal of Statistical Mechanics: Theory and Experiment* 2008.10 (2008), P10008.

[22] Christian Staudt and Henning Meyerhenke. “Engineering Parallel Algorithms for Community Detection in Massive Networks”. In: *IEEE TPDS* 27 (2016), pp. 171–184.

[23] Michael R. Garey and David S. Johnson. *Computers and Intractability*. Freeman, 1979.

[24] David W Matula, George Marble, and Joel D Isaacson. “Graph coloring algorithms”. In: *Graph theory and computing*. Elsevier, 1972, pp. 109–122.

[25] Ümit V. Çatalyürek et al. “Graph Coloring Algorithms for Multi-core and Massively Multithreaded Architectures”. In: *Parallel Computing* 38.10-11 (2012), pp. 576–594.

[26] Erik Saule and Ümit V Çatalyürek. “An early evaluation of the scalability of graph algorithms on the intel mic architecture”. In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. IEEE. 2012, pp. 1629–1639.

[27] Mohammad Almasri and Walid Abu-Sufah. “CCF: An efficient SpMV storage format for AVX512 platforms”. In: *Parallel Computing* 100 (2020), p. 102710.

[28] Arash Ashari et al. “Fast sparse matrix-vector multiplication on GPUs for graph applications”. In: *SC’14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2014, pp. 781–792.

[29] Arash Ashari et al. “An efficient two-dimensional blocking strategy for sparse matrix-vector multiplication on GPUs”. In: *Proceedings of the 28th ACM international conference on Supercomputing*. 2014, pp. 273–282.

[30] Muthu Manikandan Baskaran and Rajesh Bordawekar. “Optimizing sparse matrix-vector multiplication on GPUs”. In: *IBM Research Report RC24704* W0812–047 (2009).

[31] Nathan Bell and Michael Garland. “Implementing sparse matrix-vector multiplication on throughput-oriented processors”. In: *Proceedings of the conference on high performance computing networking, storage and analysis*. 2009, pp. 1–11.

[32] Aydin Buluc et al. “Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication”. In: *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE. 2011, pp. 721–733.

[33] Aydin Buluç et al. “Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks”. In: *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. 2009, pp. 233–244.

[34] Jee W Choi, Amik Singh, and Richard W Vuduc. “Model-driven autotuning of sparse matrix-vector multiply on GPUs”. In: *ACM sigplan notices* 45.5 (2010), pp. 115–126.

[35] Yangdong Deng, Bo David Wang, and Shuai Mu. “Taming irregular EDA applications on GPUs”. In: *Proceedings of the 2009 International Conference on Computer-Aided Design*. 2009, pp. 539–546.

[36] Michael Garland. “Sparse matrix computations on manycore GPU’s”. In: *Proceedings of the 45th annual Design Automation Conference*. 2008, pp. 2–6.

[37] Joseph L Greathouse and Mayank Daga. “Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format”. In: *SC’14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2014, pp. 769–780.

[38] Kornilios Kourtis, Georgios Goumas, and Nectarios Koziris. “Exploiting compression opportunities to improve SpMxV performance on shared memory systems”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 7.3 (2010), pp. 1–31.

[39] Jiajia Li et al. “SMAT: an input adaptive auto-tuner for sparse matrix-vector multiplication”. In: *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 2013, pp. 117–126.

[40] Ruipeng Li and Yousef Saad. “GPU-accelerated preconditioned iterative linear solvers”. In: *The Journal of Supercomputing* 63.2 (2013), pp. 443–466.

[41] Xing Liu et al. “Efficient sparse matrix-vector multiplication on x86-based many-core processors”. In: *Proceedings of the 27th international ACM conference on International conference on supercomputing*. 2013, pp. 273–282.

[42] Bor-Yiing Su and Kurt Keutzer. “clSpMV: A cross-platform OpenCL SpMV framework on GPUs”. In: *Proceedings of the 26th ACM international conference on Supercomputing*. 2012, pp. 353–364.

[43] Wai Teng Tang et al. “Optimizing and auto-tuning scale-free sparse matrix-vector multiplication on Intel Xeon Phi”. In: *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2015, pp. 136–145.

[44] Richard Vuduc, James W Demmel, and Katherine A Yelick. “OSKI: A library of automatically tuned sparse matrix kernels”. In: *Journal of Physics: Conference Series*. Vol. 16. 1. IOP Publishing. 2005, p. 071.

[45] Samuel Williams et al. “Optimization of sparse matrix-vector multiplication on emerging multicore platforms”. In: *SC’07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. IEEE. 2007, pp. 1–12.

[46] Shengen Yan et al. “yaSpMV: Yet another SpMV framework on GPUs”. In: *Acm Sigplan Notices* 49.8 (2014), pp. 107–118.

[47] Rajesh Nishtala et al. “When cache blocking of sparse matrix vector multiply works and why”. In: *Applicable Algebra in Engineering, Communication and Computing* 18.3 (2007), pp. 297–311.

[48] Mehmet Deveci et al. “Parallel graph coloring for manycore architectures”. In: *Proc. IPDPS*. 2016, pp. 892–901.

[49] Gary C Lewandowski. “Practical implementations and applications of graph coloring”. In: (1995).

[50] Satu Elisa Schaeffer. “Graph clustering”. In: *Computer science review* 1.1 (2007), pp. 27–64.

[51] George Karypis and Vipin Kumar. “A Fast and Highly Quality Multilevel Scheme for Partitioning Irregular Graphs”. In: *SIAM SISC* 20.1 (1999), pp. 359–392.

[52] Md. Maruf Hossain and Erik Saule. “Impact of AVX-512 Instructions on Graph Partitioning Problems”. In: *ICPP Workshops 2021: 50th International Conference on Parallel Processing, Virtual Event / Lemont (near Chicago), IL, USA, August 9-12, 2021*. Ed. by Federico Silla and Osni Marques. ACM, 2021, 33:1–33:9.

[53] Jierui Xie, Stephen Kelley, and Boleslaw K Szymanski. “Overlapping community detection in networks: The state-of-the-art and comparative study”. In: *ACM CSUR* 45.4 (2013), p. 43.

[54] Christian L Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. “NetworKit: A tool suite for large-scale complex network analysis”. In: *Network Science* 4.4 (2016), pp. 508–530.

[55] Hao Lu, Mahantesh Halappanavar, and Ananth Kalyanaraman. “Parallel heuristics for scalable community detection”. In: *Parallel Computing* 47 (2015), pp. 19–37.

[56] Mahantesh Halappanavar et al. “Scalable static and dynamic community detection using grappolo”. In: *Proc. IEEE HPEC*. 2017, pp. 1–6.

[57] Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. “Automatically tuning sparse matrix-vector multiplication for GPU architectures”. In: *Proc. HiPEAC*. 2010, pp. 111–125.

[58] H Carter Edwards, Christian R Trott, and Daniel Sunderland. “Kokkos: Enabling manycore performance portability through polymorphic memory access patterns”. In: *JPDC* 74.12 (2014), pp. 3202–3216.

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

[60] Peter Sanders, Christian Schulz, and Dorothea Wagner. “Benchmarking for graph clustering and partitioning”. In: *Encyclopedia of Social Network Analysis and Mining*. Ed. by R. Alhajj and J. Rokne. Springer, 2014.

[61] David A Bader et al. *Graph partitioning and graph clustering*. Vol. 588. American Mathematical Society Providence, RI, 2013.

[62] Bradley Efron and Robert Tibshirani. “Bootstrap methods for standard errors, confidence intervals, and other measures of statistical accuracy”. In: *Statistical science* (1986), pp. 54–75.

[63] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. “R-MAT: A recursive model for graph mining”. In: *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM. 2004, pp. 442–446.

[64] Chun Yew Cheong et al. “Hierarchical parallel algorithm for modularity-based community detection using GPUs”. In: *Proc. EuroPar*. 2013, pp. 775–787.

[65] M. Naim et al. “Community Detection on the GPU”. In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2017, pp. 625–634.

[66] Tze Meng Low et al. “Linear Algebraic Louvain Method in Python”. In: *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2020, pp. 223–226.

[67] Sanaz Gheibi et al. “Cache Efficient Louvain with Local RCM”. In: *2020 IEEE Symposium on Computers and Communications (ISCC)*. IEEE. 2020, pp. 1–6.

[68] Mehmet Deveci et al. “Hypergraph partitioning for multiple communication cost metrics: Model and methods”. In: *JPDC* 77 (2015), pp. 69–83.

[69] George Karypis and Vipin Kumar. “Multilevel graph partitioning schemes”. In: *ICPP (3)*. 1995, pp. 113–122.

[70] Kamer Kaya, Bora Uçar, and Ümit V Çatalyürek. “Analysis of partitioning models and metrics in parallel sparse matrix-vector multiplication”. In: *Proc. PPAM*. Springer. 2013, pp. 174–184.

[71] Ping Guo, Liqiang Wang, and Po Chen. “A performance modeling and optimization analysis tool for sparse matrix-vector multiplication on GPUs”. In: *IEEE TPDS* 25.5 (2013), pp. 1112–1123.

[72] Israt Nisa et al. “Effective machine learning based format selection and performance modeling for SpMV on GPUs”. In: *Proc. IPDPSW*. IEEE. 2018, pp. 1056–1065.

[73] Ping Guo and Changjiang Zhang. “Performance Prediction for CSR-Based SpMV on GPUs Using Machine Learning”. In: *Proc. ICCC*. IEEE. 2018, pp. 1956–1960.

[74] Erik G Boman, Karen D Devine, and Sivasankaran Rajamanickam. “Scalable matrix computations on large scale-free graphs using 2D graph partitioning”. In: *Proc. SuperComputing*. 2013, pp. 1–12.

[75] George Karypis. “METIS and ParMETIS”. In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Boston, MA: Springer US, 2011, pp. 1117–1124. ISBN: 978-0-387-09766-4.

[76] Chih-Chung Chang and Chih-Jen Lin. “Training v-support vector regression: theory and algorithms”. In: *Neural computation* 14.8 (2002), pp. 1959–1977.

[77] John D McCalpin. “STREAM benchmark”. In: *Link: www. cs. virginia. edu/stream/ref. html# what* 22 (1995).

[78] Scott Beamer, Krste Asanovic, and David Patterson. “Locality Exists in Graph Processing: Workload Characterization on an Ivy Bridge Server”. In: *2015 IEEE International Symposium on Workload Characterization*. 2015, pp. 56–65.

[79] Timothy A. Davis and Yifan Hu. “The University of Florida Sparse Matrix Collection”. In: *ACM Trans. Math. Softw.* 38.1 (Dec. 2011).

[80] Emin Nuriyev and Alexey Lastovetsky. “Efficient and accurate selection of optimal collective communication algorithms using analytical performance modeling”. In: *IEEE Access* 9 (2021), pp. 109355–109373.

[81] Tami Carpenter, George Karakostas, and David Shallcross. “Practical issues and algorithms for analyzing terrorist networks”. In: *Proceedings of the western simulation multiconference*. 2002.

[82] Ala Berzinji, Lisa Kaati, and Ahmed Rezine. “Detecting key players in terrorist networks”. In: *2012 European Intelligence and Security Informatics Conference*. IEEE. 2012, pp. 297–302.

[83] Muhammad Ali Masood and Rabeeh Ayaz Abbasi. “Using graph embedding and machine learning to identify rebels on twitter”. In: *Journal of Informetrics* 15.1 (2021), p. 101121.

[84] Rohan Chandra et al. “Forecasting trajectory and behavior of road-agents using spectral clustering in graph-lstms”. In: *IEEE Robotics and Automation Letters* 5.3 (2020), pp. 4882–4890.

[85] Leila Eskandari et al. “T3-Scheduler: A topology and traffic aware two-level scheduler for stream processing systems in a heterogeneous cluster”. In: *Future Generation Computer Systems* 89 (2018), pp. 617–632.

[86] Zainab Abbas et al. “Real-time Traffic Jam Detection and Congestion Reduction Using Streaming Graph Analytics”. In: *2020 IEEE International Conference on Big Data (Big Data)*. IEEE. 2020, pp. 3109–3118.

[87] Zafar Saeed et al. “Text stream to temporal network-a dynamic heartbeat graph to detect emerging events on twitter”. In: *Proc. PAKDD*. 2018, pp. 534–545.

[88] R Devika and V Subramaniyaswamy. “A semantic graph-based keyword extraction model using ranking method on big social data”. In: *Wireless Networks* 27.8 (2021), pp. 5447–5459.

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

[90] Linton C Freeman. “A set of measures of centrality based on betweenness”. In: *Sociometry* (1977), pp. 35–41.

[91] Alex Bavelas. “Communication patterns in task-oriented groups”. In: *The journal of the acoustical society of America* 22.6 (1950), pp. 725–730.

[92] Vincent D Blondel et al. “Fast unfolding of communities in large networks”. In: *Journal of statistical mechanics: theory and experiment* 2008.10 (2008), P10008.

[93] Ahmet Erdem Sariyüce et al. “Streaming algorithms for k-core decomposition”. In: *Proceedings of the VLDB Endowment* 6.6 (2013), pp. 433–444.

[94] Hossein Esfandiari, Silvio Lattanzi, and Vahab Mirrokni. “Parallel and streaming algorithms for k-core decomposition”. In: *International Conference on Machine Learning*. PMLR. 2018, pp. 1397–1406.

[95] Petter Holme and Jari Saramäki. “Temporal networks”. In: *Physics reports* 519.3 (2012), pp. 97–125.

[96] Prasanna Desikan et al. “Incremental Page Rank Computation on Evolving Graphs”. In: *Special Interest Tracks and Posters of WWW*. 2005, 1094–1095. ISBN: 1595930515.

[97] Jason Riedy. “Updating pagerank for streaming graphs”. In: *Proc. IPDPSW*. IEEE. 2016, pp. 877–884.

[98] Oded Green, Robert McColl, and David A. Bader. “A Fast Algorithm for Streaming Betweenness Centrality”. In: *International Conference on Privacy, Security, Risk and Trust and International Conference on Social Computing*. 2012, pp. 11–20.

[99] Ahmet Erdem Sariyüce et al. “Incremental algorithms for closeness centrality”. In: *IEEE Big Data*. 2013, pp. 487–492.

[100] Scott Beamer, Krste Asanović, and David Patterson. “Reducing pagerank communication via propagation blocking”. In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2017, pp. 820–831.

[101] Liaquat Hossain, Shahriar Tanvir Murshed, and Shahadat Uddin. “Communication network dynamics during organizational crisis”. In: *Journal of Informetrics* 7.1 (2013), pp. 16–35.

[102] Andrew Stolman and Kevin Matulef. “HyperHeadTail: a streaming algorithm for estimating the degree distribution of dynamic multigraphs”. In: *Proc. ASONAM*. 2017, pp. 31–39.

[103] Guyue Han and Harish Sethu. “Edge sample and discard: A new algorithm for counting triangles in large dynamic graphs”. In: *Proc. ASONAM*. IEEE. 2017, pp. 44–49.

[104] Xiaowei Chen and John CS Lui. “A unified framework to estimate global and local graphlet counts for streaming graphs”. In: *Proc. ASONAM*. 2017, pp. 131–138.

[105] Kasimir Georg Gabert, Ali Pinar, and Umit Catalyurek. *Finding Dense Areas of Massive Changing Graphs.* Tech. rep. Sandia National Lab.(SNL-NM), 2020.

[106] Eisha Nathan and David A Bader. “A dynamic algorithm for updating katz centrality in graphs”. In: *Proc. ASONAM*. 2017, pp. 149–154.

[107] Ahmet Erdem Sarıyüce et al. “Incremental closeness centrality in distributed memory”. In: *Parallel Computing* 47 (2015), pp. 3–18.

[108] Kasimir Gabert et al. “ElGA: Elastic and Scalable Dynamic Graph Analysis”. In: *Proc. SC*. SC ’21. 2021. ISBN: 9781450384421.

[109] David Gleich, Leonid Zhukov, and Pavel Berkhin. “Fast parallel PageRank: A linear system approach”. In: *Yahoo! Research Technical Report YRL-2004-038, available via http://research. yahoo. com/publication/YRL-2004-038. pdf* 13 (2004), p. 22.

[110] Gianna M Del Corso, Antonio Gulli, and Francesco Romani. “Fast PageRank computation via a sparse linear system”. In: *Internet Mathematics* 2.3 (2005), pp. 251–273.

[111] Md Maruf Hossain and Erik Saule. “Postmortem Graph Analysis on the Temporal Graph”. In: *ICPP poster 2021: 50th International Conference on Parallel Processing*. 2021.

[112] Zheng Zhou et al. “An Out-of-core Eigensolver on SSD-equipped Clusters”. In: *Proc. of IEEE Cluster*. Sept. 2012.

[113] Gordon Erlebacher et al. “Acceleration of Derivative Calculations with Application to Radial Basis Function - Finite-Differences on the Intel MIC Architecture”. In: *Proc. of International Conference on Supercomputing (ICS)*. 2014.