Implementation of Graph Kernels

on

Multi-Core Architecture

by

Srinidhi Renganathan

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved July 2019 by the
Graduate Supervisory Committee:

Chaitali Chakrabarti, Chair
Aviral Shrivastava
Trevor Mudge

ARIZONA STATE UNIVERSITY

August 2019

ABSTRACT

Graphs are one of the key data structures for many real-world computing applications such as machine learning, social networks, genomics etc. The main challenges of graph processing include difficulty in parallelizing the workload that results in workload imbalance, poor memory locality and very large number of memory accesses. This causes large-scale graph processing to be very expensive.

This thesis presents implementation of a select set of graph kernels on a multi-core architecture, Transmuter. The kernels are Breadth-First Search (BFS), Page Rank (PR), and Single Source Shortest Path (SSSP). Transmuter is a multi-tiled architecture with 4 tiles and 16 general processing elements (GPE) per tile that supports a two level cache hierarchy. All graph processing kernels have been implemented on Transmuter using Gem5 architectural simulator.

The key pre-processing steps in improving the performance are static partitioning by destination and balancing the workload among the processing cores. Results obtained by processing graphs that are partitioned against un-partitioned graphs show almost 3x improvement in performance. Choice of data structure also plays an important role in the amount of storage space consumed and the amount of synchronization required in a parallel implementation. Here the compressed sparse column data format was used. BFS and SSSP are frontier-based algorithms where a frontier represents a subset of vertices that are active during the current iteration. They were implemented using the Boolean frontier array data structure. PR is an iterative algorithm where all vertices are active at all times.

The performance of the different Transmuter implementations for the 14nm node were evaluated based on metrics such as power consumption (Watt), Giga Operations Per Second(GOPS), GOPS/Watt and L1/L2 cache misses. GOPS/W numbers for graphs with 10k nodes and 10k edges is 33 for BFS, 477 for PR and 10 for SSSP.

Frontier-based algorithms have much lower GOPS/W compared to iterative algorithms such as PR. This is because all nodes in Page Rank are active at all points in time. For all three kernel implementations, the L1 cache miss rates are quite low while the L2 cache hit rates are high.

ACKNOWLEDGMENTS

I would like to extend my sincere thanks and gratitude to my thesis advisor Dr. Chaitali Chakrabarti for her constant support and guidance throughout my work. I am grateful to my committee members Dr. Trevor Mudge and Dr. Aviral Shrivastava for providing me feedback on my research. I am indebted to Subhankar Pal from the University of Michigan, Jiawen Sun from the University of Edinburgh and Jian Zhou from Arizona State University for their help throughout my work.

TABLE OF CONTENTS

Page

CHAPTER Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Chapter 1

INTRODUCTION

Graph processing is integral to a very large range of applications, including social network analysis, data mining, machine learning, natural language processing etc. [**?**], [7], [10]. Real-world graphs are massive and contain millions of vertices and edges. The scale of these graphs poses hard challenges for efficient graph processing. In an attempt to speed up processing, this thesis addresses the problem of implementing several graph processing kernels on a parallel computing platform.

1.1 Challenges

Graph processing is a data-driven computation process. Since contemporary graph data sets typically have millions of vertices and edges, the number of computations that need to be performed are enormous. In addition, these graphs have sparse connections between vertices and irregular relationship between nodes, thereby posing serious challenges from an implementation perspective. Below we outline some of the challenges:

1. **Random access in graph processing**: Graph algorithms typically have irregular data access patterns [4]. Basically, the flow of data occurs through edges, requiring pointer-chasing to destination vertices leading to random access of memory. If the vertex property updates are made by iterating the edges of the graph, then $O(|E|)$ random read and/or random write accesses are likely to occur. For large graphs, the number of random accesses is thus very high.

2. **Imbalance in computation workload**: Graph algorithms have low compu-

tational to communication ratio. Typically, only a few instructions are executed for each node. Every graph computation is based on tracing the incoming or outgoing edges of the nodes and updating the property values. The work per node is unbalanced making it challenging to efficiently parallelize graph processing algorithms[12].

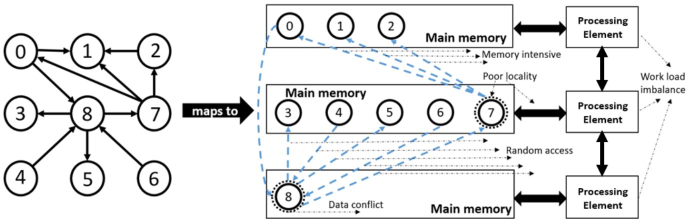


Figure 1.1: Challenges in Graph Processing (Adapted From [4])

3. **Poor memory locality**: In graph implementations, it is necessary to read/write values of vertices associated with the edges being processed. Thus random accesses may be required to retrieve values from locations scattered in memory. This results in poor spatial and temporal locality. Hardware caches are thus not suitable for graph structures.

4. **Memory-intensive**: Graph traversals involve a very large number of memory accesses. Since memory locality is low, cache accesses become inefficient and high latency memory accesses become significant in real-world graphs [14]. Graph algorithms are memory-bound [16], and memory is a bigger bottleneck in parallel implementation due to multiple cores competing for bandwidth.

5. **Large number of data-conflicts**: In a parallel execution model, different processing cores might face data conflicts when the node values are updated simultaneously. When the number of nodes and edges in the graph is large,

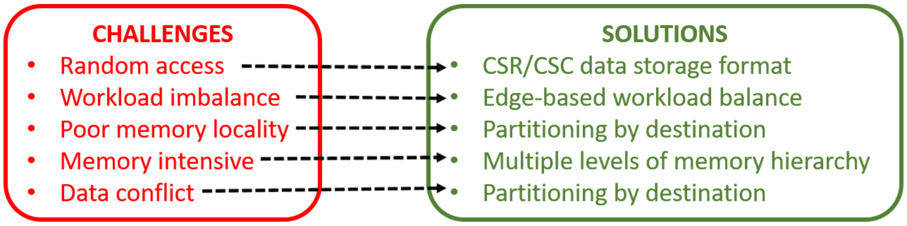

CHALLENGES
Random access
Workload imbalance
Poor memory locality
Memory intensive
Data conflict
SOLUTIONS
CSR/CSC data storage format
Edge-based workload balance
Partitioning by destination
Multiple levels of memory hierarchy
Partitioning by destination

Figure 1.2: Challenges in Graph Processing and Proposed Solutions

this becomes a major concern. The data conflict problem can be mitigated by utilizing expensive synchronization mechanisms or effective partitioning of tasks among the cores along with careful design of the data access pattern.

1.2 Problem Definition

Our goal is to combat the challenges of graph processing algorithms, namely the presence of random accesses and data conflicts, imbalance in workload and poor memory locality through a combination of algorithmic and architectural solutions. These include utilizing a Compressed Sparse Column (CSC) data storage format, designing a partitioning by destination scheme, implementing an edge-based workload balancing algorithm and supporting multiple levels of memory hierarchy. Figure 1.2 summarizes the challenges and the techniques to mitigate them. We demonstrate the use of these techniques in our implementation of graph processing kernels on a multi-core architecture.

1.3 Existing Work

Parallel graph processing has been addressed by many researchers in recent years. Pregel [13] is a vertex-centric framework which supports distributed graph processing. It has been designed mainly for sparse graphs and utilizes a message-passing model. The cost of communication is a major overhead in the case of dense graphs.

GraphLab [11] is a vertex-centric shared-memory based graph analytics framework. It supports a set of concurrent access models such as full consistency, edge consistency and vertex consistency. The GraphLab abstraction helps express asynchronous, dynamic, parallel-graph computation while still maintaining data consistency. Distributed GraphLab [10] is an extension of the GraphLab model which is designed to simplify large-scale graph data processing especially for machine learning applications. It also includes fault-tolerant capabilities.

GraphMat [18] is an efficient vertex programming, single-node, multi-core framework. It maps vertex programs to generalized sparse matrix-vector calculations.

GraphChi [8] is a disk-based system which breaks a large graph into smaller parts and incorporates parallel-sliding window method to implement very large graphs on a personal computer. The parallel-sliding window results in fewer sequential reads/writes enabling high performance.

GraphIA [9] is the first in-situ architecture for large-scale graph processing based on DRAM. To make the system scalable, a scaling ring interconnection topology and communication scheme have been devised. Interval-block partitioning method has been employed to enable graph processing on multiple chips with balanced workload.

Tesseract [3] is a programmable accelerator for in-memory graph processing design. It shows how graph processing challenges such as random accesses, poor memory locality, memory access not being able to overlap with computation, can be addressed through in-memory processing.

Graphicionado [6] presents a domain-specific accelerator for high-performance, energy-efficient processing of a large set of graph algorithms. It is based on a vertex-centric programming paradigm which makes efficient use of a specialized hardware pipeline. To overcome high latency off-chip memory accesses, prefetching has been incorporated.

1.4 Thesis Contributions

This thesis presents an efficient implementation of several key graph processing kernels on a re-configurable multi-core architecture, Transmuter, that was designed at the University of Michigan. The architecture is modeled using the Gem5 simulator. It is a multi-tiled architecture, where each tile consists of multiple processing elements. The GPEs within a tile store local data in L1 cache banks while tiles share data via the L2 data banks. Each processing element is an ARM core which implements Thumb ISA.

The key elements of this work are: (i) graph pre-processing which involves static partitioning by destination and balancing workload based on the degree of a node, (ii) mapping of Breadth-First Search (BFS), Page Rank (PR) algorithm and Single Source Shortest Path (SSSP) on Transmuter, and (iii) evaluation and analysis of performance of the Transmuter implementations on graphs of varying sizes and sparsities.

Graph Pre-Processing: Partitioning and workload balancing are important pre-processing kernels when mapping graph applications onto a parallel multi-core architecture. The unpredictable nature of graphs, the large number of data conflicts and poor memory locality problems can be mitigated when graphs are partitioned efficiently. We also found that partitioning by destination scheme is a promising solution to reduce the number of data conflicts. Since each processing core is assigned a set of nodes, update of a destination node property avoids multiple processing cores writing to the same node at the same time, and is hence desirable. In order to distribute work equally between the processing cores, we implemented an edge-based workload balancing scheme.

Mapping kernels on Transmuter: We selected a set of most commonly used kernels in graph applications, namely BFS, PR and SSP. We implemented these kernels on Gem5 for a Transmuter configuration with 4 tiles and 16 processing elements per tile. In order to reduce data conflicts and execution time overhead due to synchronization mechanisms in parallel implementations, we implemented partitioning by destination and evaluated the performance of the kernels. We implemented the kernels using a frontier based approach. In such an approach, the destination vertices from active nodes are traced and their node properties updated. High-level generic pseudo-codes of the work carried out by processing cores for frontier-based kernels such as BFS and SSSP as well as iterative kernels such as Page Rank have been presented.

Performance evaluation: All Gem5 implementations have been evaluated for 14nm technology node with respect to time, Giga Operations Per Second(GOPS), power consumption(W), GOPS/W and L1 and L2 cache hit rates. These performance metrics have been analyzed for graphs of different sizes and sparsities. For a graph with 10k nodes and 10k edges, GOPS/W numbers for BFS, PR and SSSP are 33, 477, and 10, respectively. PR has a large number of computations per iteration, compared to BFS and SSSP kernels, and hence its GOPS/W number is higher. The GOPS/W for SSSP is much lower than BFS because in every iteration, only one node that has minimum distance value is active and the neighboring nodes from only this node are visited. The L1 hit rates are found to be close to 95% for BFS, about 75% for PR and around 80% for SSSP. For all three implementations, the L2 hit rates are significantly smaller at around $20\% - 40\%$.

1.5 Thesis Organization

- **Chapter 2** discusses graph concepts such as graph layouts, forward and backward graph traversal schemes and the three kernels that have been mapped on Transmuter. This chapter also talks about the graph partitioning scheme employed and also how the workload is balanced among the processing cores. A brief description of the Transmuter multi-core architecture has also been included.

- **Chapter 3**: This chapter includes the detailed graph processing workflow from generating random graphs to processing the kernels on Transmuter. Transmuter configuration and pseudo code implemented by the local control processor and the general processing units have also been explained.

- **Chapter 4**: This chapter discusses the performance results of the implementation of BFS, Page Rank and SSSP using a CSC data structure, with a Boolean frontier array.

- **Chapter 5**: Chapter 5 summarizes the thesis and also presents work that will be done in the near future.

Chapter 2

BACKGROUND

A graph is defined as a set of vertices and edges, where V is the set of vertices and an edge, which models the connection between two vertices, is represented by $E \subset V \times V$. This chapter introduces basics of graph processing. It starts with graph layout in Section 2.1. The Compressed Sparse Row, Compressed Sparse Column and Coordinate list based layouts, and the storage requirements for each of these layouts, are discussed. Next, different types of graph traversals such as forward and backward traversals are presented in Section 2.2. The three problems that have been mapped onto Transmuter, namely BFS, PR and SSSP, are presented in Section 2.3. Graph partitioning which is an essential step in graph processing, is introduced in Section 2.4. Finally, the Transmuter architecture is presented in Section 2.5.

2.1 Graph Layout

The way graphs are stored and the corresponding storage size, greatly influences the performance of graph processing algorithms. Since real-world graphs are very large in size and memory-bound, badly designed graph storage has an adverse effect on their performance. There are three common data-storage layouts for graphs: Compressed Sparse Column (CSC), Compressed Sparse Row (CSR) and Coordinate list (COO). The CSR and CSC formats provide an index into the edge list, allowing efficient accesses to the neighboring nodes.

- **Coordinate List (COO)**: This format stores edges as a pair of source and destination nodes. Every iteration involves traversal of edges to check if the

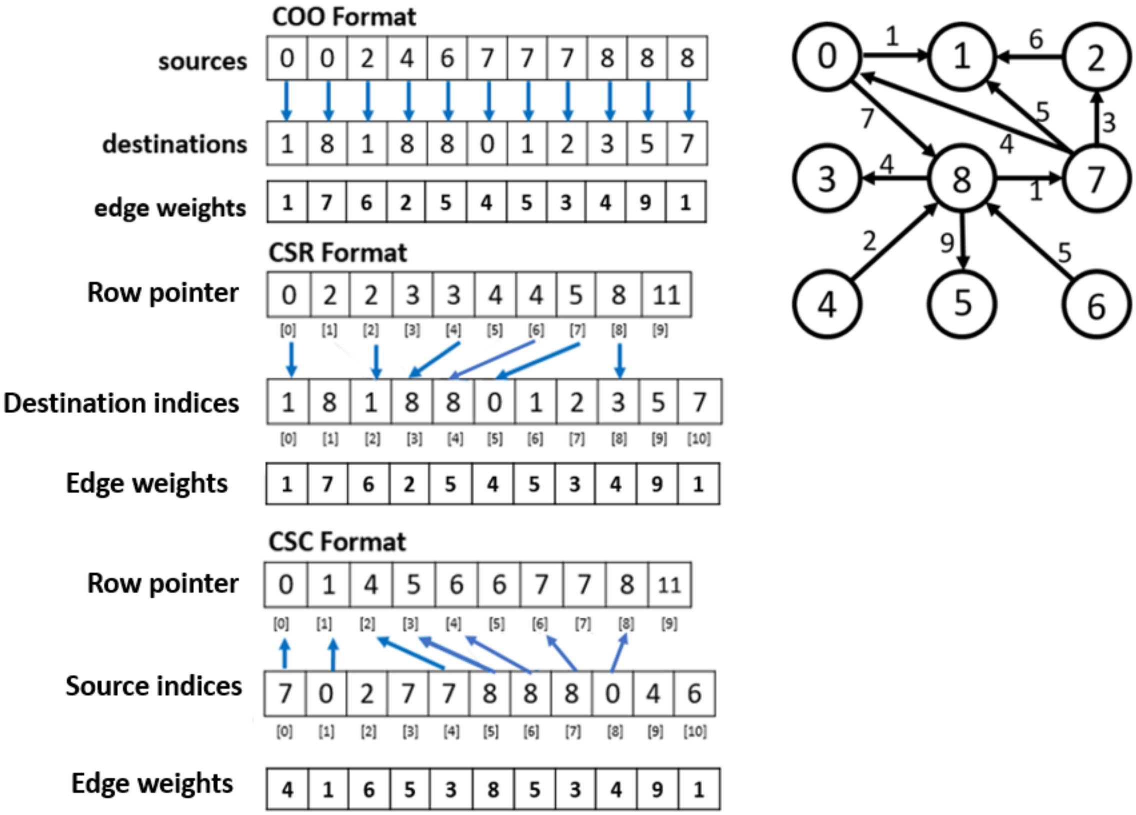


Figure 2.1: Example Graph Layouts

source node associated with an edge is active or not. If it is active, the node property values are computed, and the destination nodes become the new set of active vertices for the next iteration. In a parallel execution model, when different cores process different partitions, processing multiple edges with the same destination simultaneously result in an update of the same vertex property. This results in data race condition that has to be taken care of.

- **Compressed Sparse Row (CSR)**: CSR requires two arrays: (i) destination indices array which store the destination IDs and a (ii) row pointer array (rowptr). Every index position in the rowptr array corresponds to a source node and every entry in the rowptr array corresponds to the start index in the destination indices array where the corresponding destination IDs are stored. When multiple processing cores update the same destination vertex property, there could be a data-race condition, as in COO. To prevent data-race, synchronization mechanisms such as mutex lock are required.

- **Compressed Sparse Column (CSC)**: CSC also has two arrays (i) source indices array whch store the source IDs and a (ii) row pointer array(rowptr). Every index in the rowptr array corresponds to a destination node, and every entry in the rowptr array corresponds to the start index in the source indices array where the corresponding source IDs are stored. When using this format, the algorithm traverses the destination vertices, retrieves active source node properties sequentially by tracing the corresponding in-edges.

2.1.1 Graph Storage Size

CSR and CSC are more efficient in storing graphs in a compact format than the COO format. CSR and CSC formats also provide some sequential accesses on the indices array, which allows for better cache performance.

Storage required for a weight graph using CSR/CSC format is (number of nodes * bytes required to store every element in the rowptr array) + (number of edges * bytes required to store every element in the indices array) + (number of edges * bytes required to store every element in the edge weights array). In comparison, the storage required for a weighted graph using COO format is (number of edges * bytes required to store every element in the sources array) + (number of edges * bytes required to store every element in the destinations array) + (number of edges * bytes required to store every element in the edge weights array). Consider an example,graph with integer weights, $10,000$ nodes and $100,000$ edges. The CSR and CSC formats require 840 kB compared to 1.2 MB required by COO.

2.2 Graph Traversal

There are two different ways to traverse a graph when using the Compressed Sparse Row (CSR) or the Compressed Sparse Column (CSC) format - forward (push) and backward (pull).

- **Forward (Push)**: This method of traversal iterates the frontier and pushes the updated values computed in that iteration to all the target nodes by tracing the out-edges from the active nodes. These target nodes are made active for the next iteration. CSR format is suitable for forward traversal.
- **Backward (Pull)**: Backward traversal scheme checks all the nodes in the graph, traces the corresponding incoming edges, retrieves attribute values from the active nodes, computes updated node values and the frontier for the next iteration. CSC format inherently supports this mode of traversal.

In COO data format, processing multiple edges with the same destination vertices, could result in data-race condition. Appropriate synchronization mechanisms should be incorporated to prevent data-race hassles.

2.3 Graph Processing Kernels

Among the kernels found in applications, such as machine learning, graph statistics and graph traversal, BFS, PR and SSSP are quite common. These form the core building blocks in graph processing applications and are studied here.

2.3.1 Breadth-First Search

Breadth-First Search (BFS) is a very popular graph search algorithm that is also part of the Graph500 Benchmark. It starts from a given initial node and iteratively visits unexplored neighboring nodes [19]. The predecessor of every node is updated. This is done so as to be able to trace back the path to the source node. The BFS algorithm is a frontier-based algorithm - where the frontier is a subset of vertices that are active at any point in time. The active nodes in every iteration follow a wavefront pattern originating from the initial start node. Every time an unexplored node is encountered, this node is updated and becomes part of the new frontier.

2.3.2 Page Rank

It is one of the most popular problems which calculates the scores of websites [2]. It is an iterative algorithm that ranks web pages based on a metric such as popularity. Web pages are represented as vertices and hyperlinks are denoted as edges. The algorithm calculates the probability that a walk through the hyperlinks would end in a particular page. It is not a frontier-based algorithm; all nodes in the graph are active at all points in time.

2.3.3 Single Source Shortest Path

Djikstra's graph traversal algorithm is used for finding the shortest paths between a source node and all other nodes in the graph [1]. In every iteration, the un-visited node that is marked with the smallest distance is picked. Distance from this node to all of its unvisited neighbors are calculated and the neighbor's distance is updated if the newly calculated distance is smaller. A node is considered visited after all neighbors of this node are visited.

Algorithm 1 Breadth-First Search Algorithm

Input: Partitioned graph $G_{p(i)} = (V_i, E_i)$ for i = 0,1,2.....p-1; p : no. of partitions

Output: Breadth-First traversal path for any given node

```
$num\_nodes\_visited \leftarrow 0$
while $num\_nodes\_visited \leq MAX\_NODES\_TO\_VISIT$ do
     for $dest \in partition\_range$ do
          for $src\_index \leftarrow ind\_ptr[dest]$ to $ind\_ptr[dest + 1]$ do
               $src \leftarrow edges[src\_index]$
               if $frontier[src] is\ true$ and $bfs\_pred[dest] = -1$ then
                    $bfs\_pred[dest] \leftarrow src$
                    $frontier[src] \leftarrow 0$
                    $frontier[dest] \leftarrow 1$
                    Increment num_nodes_visited by 1
          end
     end
     if $num\_nodes\_visited$ reaches $MAX\_NODES\_TO\_VISIT$ or none of nodes active then
          Quit
     end
```

Algorithm 2 Page Rank Algorithm

```
Input: Partitioned graph $G_{p(i)} = (V_i, E_i)$ for i = 0,1,2.....p-1; p : no. of
partitions Output: Page rank value for every node
  $num\_iterations \leftarrow 0$
  while $num\_iterations \leq MAX\_ITERATIONS$ do
      for $dest \in partition\_range$ do
          for $src\_index \leftarrow ind\_ptr[dest]$ to $ind\_ptr[dest + 1]$ do
              $src \leftarrow indices[src\_index]$
              Update $node\_property[dest]$ to
        $node\_property[dest] + (node\_property[src]/out\_degree[src])$
          end
      end
      Increment $num\_iterations$ by 1
  end
```

2.3.4 Termination Condition

With millions of nodes and edges in real-world graph data sets, it is impractical to terminate the algorithm only after processing the entire graph. So, the most commonly used termination conditions are when the number of nodes visited reaches a certain number, or when the current frontier nodes do not have any children, or the number of iterations reaches a fixed value, or difference between the updated destination property and the original value is less than 5%.

Algorithm 3 Single-Source Shortest Path algorithm

```
Input: Partitioned graph $G_{p(i)} = (V_i, E_i)$ for i = 0,1,2.....p-1; p : no. of partitions
Output: Find the shortest distance of nodes from source
Initialisation:    All nodes set to 'unvisited'
                   Frontier array to 0 except frontier[START_NODE] = 1
                   Vertex prop to infinity except vertex_prop [source_node] to 0
  $num\_nodes\_visited \leftarrow 0$
  while $num\_nodes\_visited \leq MAX\_NODES\_TO\_VISIT$ do
      Pick node with minimum vertex_prop
      for $dest \in partition\_range$ do
          for $src\_index \leftarrow ind\_ptr[dest]$ to $ind\_ptr[dest + 1]$ do
              if $frontier[src] is\ true$ then
                  New $vertex\_prop[dest] = vertex\_prop[src] + edge\_value[dest]$
                      if $new\ vertex\_prop[dest] \leq vertex\_prop[dest]$ then
                          $vertex\_prop[dest] = newvertex\_prop[dest]$
          end
      end
      Increment $num\_nodes\_visited$ by 1
    end
```

2.4 Graph Partitioning

An important pre-processing step in parallelizing graph processing problems is graph partitioning. This step enables the whole graph to be distributed across multiple processing cores. The graph partitioning step allows for efficient use of local L1 caches and also reduces synchronization overhead.

2.4.1 Partitioning by source/destination

Graphs can be partitioned based on the vertex list or edge list. Vertex list partitioning results in some edges being spread out and thus the edges may cross partitions. This requires additional synchronization mechanisms when processing these edges. Alternatively, one can divide the edge list to balance workload amongst the processing elements. Since, computations performed in many graph kernels are proportional to the number of edges processed, we choose to implement edge-list based partitioning scheme. Here the edges in a graph G(V,E) are split into K non-overlapping partitions by $\mathcal{P} = \mathcal{P}_i$, i=0, 1, 2... k-1, $\mathcal{P} \subset$ E and $\bigcup_{i=0}^{k-1} \mathcal{P}_i =$ E for all i and $\mathcal{P}_i \cap \mathcal{P}_j = \varphi$ for all i $\neq$ j.

Graphs can also be partitioned by source or destination [17].

1. Partitioning by source: All the destination nodes corresponding to the outgoing edges of a vertex are grouped. $G_p^{source} = \big(\text{V}, \ \big\{\text{u,v}\big\} \ \epsilon \ \text{G}\cdot\text{E} \ ; \ \text{v} \ \epsilon \ \mathcal{P}\big)$
2. Partitioning by destination: All the source nodes corresponding to the incoming edges of a vertex are grouped. $G_p^{destination} = \big(\text{V}, \ \big\{\text{u,v}\big\} \ \epsilon \ \text{G}\cdot\text{E} \ ; \ \text{u} \ \epsilon \ \mathcal{P}\big)$

Partitioning by source: This scheme results in the set of source nodes in a partition being unique. In other words, no two partitions have the same source node. However, the same destination nodes can be part of multiple partitions. In the example shown in Fig.2.2, source nodes with IDs 0,2,4,6,7 and 8 are grouped. It can

	Source	Destination
Source 0	0	1
	0	7
	0	8
Source 2	2	1
Source 4	4	8
Source 6	6	8
Source 7	7	1
	7	2
Source 8	8	3
	8	5
	8	7

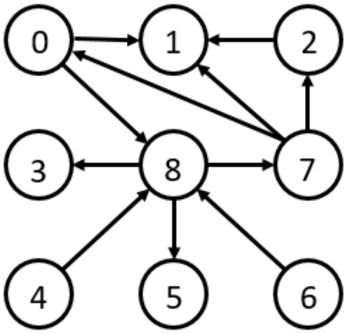


Figure 2.2: Partitioning By Source

be seen that destination node 1 is present in source 0 as well as source 2 partitions. In a parallel implementation, where each core is assigned one of these partitions, all out-going edges corresponding to the partition's source nodes are processed. The source vertices can be usually accessed in a sequential manner when using the CSR format. Multiple edges with the same destination vertices could result in properties of the same vertex being updated. This results in data race condition, which in turn necessitates the use of mutex and lock.

Partitioning by destination: Each partition is grouped by destination nodes and so no two partitions share the same destination node. In the example shown in Fig. 2.3, destination nodes with IDs 0,1,2,3,5,7 and 8 are grouped. It can be seen that the source node 7 is present in multiple partitions with destination node 0, 1 and 2. Partitioning by destination scheme allows node property update without the need to use additional synchronization overhead such as mutex and lock. Graphicionado[6]

Algorithm 4 Edge-Based Partitioning By Destination And Workload Balancing

Input : Graph G = (V,E)

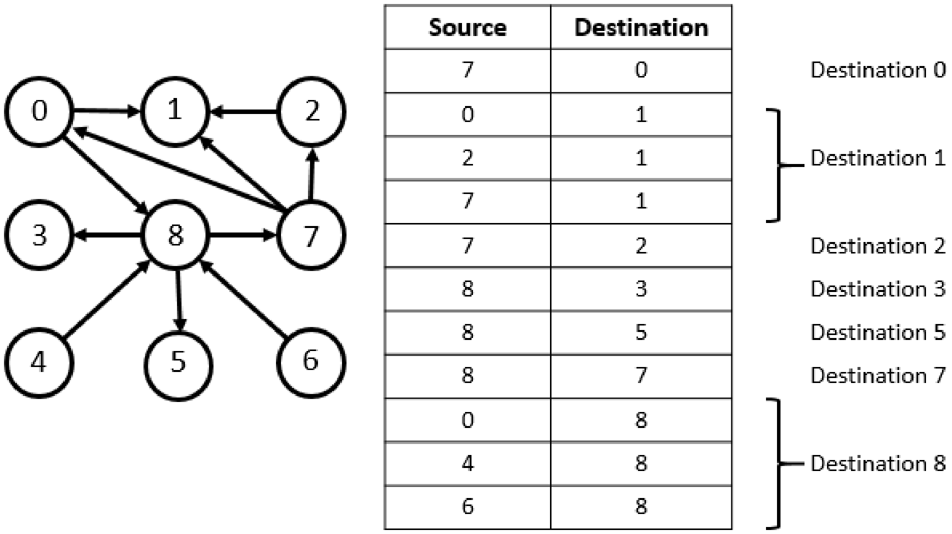
Output : Partitioned graph $G_{p(i)} = (V_i, E_i)$ for i = 0,1,2.....p-1; p : no. of partitions

```
$i \leftarrow 0$
$in\_edges\_in\_partition_i \leftarrow 0$
$avg\_edges\_per\_partition \leftarrow num\_edges/num\_GPEs$
foreach $v \in V$ do
    $in\_edges\_in\_partition_i \leftarrow in\_edges\_in\_partition_i + in\_degree[v]$
    if $in\_edges\_in\_partition_i > avg\_edges\_per\_partition$ then
        $i \leftarrow i + 1$
        $in\_edges\_in\_partition_i \leftarrow 0$
    Add vertex v to partition i
end
```

adopts this partitioning method to ensure each partition can be fitted to the scratchpad memory.

2.4.2 Edge-load balancing

In this thesis, an edge-load balancing scheme has been implemented in the graph pre-processing step. Here every partition has an approximately equal number of edges while still keeping destination by partition in-tact. As shown in the example shown in Fig.2.4, if there are 4 processing elements(GPEs) and total number of edges in the graph is 11, then the number of edges that each GPE processes is 11/4 = 3,



Source	Destination	
7	0	Destination 0
0	1	Destination 1
2	1	
7	1	
7	2	Destination 2
8	3	Destination 3
8	5	Destination 5
8	7	Destination 7
0	8	Destination 8
4	8	
6	8	

Figure 2.3: Partitioning By Destination

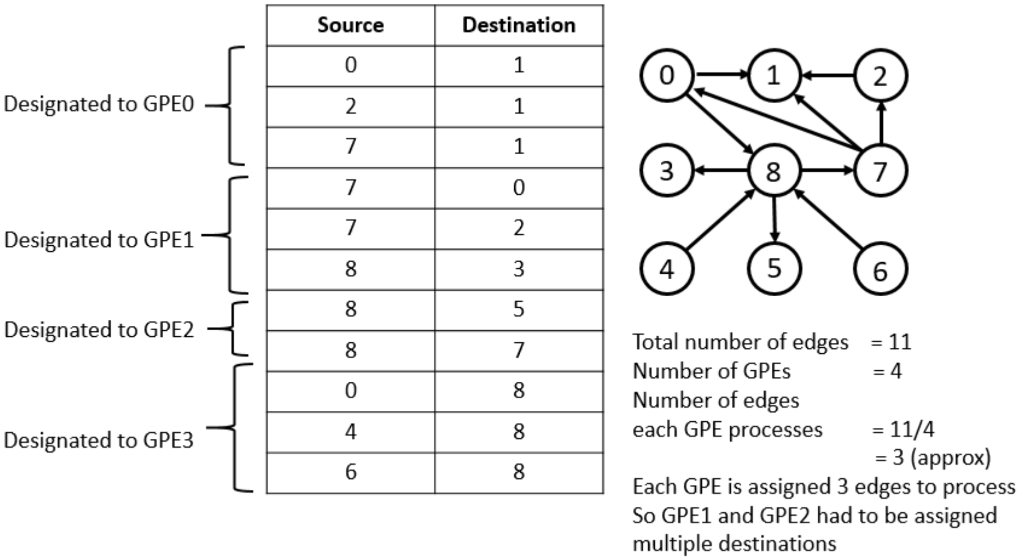
approximately. Thus, GPEs 1 and 2 are assigned multiple destinations for processing.

2.4.3 Static and Dynamic Partitioning

There are two different ways to partition graph in the pre-processing step - Static Partitioning and Dynamic Partitioning.

Static Partitioning: This partitioning scheme splits the whole graph into partitions once in the pre-processing step. These sub-graphs are delegated to each GPE and every GPE works on the same partition the entire time. This thesis implements static partitioning.

Dynamic Partitioning: This is a much more sophisticated method of graph partitioning. Unlike static partitioning, this scheme looks at the frontier array after every round of processing and based on the number of nodes that are active, the nodes are re-distributed. It has the overhead of partitioning after every single iteration and communication of partitioned graph to the processing elements.



Source	Destination
0	1
2	1
7	1
7	0
7	2
8	3
8	5
8	7
0	8
4	8
6	8

Total number of edges = 11
Number of GPEs = 4
Number of edges
each GPE processes = 11/4
= 3 (approx)
Each GPE is assigned 3 edges to process
So GPE1 and GPE2 had to be assigned
multiple destinations

Figure 2.4: Edge-List Based Workload Balancing

2.5 Multicore Architecture - Transmuter

Transmuter is a coarse-grained, reconfigurable architecture designed at the University of Michigan by Subhankar Pal and others. It consists of multiple in-order General-Purpose Processing Elements (GPE) organized into tiles and a memory-crossbar hierarchy connected to a DDR 3 memory controller. Fig. 2.5 shows the different components of Transmuter. They are as follows:

General-Purpose Processing Element (GPE): A GPE is a single-issue, 4-stage, in-order core that is optimized for energy-efficient computation. It has significantly lesser silicon footprint and has lower power consumption than modern processors. This allows Transmuter to be built with many such cores. Each GPE executes instructions in ARM Thumb ISA. They do not support SIMD vector instructions but have a Floating Point Unit, capable of executing single-precision floating-point operations.

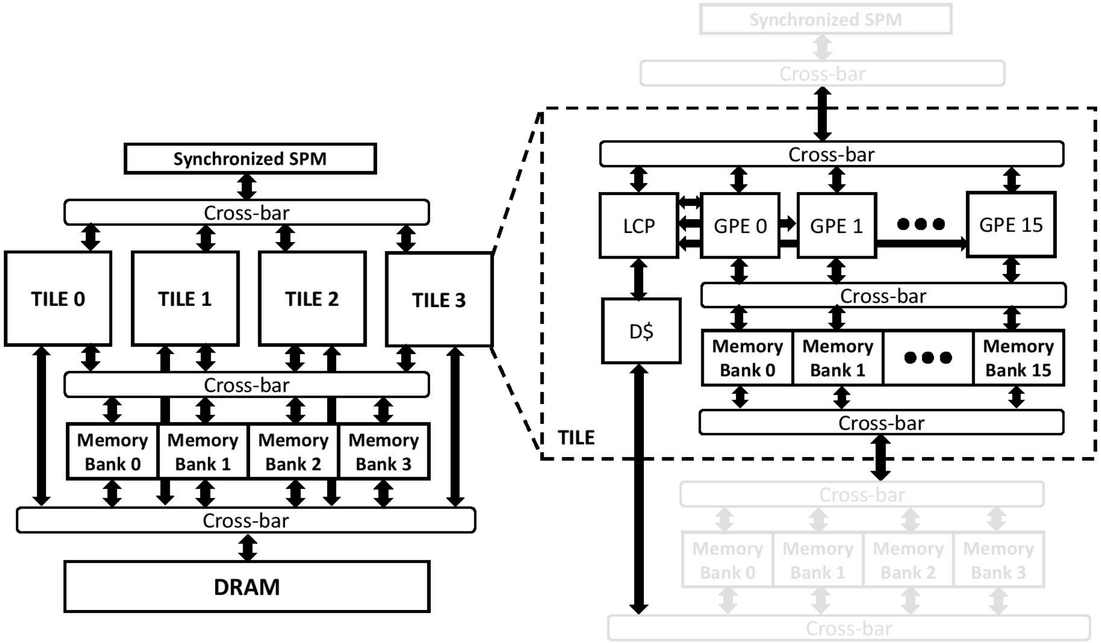


Figure 2.5: Transmuter Architecture (adapted from [15])

Local Control Processor: The GPEs within a tile are managed and scheduled to work through the Local Control Processor(LCP). Each LCP has the same micro-architecture as GPE and also executes instructions from the Thumb ISA.

Work/Status Queue: Every tile has a set of hardware FIFO work queues through which 32-bit packets are pushed to the GPEs by the LCP. These work queues are private to every GPE. Data can be pushed to or popped from the work queue using a single load, store instruction. The load is blocked if the queue is empty and similarly store is blocked when the queue is full.

Reconfigurable Cache: The architecture has two levels of reconfigurable data

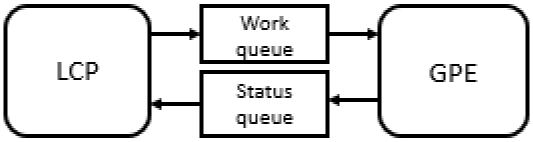


Figure 2.6: Work Queue/Status Queue

cache (R-D cache) banks - L1 and L2. Every level consists of an SRAM memory array connected to the control units. They can operate in the following modes:

1. **Cache**: Every SRAM memory bank functions as a traditional write-back cache with LRU replacement policy.

2. **Scratchpad**: The SRAM memory bank can be configured as software-managed scratchpad.

Reconfigurable Crossbar: An m × n crossbar allows for *m* requesters access to *n* resources. The crossbars within a tile (L1) and outside the tiles (L2) supports the following modes of reconfiguration.

- ARBITRATE: If two or more requesters try to the same resource, the crossbar grants only one of them access using a Least-Recently Granted (LRG) policy. This is the traditional crossbar configuration where a penalty for arbitration is incurred for each access and requests to the same memory bank get serialized.

- TRANSPARENT: In this mode, requester *i* has direct access to its corresponding resource. GPEs can access adjacent (private) memory banks with no arbitration latency.

- ROTATE: The crossbar port connections rotate through a set of pre-programmed patterns that define static links between a requester and a resource.

Sychronization ScratchPad: Transmuter has a global scratchpad memory for synchronization. This allows for the implementation of software coherence and standard primitives such as locks, condition variables, barriers, and semaphores. Most algorithm implementations only need to use barriers for synchronization before and after a kernel and the need to synchronize during execution can usually be avoided.

Chapter 3

GRAPH PROCESSING ON TRANSMUTER

3.1 Transmuter Configuration

The Transmuter configuration used in the thesis has 4 tiles with 16 GPEs per tile. Each GPE in a tile has L1 memory bank of size 4 kB and so a total of 4 kB ×16= 64 kB per tile. Each tile has L2 memory bank of size 64 kB and a total of 256 kB across 4 tiles. While the L1 and L2 memory banks can operate in the cache or scratchpad mode, we choose to use the cache mode. We use unified-shared cache configuration for both L1 and L1 caches. In this mode of operation, all the GPEs in a tile and all tiles see L1 and L2 R-DCache banks as shared cache banks; the corresponding reconfiguration crossbar is set to Arbitrate mode. When many hardware units request access to the same resource, requests get serialized. The priority amongst these requesters is handled using a LRG (Least Recently Granted) policy. The overhead of arbitration process is one clock cycle. Transmuter configuration parameters have been listed in Table 3.1.

Real-world graphs have unpredictable accesses to data at different points in time. So use of shared cache mode and storing graphs using CSC or CSR formats help improve cache performance. In the CSC format, access to the rowptr array is random while source indices are accessed sequentially. The sequential access aids in better management of cache data by providing spatial locality when obtaining data from neighbours of a particular node.

In the cache configuration, each memory bank is used as a non-blocking, write back cache with LRU(Least Recently Used) replacement policy. The caches must

General Processing Element(GPE)	1-issue, 4-stage, in-order CPU @ 1.0 GHz
Number of GPEs per tile	16
Number of tiles	4
Number of L1 cache banks per tiles	16
Total number of L2 cache banks	4
Size of L1 cache bank	4 kB
Size of L2 cache bank	256 kB
L1 configuration	Unified shared-cache
L2 configuration	Unified shared-cache
Main memory size	4 GB
Work/Status queue	32 b 4-entry FIFO buffer
Sync SPM	16 kB, 1-ported
R-DCache (per bank)	CACHE mode: 4 kB, 4-way set-associative, 1-ported

Table 3.1: Transmuter Configuration Parameters

be flushed after every round or phase of computation to make sure data in different memory hierarchies are consistent.

3.2 Graph Processing Overview

Figure 3.1 details the workflow for processing graphs on Transmuter. The steps in this workflow consists of generating a random graph using the Python Random Graph Generation tool, the graph pre-processing step and graph processing on Transmuter.

The random graph generation step takes as input configurable parameters such as number of nodes, number of edges, the file format of the output. The output edge list file is used as input to the pre-processing step. The pre-processing step performs partitioning by destination and balancing workload based on edges. Transmuter graph processing involves initialisation of data structures, assigning of partitions to the GPEs, and then processing by the GPEs.

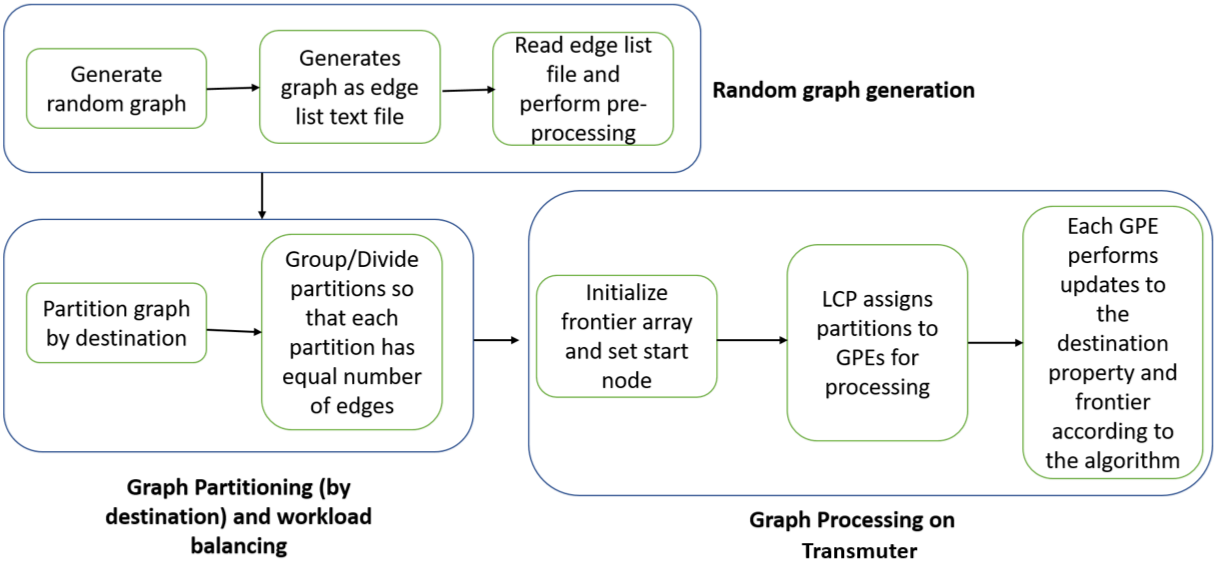

Generate random graph
Generates graph as edge list text file
Read edge list file and perform pre-processing
Random graph generation
Partition graph by destination
Group/Divide partitions so that each partition has equal number of edges
Graph Partitioning (by destination) and workload balancing
Initialize frontier array and set start node
LCP assigns partitions to GPEs for processing
Each GPE performs updates to the destination property and frontier according to the algorithm
Graph Processing on Transmuter

Figure 3.1: Graph Processing Workflow

3.2.1 Random Graph Generation

The first step of graph processing is to generate a random graph. Python Random Graph Generator [5] was used to generate a graph. It generates graphs with different parameters such as number of nodes, minimum number of edges per node, maximum number of edges per node, minimum edge weight value, maximum edge weight value and the file format of the output edge list.

3.2.2 Graph Pre-Processing

The pre-processing steps involve taking the edge list file generated by the random graph generator tool to form the graph data structure, followed by static partitioning by destination and also workload balancing.

Forming graph data structure from edge list file: The edge list generated by the random graph generator tool is used as an input to the graph processing step. The tool generates a text file in Comma-Separated Values of source node, destination node and edge weight. Every line in the edge list file corresponds to an edge in the graph. Python Pandas package is used to read the input graph file and convert to Scipy-based CSC format.

Partitioning by destination and edge load balancing: Static partitioning by destination scheme has been implemented. Note that CSC format inherently supports partition by destination. To achieve edge load balancing, the in-degree for every vertex is computed and number of edges each GPE should process is computed. The assignment is done such that the number of edges in each partition is close to each other in the destination-based partition. By assigning equal workload to each GPE, the stall time to hit the global barrier after every iteration is minimized.

3.3 Data Structure

Compressed Sparse Column(CSC) format involves use of 3 data structures :

1. **Row pointer array**: This is an unsigned integer array with the size equal to one more than number of nodes in the graph. Every index in this array corresponds to the destination node value. Each element in this array holds the value of starting index of in-edges with respect to the destination node. Accesses to this array are random.

2. **Source indices array**: This is also an unsigned integer array which holds the source nodes correspond to incoming edges of the destination nodes. All source vertices for a particular destination node are stored consecutively and are accessed sequentially.

3. **Edge weight value array**: This is an array of type float. Every value corre-

sponds to the edge weight value.

Apart from the above data structures, the following arrays are also used:

4. **Vertex list**: This float array holds the property value for every node. Based on the algorithm being processed, this could be the distance value of a node in the case of BFS algorithm, or page rank of a node for Page Rank algorithm.

5. **BFS Predecessor array**: Breadth-First Search algorithm requires the BFS path of a node from a given start node to be stored. The BFS predecessor array also helps in checking if a particular node has already been visited or not.

6. **Frontier**: Frontier array can be implemented in 2 ways - as a Boolean array and a variable sized array.

    - Boolean array: In the Boolean array mode, every node is assigned a '1' or '0' based on whether it is active or not. For BFS algorithm, every round of processing requires an update of the frontier array. $frontier(i) = 1$, if node i is active

    - Dynamic-sized workqueue: An array is used to store the indices of the active nodes. In most graph processing cases, the number of active nodes is only about 5-10% of the total number of nodes in the graph. So the storage overhead of this data structure is significantly lower.

3.4 GPE Processing

In frontier-based kernels, in every computation round, the frontier identifies the active nodes in the current iteration. Neighbours of the active nodes are visited by traversing the edges and an algorithm-specific update function is computed. At the end of every round of graph processing, a new frontier is calculated. The nodes whose properties were updated in the current iteration become active nodes in the next iteration. These series of steps for every iteration are performed until the frontier is empty or a termination criterion is met. In a bulk synchronous parallel model of implementation, there is a synchronization step where processing of all GPEs is stalled until all the computing nodes reach the end of one round of iteration. This is done to make sure operation of all cores are synchronized and updates done by each computing node during an iteration is available for the next iteration. BFS and SSSP are frontier-based algorithms

In our implementation, the LCP is responsible for scheduling work to the GPEs. It assigns partitioned graphs to each GPE. The LCP pseudo-code is described in Algorithm 5.

The GPEs work on the partition assigned to it. For frontier-based kernels like BFS and SSSP, the current frontier is checked to trace the destination nodes, their properties are updated and the frontier for the next iteration is computed. A generic algorithm for the implementation of any frontier-based algorithm on Transmuter has been described in Algorithm 6.

Iterative kernels like Page Rank do not work based on a frontier. All nodes are active in all iterations and the node property values of all nodes are computed during every round of processing. The pseudo-code for iterative kernels is explained in Algorithm 7.

The destination property update varies for different kernels.

- **BFS**: Since BFS is a traversal algorithm, in order to be able to trace the path from a node upto the source node, the predecessor node for every vertex is recorded as the property of a node.

$$V_{bfs\_predecessor} = V_{source} \tag{3.1}$$

Algorithm 5 LCP Pseudo-Code

Input: Entire graph G = (V,E)

Output: Assigning work to GPEs

```
Read the CSR/CSC data structures of the partitioned graph from
Python-generated pre-processing file.
Assign the partitions to GPEs
```

- **Page Rank**: In this simplified version of Page Rank, the rank of every destination node is updated in every iteration by utilizing the rank value of its source node and the out-degree of the source node [6].

$$V_{rank} = V_{rank} + \sum_{U|(U,V)\in E} \frac{U_{rank}}{U_{deg}} \tag{3.2}$$

- **Single Source Shortest Path**: The shortest distance to reach the node from the source node is updated as part of destination property update step[6].

$$\min_{U|(U,V)\in E}(V_{dist}, U_{dist} + E_{weight}(U, V)$$

Algorithm 6 Kernel Pseudo-Code For Frontier-Based kernels

```
Input: Partitioned graph and CSR/CSC data structures
  For each active vertex v
          If number of nodes visited not reached user-specified value
              For each vertex v in partition, iterating indices array
                  If corresponding source active and node not visited
                      Read source, edge value or node degree, if necessary
                      Update destination property
                      Make destination node active for next iteration
                      Make current active node inactive for next iteration
                      Increment number of nodes visited by 1
```

Algorithm 7 Kernel Pseudo-Code For Iterative kernels

```
Input: Partitioned graph and CSR/CSC data structures
  For each active vertex v
          If number of iterations not reached user-specified value
              For each vertex v in partition, iterating indices array
                  Read source, edge value or node degree, if necessary
                  Update destination property
              Increment number of iterations performed by 1
```

Chapter 4

RESULTS

We present the Transmuter implementation results on graphs of varying sizes and sparsities. Sparsity of a graph can be viewed as the average degree of every node in the graph. Graph size can be varied by altering the number of vertices and edges present in the graph. The metrics to evaluate performance on Transmuter are execution time, Giga Operations Per Second(GOPS), power consumption(W), GOPS/W and the L1 and L2 cache hit rates.

4.1 Calculation of performance metrics

The statistics file generated by Gem5 are used for obtaining the GPE-wise L1 cache hit rates, tile-wise L2 cache hit rates, the number of operations, total execution time etc. The power consumption values are obtained for 14nm using a script designed by Siying Feng from the University of Michigan.

A summary of the procedure to calculate power is as follows: Static power is computed by summing up the static power of the individual Transmuter components. Dynamic power is calculated by dividing the total dynamic energy of the system by the total execution time.

For ARM cores, the data used are the static and dynamic power of its online specification (40 nm) scaled to 14 nm. The dynamic energy is calculated by multiplying the given dynamic power with the number of active cycles.

For re-configurable caches, the static power and energy per transaction are generated using CACTI model for 14 nm node. The dynamic energy is computed by multiplying energy per transaction with the total number of accesses obtained from the input statistics file.

For re-configurable crossbars, the dynamic energy is calculated in the same way as the re-configurable caches. All muxes and memory controllers are modeled as re-configurable crossbars and the scratchpads are modeled as re-configurable caches.

The performance is measured in the following way:

$$GOPS = committed\_ops/(sim\_secs * 10^9); \tag{4.1}$$

committed_ops and *sim_secs* are obtained from GEMV stats file. *committed_ops* correspond to number of operations executed and *sim_secs* refer to the execution time for processing.

4.2 Partitioning by destination results

To study the importance of partitioning a graph, BFS was implemented with and without partitioning on Transmuter. In this experiment, the graphs were represented in the COO data format. Transmuter configuration was 2 tiles with 4 GPEs per tile, 4 kB per L1 cache bank, 4 kB per L2 cache bank and 8 kB of synchronization scratchpad.

Figures 4.1 and 4.2 present results with and without partitioning for graphs with 100 nodes and 100 edges, and 100 nodes and 200 edges, respectively. We see that processing of un-partitioned graphs takes about 3x more than a graph that has been partitioned by destination. This happens because update of destination property in the case of un-partitioned graph leads to large number of data conflicts since the same destination node property could be updated by different processing cores. To avoid this, synchronization mechanisms such as mutexes have to be used which cause a significant overhead and consume about 20% of the total execution time per iteration. Thus, further graph processing in this thesis has been performed on graphs that are

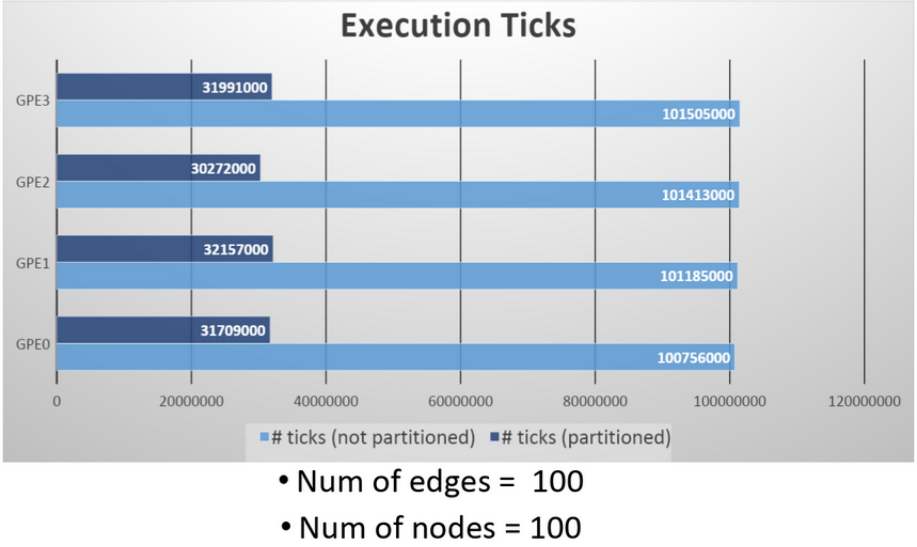


Figure 4.1: Comparison Of Execution Times For BFS On Graphs With And Without Partitioning by Destination. $|V| = 100$, $|E| = 100$

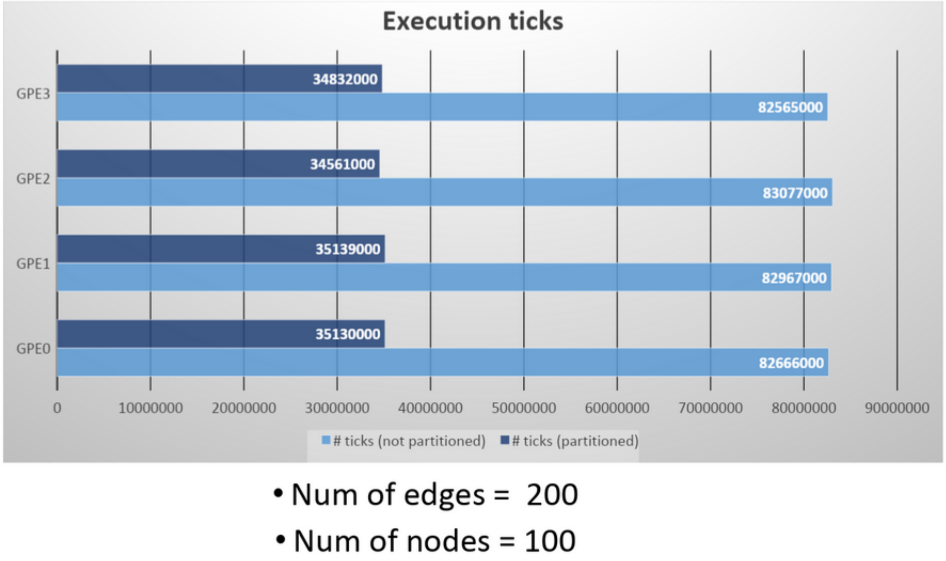


Figure 4.2: Comparison Of Execution Times For BFS On Graphs With And Without Partitioning by Destination $|V| = 100$, $|E| = 200$

statically partitioned by destination.

4.3 Breadth-First Search Algorithm

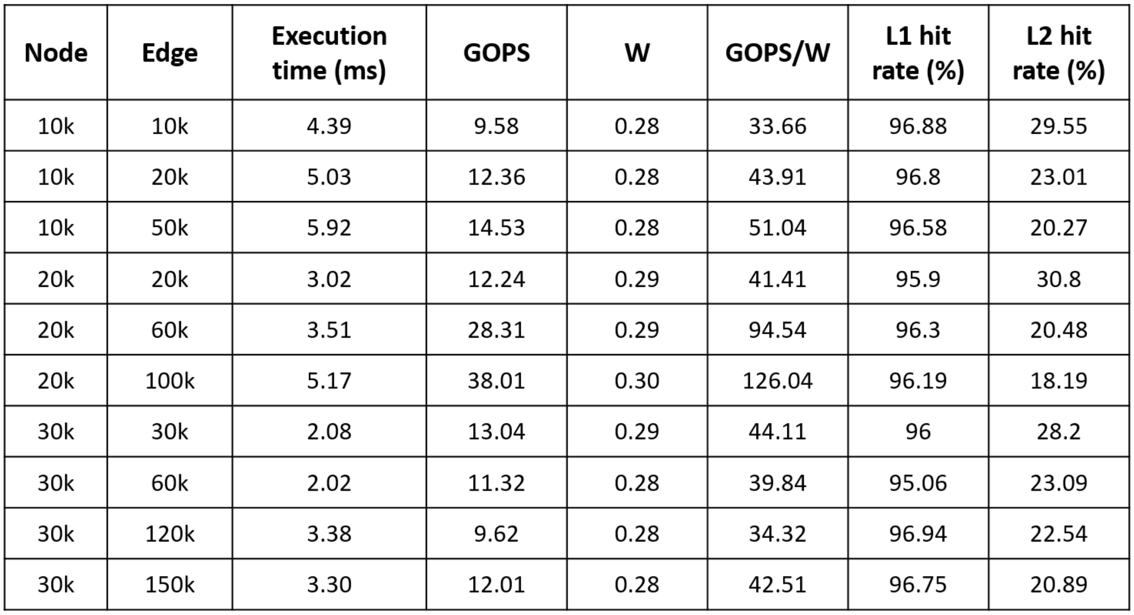
Breadth-First Search (BFS) is a frontier-based algorithm - the number of nodes active at any point in time depends on the frontier state at that time. Graph processing happens only on the frontier nodes. BFS is performed until the number of nodes visited reaches a user-specified value.

BFS has been implemented using a Boolean based frontier array. The storage space required for Boolean frontier array is equal to the number of nodes in the graph. Every element corresponds to the node being inactive or active in the current iteration.

Figure 4.3 shows the execution time, GOPS, GOPS/W and L1 and L2 cache access data for BFS implementation on different graph sizes. It can be observed that as the graphs get more and more dense, the GOPS numbers increase. For instance $|V| = 10k$, then as $|E|$ increases from 10k to 50k, the GOPS increases from 33.66 to 51.04. As the graphs become more dense, every node has a lot more edges to process in every iteration and so the GOPS numbers increase.

From the L1 and L2 cache data, it can be observed that the L1 cache performance is superior an average hit rate of 95 to 97% as shown in Fig. 4.3. This indicates that most of the data required by all the GPEs in every tile are fetched and reused as much as possible causing minimum number of L1 misses. However, the L2 hit rates are quite low.

To improve the cache performance, a prefetcher can be utilized which will retrieve the data from main memory into the caches much before the data is required. Prefetching is a common solution to boost L1 and L2 hit rates. We project that the use of a prefetcher is likely to increase the number of sequential data accesses to the



Node	Edge	Execution time (ms)	GOPS	W	GOPS/W	L1 hit rate (%)	L2 hit rate (%)
10k	10k	4.39	9.58	0.28	33.66	96.88	29.55
10k	20k	5.03	12.36	0.28	43.91	96.8	23.01
10k	50k	5.92	14.53	0.28	51.04	96.58	20.27
20k	20k	3.02	12.24	0.29	41.41	95.9	30.8
20k	60k	3.51	28.31	0.29	94.54	96.3	20.48
20k	100k	5.17	38.01	0.30	126.04	96.19	18.19
30k	30k	2.08	13.04	0.29	44.11	96	28.2
30k	60k	2.02	11.32	0.28	39.84	95.06	23.09
30k	120k	3.38	9.62	0.28	34.32	96.94	22.54
30k	150k	3.30	12.01	0.28	42.51	96.75	20.89

Figure 4.3: Breadth-First Search Algorithm Execution Results For Frontier-Based Implementation

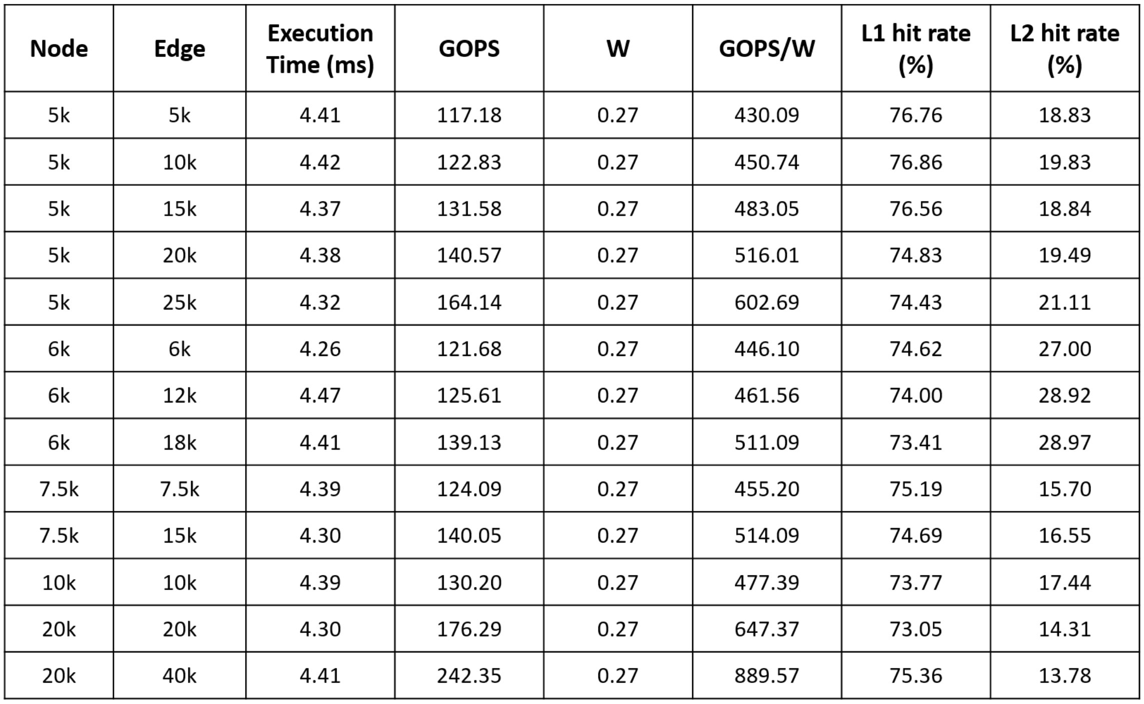
sources/destination indices array, visited array and the work queue array.

4.4 Page Rank Algorithm

Page Rank (PR) is an iterative algorithm where all nodes are active in all iterations and the rank for every node is calculated. This is not a frontier-based algorithm, like BFS. The termination condition used for this algorithm is when the number of iterations processed reaches a user-defined value.

Figure 4.4 shows the execution time, GOPS, GOPS/W results for implementing PR on different graph sizes. Similar to BFS results, it can be observed that as graphs get more and more dense, the GOPS/W numbers increase. The reason being, as the graphs become more dense, every node has a lot more edges to work on and so the GOPS increase.

It can also be seen that GOPS and the GOPS/W numbers for PR are much higher



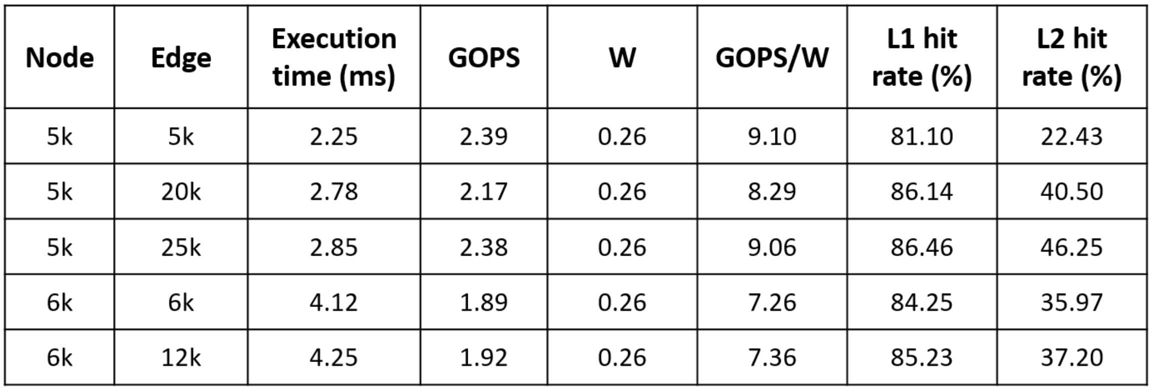
Node	Edge	Execution Time (ms)	GOPS	W	GOPS/W	L1 hit rate (%)	L2 hit rate (%)
5k	5k	4.41	117.18	0.27	430.09	76.76	18.83
5k	10k	4.42	122.83	0.27	450.74	76.86	19.83
5k	15k	4.37	131.58	0.27	483.05	76.56	18.84
5k	20k	4.38	140.57	0.27	516.01	74.83	19.49
5k	25k	4.32	164.14	0.27	602.69	74.43	21.11
6k	6k	4.26	121.68	0.27	446.10	74.62	27.00
6k	12k	4.47	125.61	0.27	461.56	74.00	28.92
6k	18k	4.41	139.13	0.27	511.09	73.41	28.97
7.5k	7.5k	4.39	124.09	0.27	455.20	75.19	15.70
7.5k	15k	4.30	140.05	0.27	514.09	74.69	16.55
10k	10k	4.39	130.20	0.27	477.39	73.77	17.44
20k	20k	4.30	176.29	0.27	647.37	73.05	14.31
20k	40k	4.41	242.35	0.27	889.57	75.36	13.78

Figure 4.4: Page Rank Algorithm Execution Results

than that for the BFS algorithm. This is because BFS is a frontier-based algorithm - at any point in time only a fraction of nodes are active and graph computations are performed only on these active/frontier nodes. In contrast, for PR, all nodes are processed every time and their properties updated. Also, PR computes the rank for every node which involves floating point values.

4.5 Single-Source Shortest Path algorithm

The Single-Source Shortest Path (SSSP) algorithm involves computing the distance of all nodes in the graph from a given source node. A frontier Boolean array is used to keep track of the node with minimum distance value in every iteration. The neighbours of the unvisited node with minimum distance value are traversed and the distance property values are updated for these neighbouring nodes.



Node	Edge	Execution time (ms)	GOPS	W	GOPS/W	L1 hit rate (%)	L2 hit rate (%)
5k	5k	2.25	2.39	0.26	9.10	81.10	22.43
5k	20k	2.78	2.17	0.26	8.29	86.14	40.50
5k	25k	2.85	2.38	0.26	9.06	86.46	46.25
6k	6k	4.12	1.89	0.26	7.26	84.25	35.97
6k	12k	4.25	1.92	0.26	7.36	85.23	37.20

Figure 4.5: Single-Source Shortest Path Execution Results

The GOPS and GOPS/W numbers are presented in Figure 4.5. These numbers are a lot smaller than BFS and PR. This is because in this algorithm, only the node which has minimum distance value in the entire graph is active during any iteration. This causes the utilization of the processing cores to drop compared to BFS and PR kernels. A much more efficient implementation would be to have each tile process different shortest paths with different start nodes.

From the results, it can be observed that the L1 hit rate is between 81 to 87%, which is lower than BFS and PR. As the density of graphs increases, the L1 and L2 hit rates tend to increase. This shows that as graphs become more dense, the number of vertices that a node is connected to, is higher. Spatial and temporal locality of sequential access of neighbouring nodes in CSC format might be the reason for the increased L2 hit rate.

Chapter 5

CONCLUSIONS AND FUTURE WORK

5.1 Conclusions

The work focuses on an efficient implementation of three popular graph kernels, namely Breadth-First Search (BFS), Page Rank (PR) and Single Source Shortest Path (SSSP) on a multi-core architecture, Transmuter, that was developed at the University of Michigan. The challenges in graph computations such as imbalance in workload, presence of random accesses and memory-intensive operations, poor memory locality and large number of data conflicts have been addressed by partitioning by destination, balancing workload, utilizing the CSC data format and having multiple levels of memory hierarchy.

All three implementations use a static partitioning by destination scheme coupled with equal distribution of the workload in the pre-processing step. Splitting the graph into disjoint destination-based partitions and having the processing cores perform computation on these smaller sub-graphs improve both cache performance and reduce the data conflicts. For instance, BFS on a graph that has not been partitioned by destination takes 3x more execution time than a partitioned graph. Balancing workload by assigning similar number of edges to the cores ensures effective utilization of the cores, thus improving execution time.

We evaluated the performance of the kernels using multiple metrics including GOPS/W. We found that kernels with more number of computations per iteration have higher GOPS/W number. Thus, Page Rank algorithm which involves computing rank values for all nodes in the graph at all times, exhibits a higher GOPS/W. For instance, for graphs of sizes ranging from 5000 to 20000 nodes, it achieves 400 to 900 GOPS/W. In contrast, BFS and SSSP have significantly lower GOPS/W. SSSP has the lowest GOPS/W, due to the fact that only one node with the minimum distance from the source is active in any iteration.

Transmuter supports a multi-level memory hierarchy which helps in reducing the frequent high latency accesses to the main memory and thus improving execution time. Our analysis shows that BFS exhibits high L1 cache hit rates of about 95% for a Boolean array-based frontier implementation. The L2 hit rates are however very low ($\approx 20\%$) and can be possibly improved through efficient prefetching. PR and SSSP also have fairly high L1 hit rates ($75\% - 85\%$) but low L2 hit rates.

5.2 Future Work

The work presented in this thesis is an initial exploration of graph processing implementations on a parallel multi-core architecture.

- The frontier-based version suffers from high storage since the frontier array is as large as the number of nodes in the graph. For large graphs, this can be a significant concern. The work queue based implementation makes use of an integer array which stores the IDs of the nodes that are active. Since the number of active nodes is quite small, the storage requirement of such a scheme is significantly reduced. A work queue based method can be employed for Breadth-First Search. The processing performance when a prefetcher takes advantage of the sequential accesses inherent in CSC data structure can also be studied.

- While a static partitioning scheme was used in this thesis, a dynamic partitioning scheme can be employed which splits the graph consisting of active nodes into partitions at the end of every iteration. This makes sure that for a frontier-based algorithm such as BFS and SSSP, all the GPEs that perform graph processing do not have inactive nodes in its partition resulting in workload imbalance.

- The performance of graph processing kernels for CSR, CSC and COO layouts differ based on the size of frontier and sparsity of graph. Switching between these layouts dynamically has the potential to improve performance and efficiency. Such a study has to be conducted.

- The performance of the graph processing kernels is also a function of the partitioning kernels, sparsity of the frontier array, etc. Based on size of the frontier, switching dynamically between data structures and partition mechanism can have a profound impact on the performance. This needs to be investigated as well.

REFERENCES

[1] Djikstra's Algorithm. `https:https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm/`.

[2] Simplified Page Rank Algorithm. `https://www.geeksforgeeks.org/page-rank-algorithm-implementation/`.

[3] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. *ACM SIGARCH Computer Architecture News*, 43(3):105–117, 2016.

[4] Guohao Dai, Tianhao Huang, Yuze Chi, Jishen Zhao, Guangyu Sun, Yongpan Liu, Yu Wang, Yuan Xie, and Huazhong Yang. Graphh: A processing-in-memory architecture for large-scale graph processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[5] Sepand Haghighi. Pyrgg: Python random graph generator. *The Journal of Open Source Software*, 2(17), sep 2017.

[6] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[7] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-marl: a dsl for easy and efficient graph analysis. *ACM SIGARCH Computer Architecture News*, 40(1):349–362, 2012.

[8] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a {PC}. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 31–46, 2012.

[9] Gushu Li, Guohao Dai, Shuangchen Li, Yu Wang, and Yuan Xie. Graphia: an in-situ accelerator for large-scale graph processing. In *Proceedings of the International Symposium on Memory Systems*, pages 79–84. ACM, 2018.

[10] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[11] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, 2014.

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

[13] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[14] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sanchez. Exploiting locality in graph analytics through hardware-accelerated traversal scheduling. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–14. IEEE, 2018.

[15] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. Outerspace: An outer product based sparse matrix multiplication accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 724–736. IEEE, 2018.

[16] Julian Shun, Laxman Dhulipala, and Guy E Blelloch. Smaller and faster: Parallel processing of compressed graphs with ligra+. In *2015 Data Compression Conference*, pages 403–412. IEEE, 2015.

[17] Jiawen Sun. *The GraphGrind Framework: Fast Graph Analytics on Large Shared-memory Systems.* PhD thesis, Queen's University Belfast, 2018.

[18] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R Dulloor, Michael J Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. Graphmat: High performance graph analytics made productive. *Proceedings of the VLDB Endowment*, 8(11):1214–1225, 2015.

[19] Wikipedia. Breadth First Search. `http://en.wikipedia.org/wiki/Breadth-first_search/`.