ABSTRACT GRAPH MACHINE: MODELING ORDERINGS IN ASYNCHRONOUS DISTRIBUTED-MEMORY PARALLEL GRAPH ALGORITHMS

Thejaka Amila Kanewala

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the School of Informatics, Computing, and Engineering
Indiana University
September 2018

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Doctoral Committee

Andrew Lumsdaine, Ph.D.

Esfandiar Haghverdi, Ph.D.

Haixu Tang, Ph.D.

Predrag Radivojac, Ph.D.

Alex Pothen, Ph.D.

July 30, 2018

Dedicated to my wife Eroma, daughter Methindri and to my parents. For believing in me and being strong pillars of my life.

Acknowledgements

My parents motivated and encouraged me to always to reach the highest, I possibly could in life, in studies, which provided the initial seed for pursuing my Ph.D. My wife Eroma and my daughter Methindri was with me through all the ups and downs of my Ph.D. journey, without whom I would not be able to complete it. My father is the most courageous person I have seen, his courage shaped me to be who I am today and what I have achieved. I am ever grateful and fortunate to have them in my life.

Prof. Andrew Lumsdaine was the pillar who continuously guided and advised me throughout the Ph.D. carrier. While Andrew's advises shaped the research, they also helped me to improve myself as a scientist, to think and act like one. I took his guidance and direction to my heart as valuable lessons. He was there to lift me up and encourage when I was discouraged and felt low. Most of all he believed in me and he believed in the work I was doing and continued to support and improve that work by giving continuous feedback. Equally I had a great committee, and my sincere gratitude to: Andrew Lumsdaine, Esfandiar Haghverdi, Haixu Tang, Predrag Radivojac, and Alex Pothen. Their time and feedback helped me to improve this thesis.

I had many productive discussions with Marcin Zalewski (Research Scientist at PNNL) which always resulted in improving the work presented in this thesis. I was fortunate to work at Pacific Northwest National Laboratory (PNNL) for a year and during this period I got the chance to work with some of the most amazing computer scientists related to my research field. I am very thankful to John Feo and PNNL HIVE project team members. Kelsey Shepard went above and beyond to help me with some of the administrative tasks which are pretty hard for me to take care of. Kelsey's excellent organization skills helped us to win several grants that supported our research.

As a secondary English speaker, I had a hard time writing papers and proposals at the beginning. Laura Hopkins helped me to edit and review my paper drafts. While correcting my English she also taught me how to write a research paper like a story, how to write coherent sentences, when to break paragraphs, etc. Her teachings and feedback immensely helped me to improve my writing ability. She was very kind to me and one of the people who really wanted to see I am completing my Ph.D. Sadly she passed away before I completed my Ph.D. and left me in a shock.

Doing a Ph.D. with a child was not an easy task and myself and Eroma went through many hardships and many challenges. In all those hard times friends around us helped us to overcome stressed periods. The small Sri Lankan community in Bloomington was like a family to us. Lahiru, Saminda, and Heshan were among the few friends who were very close to us from the beginning. Both Chathuri and Supun became like our siblings. Apart from us, Methindri was most close to them. Thilina, Bimalee, Samitha, Milinda, Isuru, Pavithri, Udayanga, Charitha, Saliya, Kalani, Buddhika are few among many friends who helped us.

Professor Sanath Jayasena was my M.Sc. thesis advisor and he encouraged me to perceive a Ph.D. Not knowing much about what I am signing up for, he helped me a lot to understand what a Ph.D. is and continuously encouraged me to go for higher studies. As a person and also as a mentor I have an utmost respect for Dr. Sanjiva Weerawarana. His recommendation, enabled getting selected by Indiana University.

Chathura Herath (CK) is one of the brilliant minds and a loyal friend I have met in my life. His recommendation helped to gain the trust of Suresh Marru and Marlon Pierce at Science Gateways Research Center (SGRC) in Indiana University. SGRC is a unique place and one of the best times in my academic life. Suresh gave me many opportunities with the freedom of selecting which I want to proceed with. I selected the opportunities where I can make a significant impact and early exposure to research, papers and proposals I got through at SGRC later helped me to excel in my research. Suresh was not only my mentor but also a dear family friend.

I love mathematics. However, my mathematical skills were unpolished and without practice when I enrolled at IU. I wanted to take Logic and Mathematics subjects but in doubt whether I will be able to successfully complete those courses. It was Jayampathy Ratnayake (J) and Prabath De Silva who helped me resurrect my mathematics skills. I struggled a lot to understand some of the Logic subjects such as Co-Algebra, Modal Logic, etc. Jayampathy and Prabath spent their valuable time teaching me and helping me to understand those complex subject matter. Jayampathy introduced me to some of the great professors in the Mathematics department at IU (Prof. Larry Moss) and encouraged me to take their subjects. I had many discussions with Prabath about analytics of AGM and greatly helped me to understand characteristics of certain algorithms and helped me to improve the AGM model.

Vanderburgh, Dana Scott was invaluable in editing and reviewing drafts of my thesis. Chatura Wickramaratne, Hasini Gunasinghe, Nihal Liyanage, Sasith Rajasooriya are few other friends who helped me to complete this journey in various ways.

Thejaka Amila Kanewala

ABSTRACT GRAPH MACHINE: MODELING ORDERINGS IN ASYNCHRONOUS DISTRIBUTED-MEMORY PARALLEL GRAPH ALGORITHMS

Graphs are ubiquitous data structures. Processing large graphs require distributed-memory parallel graph algorithms. Many existing distributed-memory parallel graph algorithms are extended from parallel graph algorithms developed for shared-memory platforms. These extensions show poor performance in distributed execution, due to factors such as frequent synchronizations, increased irregular memory access overhead influenced by operations like subgraph computations. One way to mitigate such overheads is to adopt purely asynchronous algorithms, yet they show poor performance due to a high amount of work generated.

This thesis presents Abstract Graph Machine (AGM), a model that can control the amount of synchronization needed for a distributed-memory parallel graph algorithm. AGM represents an algorithm with a function that encapsulates logic and a strict weak ordering relation. The strict weak ordering relation separates work into equivalence classes; work units that are not comparable are inserted into the same equivalence class and can be executed in parallel. Work units that are comparable are separated into different equivalence classes and processing equivalence classes are sequenced according to the induced ordering. The model is further extended (Extended-AGM) to specify orderings at lower spatial levels in a memory hierarchy. The thesis shows that with the AGM and Extended-AGM (EAGM) models, one can derive families of graph algorithms by specifying different orderings while keeping the same processing logic.

Both AGM and EAGM models are implemented on top of a Message Passing Interface (MPI) based runtime. The thesis discusses challenges faced when mapping AGM and EAGM models to an implementation and how we overcame them. We compare the performance of AGM framework graph applications with the performance of popular graph processing frameworks such as Parallel Boost Graph Library, GraphLab-PowerGraph, and

CombBLAS. Further, thesis analyzes the performance of different orderings on power-law graphs, uniform degree distribution graphs, low diameter graphs and high diameter graphs.

Andrew Lumsdaine, Ph.D.

Esfandiar Haghverdi, Ph.D.

Haixu Tang, Ph.D.

Predrag Radivojac, Ph.D.

Alex Pothen, Ph.D.

Contents

List of Figures

List of Acronyms

EAGM: Extended Abstract Graph Machine 242

CC: Connected Components 241

MST: Minimum Spanning Tree 14

St: statement 34

BFS: Breadth First Search 241

KLA: K-Level Asynchronous 240

DCSCC: Divide & Conquer Strongly Connected Components 25

DFS: Depth First Search 18

PGAS: Partitioned Global Address Space 19

DAG: Directed Acyclic Graph 235

GAS: Gather-Apply-Scatter 227

NUMA: Non-Uniform Memory Access 150

API: Application Programming Interface 149

STL: Standard Template Library(C++) 181

BST: Binary Search Tree 242

CSR: compressed sparse row 86

PSP: *predecessor, successor's predecessor* 23

SPS: *successor, predecessor's successor* 124

SSS: *successor, successor's successor* xvi

1

Introduction

Graphs are the main data structures that best represent relations in data. Graph algorithms are used to extract important information about the depicted data. For many graph applications, there is more than one sequential graph algorithm. However, in most cases, sequential graph algorithms are not directly parallelizable for efficient execution. Developing an efficient parallel graph algorithm is challenging. Designing an efficient distributed-memory parallel graph algorithm is even harder due to the involvement of the network and the complications that arise from the data distribution.

Most of the existing parallel graph algorithms are developed focusing shared-memory systems and these shared-memory parallel graph algorithms do not immediately extend as efficient distributed-memory parallel graph algorithms. In shared-memory, the algorithm has access to the whole graph data structure and communication cost between processors is low compared to distributed-memory. Therefore, shared-memory parallel graph algorithms use techniques such as barriers, synchronous communication and sub-graph computations. While these techniques have less effect on the performance of shared-memory parallel graph algorithms, they add a significant overhead on extended distributed-memory parallel graph algorithms. Asynchronous graph algorithms is an alternative approach that does not have the overheads discussed above. However, asynchronous algorithms tend to generate more work and messages compared to synchronous graph algorithms and may result in poor performance due to low compute/communication ratios and may consume more power. In this thesis, we use ordering to control the execution of asynchronous graph

algorithms and hence to control the amount of work and messages generated. We present, *Abstract Graph Machine (AGM)*: an abstract model to represent asynchronous distributed-memory parallel graph algorithms. Thus, we show that families of graph algorithms can be derived by changing ordering while keeping the processing logic intact. Further, we extend our model to explore orderings at different spatial memory levels (the extended machine is called *Extended Abstract Graph Machine (EAGM)*) and show that more efficient algorithms can be obtained by avoiding global synchronization and by applying ordering at lower spatial levels.

The thesis presents *families* of distributed-memory parallel graph algorithms derived using the AGM model for Single-Source Shortest Paths (SSSP) (Chapter 5) and shows that they outperform some of the widely used distributed-memory parallel SSSP algorithms. We developed, implemented and evaluated new asynchronous distributed-memory graph algorithms for Connected Components (Chapter 6), Maximal Independent Set (Chapter 8 and Chapter 7) and Triangle Counting (Chapter 9). These new algorithms are modeled using the AGM and EAGM to derive variations. The scalability of these algorithms and their variations are evaluated with a number of different synthetic graph inputs and real-world graph inputs.

In Chapter 11 and Chapter 12 we present an efficient implementation of the AGM model and EAGM model. The AGM framework mainly takes a function that encapsulates algorithm logic and an ordering (A *strict weak ordering* relation. Both inputs are specified as C++ functors) and executes the algorithm by ordering generated work according to the input relation. The EAGM framework takes a list of orderings specified for each spatial level and executes the algorithm by ordering generated work according to the relations specified for the spatial level. We experimentally evaluate the performance of the AGM framework for a number of different orderings and for a number of different graph applications. We show that the performance of framework algorithms is competitive with the hand-written graph algorithms and also outperforms some of the well-known graph processing frameworks (e.g., PowerGraph-GraphLab **[90]**, CombBLAS **[20]**).

When it comes to performance, distributed-memory parallel graph algorithms are inseparable from the underlying runtime. These runtime features are discussed in detail in Chapter 10. Further, the performance of graph algorithms are data dependent and heavily depend on the characteristics of the input graph. This framework allows us to execute an algorithm with a given spatial and temporal ordering and hence we can derive algorithms with different synchronization levels. For example, we can execute an algorithm as a globally synchronous algorithm, or a globally asynchronous, but locally (process level) synchronous algorithm etc.. We show that the best ordering choice to execute an algorithm depends on the input and that, globally asynchronous execution generally shows better performance for high diameter graphs and power-law graphs while globally synchronous execution shows better performance for graphs with uniform degree distributions (e.g., Erdos-Renyi graphs [**42**]).

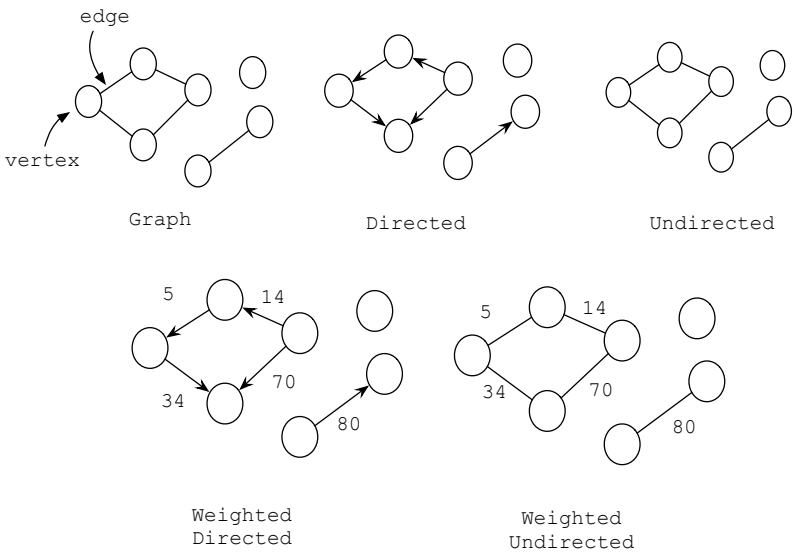


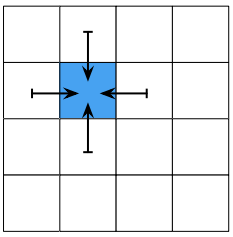
FIGURE 1.1. Graph primitives.

1.1.1. Graph Terminology. A *Graph* is an ordered pair, $G = (V, E)$ where V is a set of vertices and E is a set of edges. An edge is a pair of vertices. When an edge has a direction we call that graph a *directed* graph and when the direction of an edge is not specified, the

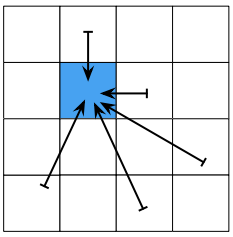
graph becomes an *undirected* graph. By mixing some of the features we can derive different types of graphs (See Figure 1.1). The degree of a vertex is the number adjacencies a vertex has.

A *scale-free* graph is a graph whose degree distribution follows a *power law*. Most real-world graphs are scale-free and those graphs have a common number of vertices with a high degree. The *diameter* of a graph is the highest shortest path between any pair of vertices.

A graph algorithm is *ordered* if the execution order affects the correctness of the algorithm. If the execution order does not affect the correctness of the algorithm, then it is an *un-ordered* algorithm. An asynchronous graph algorithm is *label setting* if the algorithm writes to output state only once. If the algorithm writes more than once to the output state before it converges, then the algorithm is *label correcting*.



(A) Communication pattern in a regular application.



(B) Communication pattern in a graph application.

FIGURE 1.2. Comparison of communication patterns in a regular application and a graph application.

1.1.2. Irregularity. Unlike in regular applications, graph algorithms' memory access patterns are irregular. Regular applications distribute data in such a way that communication is necessary only to process data within boundaries. Data that do not lie on boundaries are processed locally. For this reason, regular applications maintain a high compute communication ratio. *Stencil* computation is an example application that processes data this way. Figure 1.2a shows how data in one process depends on data in other processes in a regular application (2D Stencil computation, more specifically). On the other hand, Figure 1.2b shows how the graph data in one process depends on graph data owned by other

processes. As shown in the figure, graph algorithm data in a single process can depend on other distributed data. Figure 1.2b assumes the graph is distributed by distributing its vertices equally among processes. Further, data dependency in an algorithm is usually defined by how vertices connect to each other. Therefore, it is almost impossible to come up with a strategy to distribute graph data in such a way that communication is similar to a regular application. Due to irregular memory access patterns, the compute/communication ratio in graph applications is lower than regular applications.

The effect of irregularity on the performance of distributed memory parallel graph algorithms is higher than its effect on sequential graph algorithms. Irregular memory access patterns cause performance overhead even in sequential algorithms in-terms of cache misses, compared to sequential regular applications. Shared-memory is the first step of distributing processing among several processors. In shared-memory, we have multiple processors running independently and communicating via a single shared-memory. As we discussed previously, regular application's data can be laid over shared-memory, in such a way, each processor processes only a set of data that is local to the processor and does not need to share data with another processor. In irregular applications, a processor may access any data element in shared-memory.

The second step of distributing processing is the distributed-memory. In distributed-memory, each processor holds a local memory and processors communicate through a network. When going from shared-memory to distributed-memory, a constant factor is added to the communication cost. However, irregular applications' communication is more frequent than that of regular applications. For irregular applications that constant factor adds a significant overhead and is no longer negligible.

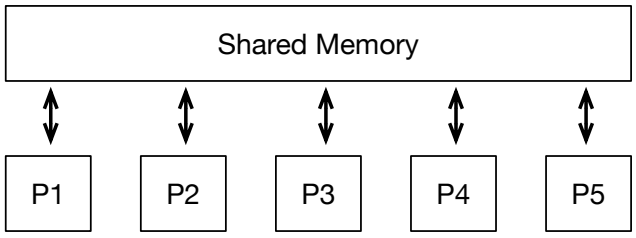


FIGURE 1.3. Shared-Memory Model

Algorithm 1 Shared-Memory Parallel Graph Algorithm

```
1: procedure ALGORITHM(G = (V, E))
2:   while some condition do
3:     for each Vertex v in V in parallel do
4:       ...
5:     end for
6:   end while
7: end procedure
```

1.1.3. Shared-Memory Execution. The majority of the existing parallel graph algorithms were developed for shared-memory parallel platforms and analyzed using the Parallel Random Access Memory (PRAM) [**45**] model. The PRAM model consists of a set of individual *Random Access Memory* (RAM) processors, a memory local to each independent processor (e.g., registers), and, a shared-memory (Figure 1.3). Different processors write to and read from the shared-memory. The model assumes that reads and writes to the shared-memory are synchronized. Each read or write is considered as a single time-step. Then, the cost of an algorithm is expressed in terms of the number of times the algorithm reads and writes to the shared-memory. Further, most of the algorithms designed for shared-memory are iterative, which means that those algorithms iterate over vertices and/or edges until they reach a particular condition (See Algorithm 1). Thus, the algorithm complexity boils down to the total number of iterations (depth), and the number of processors needed to complete a single iteration in parallel (breadth).

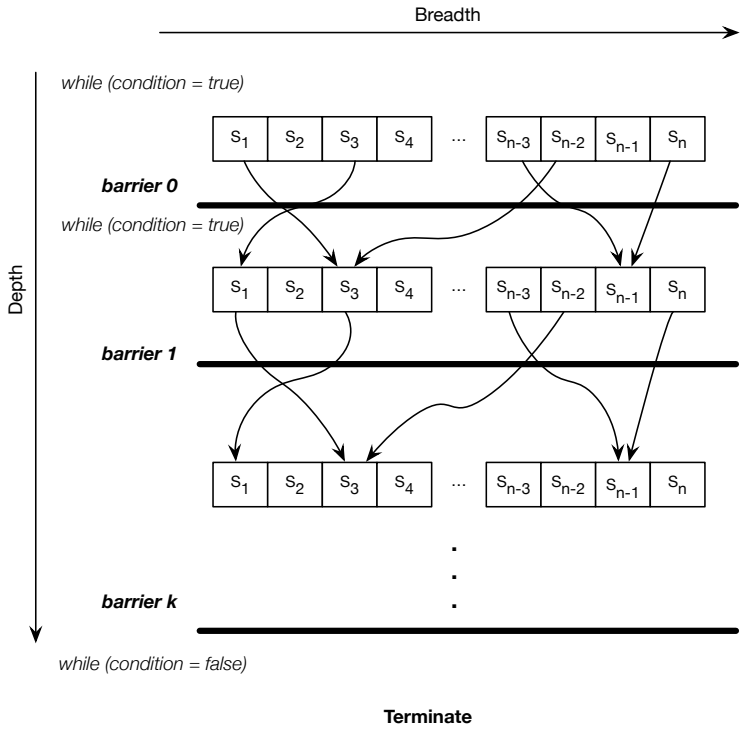


FIGURE 1.4. Parallel execution pattern of a shared-memory parallel graph algorithm

An example execution of an iterative shared-memory graph algorithm is shown in Figure 1.4. For this example, suppose every vertex keeps a state (S_i) in shared-memory and every vertex is processed in a separate parallel thread (n is the number of vertices). In an iteration, every thread reads a state from a previous state (updated by a previous iteration) and performs some calculation based on the algorithm logic and writes the outcome to the shared-memory. The algorithm terminates when the condition is evaluated to true. Between iterations, algorithm synchronizes all the processors. For this algorithm, depth is the number of iterations algorithm needs to perform until the condition is evaluated to true.

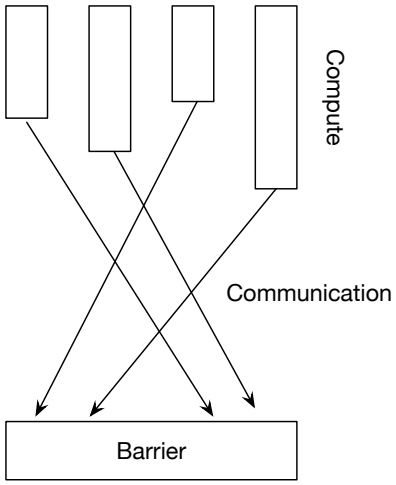


FIGURE 1.5. Bulk Synchronous Parallel Execution

1.1.4. Extending Shared-Memory Algorithms to Distributed-Memory. Parallel graph algorithms designed for shared memory do not immediately extend to distributed memory. One way to extend existing shared-memory graph algorithms to distributed memory is to use the Bulk Synchronous Parallel (BSP) [**132**] approach. The BSP model is regarded as a generalization of the PRAM model, which permits the frequency of barrier synchronization, and hence the demands on the routing network, to be controlled (See [**127**], Question 20 for details). The BSP model has *super-steps*. Each super-step consists of a local computation phase, a distributed communication phase, and a global synchronization phase (See Figure 1.5). However, when using the BSP approach to extend a graph algorithm designed for shared-memory to a distributed setting, the algorithm needs to be restructured into compute, communication and barrier synchronization phases and may need to introduce additional data structures too. Hence, the resulting distributed algorithm behaves differently from the original shared-memory algorithm both in the way it performs computations and in the way it uses the data structures.

For example, a shared-memory graph algorithm has access to the whole graph data structure whereas a distributed graph algorithm needs to perform communication to access a certain part of the data in the graph (assuming the graph structure is distributed).

Therefore, some of the techniques used by shared-memory algorithms create extra overhead in extended distributed algorithms. Sub-graph computation is a such an example. Barrier synchronizations is another example that becomes inefficient in distributed execution. In addition, some of the shared-memory parallel algorithms use techniques such as double pointer jumping to achieve better running time (e.g., Shiloach-Vishkin (SV)). However, such methods increase the number of distributed messages. In the following, we discuss the effect of these techniques on extended algorithms in detail.

1.1.5. Low Compute-Communication Ratio. Under a distributed runtime, time-steps to read a remote memory can no longer be counted as one as in the PRAM model. For distributed algorithms, it is necessary to reduce the remote memory accesses. Therefore, some of the shared-memory algorithms that have sound PRAM complexities show poor performance when they are extended as distributed-memory parallel graph algorithms. For example, SV Connected Components **[123]** algorithm uses techniques such as *double pointer jumping* and *tree hooking* to reduce the PRAM complexity (PRAM complexity $O(\log n)$ with $n+2m$ processors, where n is the number of vertices and m is the number of edges). In shared-memory, SV algorithm shows sound performance but in distributed-memory it generates very high number of remote messages. Figure 1.6 shows the performance of SV algorithm in distributed-memory. Within the node (i.e., when the number of cores is less than 16) algorithm shows better performance, but as we distribute the execution (after 16 cores) algorithm execution time increases.

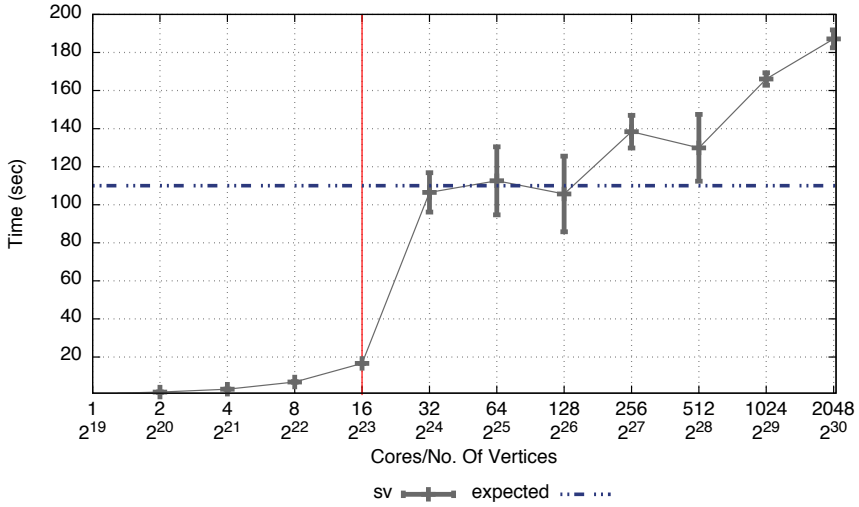
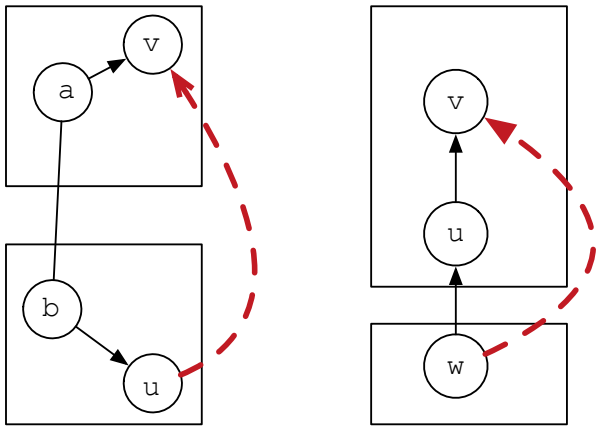


FIGURE 1.6. Weak scaling execution times for Shiloach-Vishkin Connected Components with RMAT-1 graph inputs. After 16 cores execution is distributed.

Both tree-hooking and double pointer jumping can trigger remote communication. Tree hooking is the process of connecting two parent vertices when their children are connected with an edge (Figure 1.7a). Double pointer jumping connects a vertex to its grandparent (Figure 1.7b). When the graph is distributed, hooking vertices may be in two different nodes and may need message communication to complete the operation. In distributed execution, this operation generates more messages and increases the communication and reduces the computation. A similar argument applies to double pointer jumping. Therefore, even though these operations are efficient in shared-memory, they slow down the algorithm when it is executed in distributed-memory.

1.1.6. Sub-Graph Computations. In the PRAM model, sub-graph computations add a constant factor to algorithm execution time. In shared-memory, sub-graphs can be constructed without much overhead. However, sub-graph computations in a distributed environment are quite expensive. In a distributed setting there are two main approaches to compute sub-graphs: 1. Create the sub-graph by filtering vertices and edges locally, or 2. Create the local subgraph by instantiating a new local graph by adding the subset of vertices and the subset of edges. The first approach does not need much memory, but it breaks



(A) Tree hooking operation. (B) Double pointer jumping

FIGURE 1.7. Tree hooking and double pointer jumping in SV algorithm.

the ability to parallelly iterate over edges and vertices. The second approach allows parallel iteration over subgraph vertices and edges. However, it consumes space and takes more time to construct the subgraph data structure. Both the above approaches require additional synchronizations and the cost of the subgraph computation is proportional to the size of the input graph as well as to the number of distributed processors. Luby **[91]** proposed four randomized Maximal Independent Set (MIS) algorithms that use sub-graph computations. Figure 1.8 shows the performance of Luby's algorithm (B) in distributed execution. Luby proved that under the PRAM model, Luby(B) algorithm takes $EO(\log n)$ [1] time-steps with $O(m)$ number of processors (n - the number of vertices and m - the number of edges). However, as the experimental results presented in Figure 1.8 demonstrate, Luby's algorithms do not scale as expected when the number of distributed processors and graph scale are increased.

[1] $EO(k)$ denote "the expected values is O(k)."

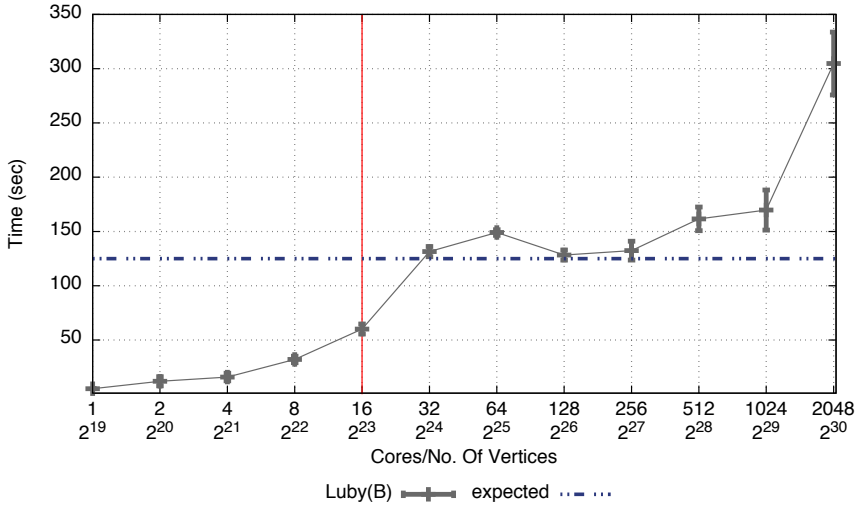


FIGURE 1.8. Weak scaling execution times for Luby's B MIS algorithm with RMAT-1 graph input. (After 16 cores execution is distributed.)

1.1.7. Iterative Graph Computations. A shared-memory parallel graph algorithm iterates over all vertices (or edges) in parallel until some condition is met. The basic structure of a shared-memory parallel graph algorithm is shown in Algorithm 1. Every iteration in the *while* loop involves parallelly iterating vertices (or edges). Further, a typical shared-memory algorithms maintain states against vertices or edges. These states are updated in every iteration in the *while* loop depending upon the logic. The state values for the second iteration of the *while* loop depend on the state values of the first iteration of the *while* loop. States are changed until the condition in the *while* loop evaluates to *false*. The execution pattern of a typical shared-memory graph algorithm is depicted in Figure 1.4.

Usually, the PRAM analysis calculates the number of steps needed for the "Depth", provided there is a "Breadth" number of processors. However, a shared-memory algorithm extended for distributed memory also needs to execute "k" number of barriers. The overhead of barrier synchronization is higher in distributed memory run-times than in shared-memory run-times. When a barrier is executed, in addition to the overhead of synchronization, it also creates *stragglers*, which waste the computation power of a subset of processors. In addition to barriers, iterative algorithms usually require a collective operation (e.g., min, max, add, multiply, etc.,) to evaluate the condition in the *while* loop. The

cost of collective operations is also higher in distributed runtimes than in shared-memory runtimes. When the number of distributed processors increases, the overhead of synchronization and collective operations also become significant.

2

Related Work

Graphs and graph algorithms have been studied for more than a century. There are widely used sequential graph algorithms with promising execution times for almost all the graph problems. However, most of these sequential graph algorithms are not directly parallelizable (there are exceptions such as Borouka's Minimum Spanning Tree (MST)). Parallel algorithms need to be developed from the scratch with a different view on the graph problem. Researchers started studying about parallel graph algorithms in 1980's. Since then there has been signinificant effort made on parallel graph algorithms for shared memory architectures. Theoritically it is possible to implement Parallel Random Access Memory (PRAM) algorithms as distributed memory parallel algorithms. However, not all PRAM-designed algorithms are guranteed to perform better in distributed memory parallel runtimes for reasons explained in Chapter 1.

Developing parallel graph algorithms is not straightforward and is usually much harder than designing a sequential algorithm. In fact, designing efficient distributed memory parallel graph algorithms is even harder. Therefore, a model for designing distributed memory parallel graph algorithm would benefit for algorithm designers.

Various programming models to write graph algorithms are proposed in graph processing frameworks, and there are numerous amount of graph processing frameworks (See [**35**] for a survey of graph processing frameworks). However, these programming models are not meant to used as graph algorithm design models, rather they are used

to implement existing graph algorithms. These programming models are engineering efforts built on top of PRAM or Bulk Synchronous Parallel (BSP). In Section 2.1, I review some of the widely available graph processing frameworks and programming models proposed by them. This thesis proposes several new algorithms based on the Abstract Graph Machine (AGM) framework. The work related to those parallel graph algorithms are discussed under Section 2.2. For shared memory graph algorithms spatial ordering has been studied in terms of schedulers. These scheduler abstractions are discussed in Section 2.3.

The performance model proposed in this thesis requires to model the graph algorithm, runtime and the input graph. The graph algorithm is modeled in terms of the AGM and/or Extended Abstract Graph Machine (EAGM). There are several abstract machine models for the runtime. Existing abstract machine models are discussed in Section 2.3.1. Input graph can be modeled in terms of the graph characteristics. Related work for modeling input graph characteristics is discussed in Section 2.3.5.

2.1. Graph Processing Frameworks

Pregel – Pregel [**94**] is a distributed graph processing framework that uses the BSP execution model. In Pregel, algorithm logic is expressed using the vertex-centric approach. In other, words, Pregel executes a function on a vertex locally and communicates state changes to distributed processors in a single super step. After each local computation and communication there is a barrier synchronization. The model of computation in Pregel and AGM are different in several ways. First, Pregel uses BSP approach while AGMs execution without any ordering on work is asynchronous. While Pregel is vertex-centric, AGMs are work driven. A work unit definition could be vertex centric, edge centric or possibly a collection of vertices. Further, the concept of ordering is not explicit in the Pregel programming model but AGM orderings are explicit. In summary, Pregel is proposed as a graph processing framework to express existing graph algorithms, but I propose an AGM as a model for designing families of distributed memory parallel graph algorithms. AGM algorithms achieve high performance in distributed systems by balancing synchronizing

overhead and local computations while avoiding calculations such as subgraph computations. In fact, AGM algorithms with a single equivalence class that have a vertex as the first element in a work unit can be implemented in Pregel in a single superstep.

Giraph – Apache Giraph [**25**] is an open source implementation of Pregel concepts. Therefore, the discussion carried out above applies to Giraph as well.

GraphLab – GraphLab [**90**] is an asynchronous distributed graph processing framework. GraphLab programming abstraction provides access to information on the current vertex, adjacent vertex and adjacent edges, irrespective of the edge direction. GraphLab essentially uses a Gather-Apply-Scatter (GAS) computation model. The Gather phase requires synchronous communication. Because of this, GraphLab programs inefficient when the graph is 1D distributed. However, GraphLab internally uses efficient scheduling (execution ordering) to reduce the amount of distributed communication. GraphLab and AGMs are different in several ways. First, AGM is not based on GAS primitives. Second, just like for Giraph and Pregel, ordering is not explicit in GraphLab abstraction while ordering is explicit in AGMs. Third, the AGM model does not rely on any synchronous communication.

There are many more distributed graph processing frameworks (See [**35**], Table I, for a list). Most of those graph processing systems use BSP or GAS as the programming model. However, it is not straightforward to implement existing graph algorithms on those models efficiently, mainly due to factors like synchronization overheads and increased distributed communication. Therefore, it is necessary to introduce graph algorithms that minimize synchronization overhead and reduce distributed communication.

2.2. Parallel Graph Algorithms

In this research, I study the internal behaviour of many available parallel graph algorithms. I did not find a well-defined classification of parallel graph algorithms based on the execution behaviour of existing parallel graph algorithms. However, I observed

that there are two classes of graph algorithms, they are: 1. iterative graph algorithms, and 2. stat-driven algorithms. The class of state-driven graph algorithms is introduced in this thesis (See Chapter 3), but some of the existing parallel graph algorithms are already state-driven. In some contexts, these algorithms are referred as *data-driven* algorithms, but I was unable to find a precise definition for data-driven algorithms; therefore, I use the term "state-driven graph algorithms" to identify algorithms that can be modeled using an AGM.

Most of the existing parallel graph algorithms designed for shared memory are iterative algorithms, meaning they iterate through vertices or edges in parallel and terminate the algorithm when it satisfies a certain condition. These algorithms further incorporate techniques such as subgraph computations (e.g., [**91**]), set operations on vertices or edges (e.g., [**44**], [**106**]). In these algorithms, the available parallelism is defined by the iterating variable, (e.g., number of vertices or number of edges).

State-driven algorithms implicitly define a data dependency by the structure of the graph. Usually these algorithms maintain a state at the vertex level and state changes are pushed on edges to other vertices. Dijkstra's Single-Source Shortest Paths (SSSP) algorithm is an example of a state driven algorithm. Further, most of the existing Single Source Shortest Path algorithms are state-driven algorithms (except Bellman-Ford SSSP [**15**]). State-driven graph algorithms can be further divided into two sub-categories:

(1) Label setting algorithms.

(2) Label correcting algorithms.

Ahuja et al. [**3**] used the above classification to categorize graph traversal algorithms. However, we can use above classification to study state-driven algorithms (since state-driven algorithms also use a form of a traversal). The label essentially represents the state associated with a vertex or an edge. In label setting algorithms, a vertex state is changed only once, but in label correcting algorithms the label can be set many times. However, at the end of the execution, the algorithm assures states are correct. Another orthogonal way to look at state-driven algorithms is the order in which labels are getting updated. If

a label setting or label correcting algorithm requires a specific order on how those labels are set, then we call such algorithms *ordered* algorithms. However, if the execution order is irrelevant we call those algorithms *un-ordered* algorithms.

The un-ordered, state-driven algorithms are the main focus of this research. The AGM algorithms (state-driven) are a subclass of un-ordered algorithms. The algorithms I derive are either label-setting or label-correcting. AGM algorithms start with an initial set of work units. Then, the algorithm execution creates a Directed Acyclic Graph (DAG) starting from initial vertex set or edge set encapsulated in work. Because of the un-ordered characteristic, there is more than one way to construct execution DAGs. A suitable ordering defines an efficient execution of the DAG.

Un-ordered algorithms are not common for many graph applications. There are un-ordered algorithms for graph applications such as Single Source Shortest Paths and Breadth First Search. However, such algorithms are not widely available for graph applications such as Connected Components, Maximal Independent Set or Strongly Connected Components. As part of this research, I show that existing un-ordered algorithms for a particular graph application can be generalized using the AGM framework. For graph applications that do not have state-driven graph algorithms, new algorithms can be derived. In the following, I briefly review previous work for algorithms I introduced/introducing in this research.

2.2.1. Breadth First Search. Breadth First Search (BFS) graph problem involves visiting every vertex in the graph from a given source vertex. Unlike Depth First Search (DFS), BFS is easily parallelizable. A simple parallel solution is to visit vertices in parallel. A more sophisticated approach is to visit vertices level by level and vertices in each level are visited parallely (aka level-synchronous BFS [**21**]). The PRAM algorithm for BFS is a straightforward extension of the sequential BFS algorithm. Multi-threaded solutions for BFS problems are discussed in [**10**, **102**]. These algorithms are mostly based on level-synchronous BFS discussed in [**21**]. The same level-synchronous algorithm is also used in GPU systems in [**63**, **92**, **140**]. The level-synchronous algorithm is further optimized to

different multi-core platforms and these efforts are discussed in [**1**, **81**, **138**]. Several distributed implementations of BFS are also available. Yoo et al. [**139**] discuss a BFS implementation on the BlueGene/L system. [**29**] discuss a BFS implementation for Partitioned Global Address Space (PGAS) systems and [**37**] discuss a BFS implementation on active message framework. Further, there are many BFS implementations on graph processing frameworks (e.g., Boost [**60**], Pregel [**94**] etc.,). The AGM formulation for BFS is straightforward. The original sequential algorithm can be directly modeled in an AGM. Further, we can see that level-synchronous BFS is a specialized version of sequential BFS, where it orders work by level.

2.2.2. Single Source Shortest Path. SSSP is a well-studied problem in the graph algorithm community. A number of sequential algorithms were introduced including Bellman-Ford SSSP algorithm [**14**], Gabow's SSSP Algorithm [**47**], Dijkstra's SSSP algorithm. Dijkstra's algorithm is the most popular sequential algorithm and several variations of Dijkstra's algorithm are also found in the literature; e.g., Dijkstra's algorithm with lists [**86**], Dijkstra's algorithm with Fibonacci heap [**46**], etc. All algorithms were developed based on the principles of relaxation and ordering, and then all can be modeled using an AGM.

Orderings in SSSP algorithms were studied by Crauser, Meyer, and Sanders for developing parallel SSSP algorithms. Crauser introduced a parallel version of Dijkstra's algorithm [**32**] that divides Dijkstra's SSSP algorithm into some phases so that each phase can be processed in parallel. Meyer researched adaptive bucket splitting for SSSP problem in [**99**]. Then, Meyer and Sanders presented Δ-Stepping which is another form of SSSP algorithm that operates in phases with parallel work. KLA uses levels to separate parallel processing phases. The above algorithms can also be used in distributed settings. Distributed implementations of Crauser's SSSP algorithm and Δ-Stepping SSSP can be found in *Parallel Boost Graph Library* [**39**].

2.2.3. Connected Components. Computing connected components is another well-studied problem. The existing sequential algorithms to find connected components either

use BFS or DFS [**117**]. While BFS based connected components algorithms can be parallelised, DFS based algorithms are not directly parallelizable.

Over the past 2-3 decades many parallel connected components algorithms were introduced focusing PRAM architectures. Chandra and Daniel [**66**] made an early attempt to solve the parallel connectivity problem in $O(log^2 n)$ time with n^2 processors. More parallel connectivity algorithms for PRAM are discussed in [**2, 8, 24, 26, 61, 68, 71, 74, 77, 88, 105, 108, 116, 123, 134**]. However, the above list is not complete. Out of the existing PRAM parallel connectivity algorithms, Shiloach-Vishkin CC algorithm is widely used. Shiloach-Vishkin (SV) algorithm requires $O(logn)$ iterations and needs $O(n+m)$ processors. However, SV algorithm requires $O((m+n)logn)$ amount of work. A linear work, polylogarithmic depth parallel algorithm was discussed in [**124**]. Another work-efficient connectivity algorithm is discussed in [**126**] for streaming graphs.

Most of the existing distributed memory parallel CC algorithms are either based on parallel search (BFS or DFS) or SV connected components algorithms. Edmonds et al. [**38**] presented an implementation of distributed SV algorithm using active messages. They also combined parallel search to optimize SV algorithms (called PS+SV). Srinivas et al. [**69**] presented a distributed memory parallel CC algorithm based on SV algorithm. [**69**] do several optimizations to reduce the communication volume and balance the load. Further, they optimize CC algorithm performance for short diameter graphs by running parallel BFS.

Two BSP-style connected components algorithms for the Map-Reduce paradigm is discussed in [**115**]. However, synchronization cost of these algorithms is higher than that of the algorithm discussed in Chapter 6. A detailed scalability analysis on five distributed connected components algorithms is carried out in [**27**]. All the algorithms discussed in [**27**] calculate connected components locally and then resolves global connectivity. Algorithms differ based on the way they resolve the global connectivity. To resolve global connectivity algorithms use five strategies: 1. All-reduce, 2. Union-find merging, 3. graph

contraction, 4. label propagation, 5. distributed union-find. The label-propagation algorithm discussed in [**27**] is similar to the algorithm discussed in Chapter 6, but it does not use ordering as method to reduce work.

2.2.4. Maximal Independent Set. Many parallel graph algorithms have been proposed to solve the Maximal Independent Set (MIS) problem. Most of the work is focused on the PRAM memory model; for example see algorithms in [**6**, **16**, **52**–**54**, **75**]. Some of these algorithms use randomization to break the symmetry. Algorithms that use randomization assume vertices are anonymous and do not have priorities on vertices or generate a new priority at every step.

Four related randomized algorithms were introduced by Luby [**91**]. Luby's algorithms select an arbitrary nonempty independent set from the original graph and remove all the selected vertices and their neighbors from the original graph. This process is repeated in iterations until all the graph vertices are removed from the original graph. Luby came up with four different ways to select an independent set (*select algorithm*) in an iteration (based on randomization). Every select function gives rise to a different MIS algorithm. These algorithms (namely A, B, C, D) are discussed in detail in [**91**]. Luby's algorithm steps are executed in a synchronized fashion, that is each step (inquiring neighbor states and updating current state) is carried out in a single lock-step. Further, the algorithm needs to synchronize after each iteration.

Luby did a detailed analysis of his algorithm for PRAM machine models. Later Luby's algorithm concepts are used to implement distributed versions. Lynch discusses a distributed version of Luby's algorithm for synchronous distributed networks in [**93**] (called LubyMIS). [**98**] presents a LubyMIS algorithm with improved communication message complexity. Fabian et al. [**78**] presented a deterministic distributed MIS algorithm. However, these algorithms assume a synchronous communication model and provides no experimental results. An asynchronous distributed memory parallel MIS algorithm was discussed under graph coloring problem in [**72**]. This algorithm is the basis for the AGM formulation presented in Proposition 5.

More recently, Blelloch et al. [**16**] showed that a trivial parallelization of the sequential greedy algorithm also called lexicographically first MIS, is in fact highly parallel (poly-logarithmic time) when the order of vertices is randomized. The algorithms discussed in [**16**] use a *priority DAG*: a directed acyclic graph (DAG) over the input vertices where edges are directed from higher priority to lower priority endpoints based on random values assigned to vertices. Each step adds the nodes with no incoming edges in the DAG to the MIS and removes them and their children from the priority DAG. This process continues until no vertices remain. However, they assumed shared memory execution and algorithm relies on computing a subset of vertices based on a predefined function to parallel execute greedy MIS. Further, each round and step in the algorithm needs to be synchronized, which may not be efficient in distributed execution.

2.2.5. Triangle Counting. Triangle counting has been a well-studied graph problem. There are a number of parallel distributed-memory, parallel shared-memory, and external memory (e.g., [**97**]) algorithms and implementations. Another classification of triangle counting is whether an algorithm is counting an exact number of triangles or making an approximation (e.g., [**76**]). In this paper our focus is distributed, shared-memory implementation of triangle counting algorithms that counts the exact number of triangles. Therefore, we limit our review to work that is more closely related to parallel triangle counting algorithms that count exact numbers of triangles.

Shared Memory : *Node-Iterator* [**121**] is an algorithm that iterates over all vertices and intersects adjacency lists of each pair of vertex neighbors to find the number of triangles. Green et al. [**57**] optimized this algorithm using vertex-covers. Green et al.also presented a multi-core implementation of Node-iterator in [**58**] and a GPU implementation in [**59**]. [**125**] mainly discuss two parallel cache-oblivious exact triangle counting algorithms for shared memory. The first algorithm merges sorted-directed adjacency lists of a vertex for intersection. The second algorithm uses a hash table to store edges and perform intersections. The paper shows that these algorithms can achieve better cache utilization. Madduri et al. [**109**] presented variations of triangle counting algorithms and

how they related to performance in shared-memory platforms. [**130**] also discuss number of optimizations to speedup sequential and shared-memory parallel triangle counting algorithms.

Distributed Memory : [**7**] discuss a Message Passing Interface (MPI) based distributed-memory parallel triangle counting algorithm called *PATRIC*. PATRIC is similar to the *predecessor, successor's predecessor* (PSP) algorithm described above. PATRIC uses vertex ids to partition neighbors of a vertex and performs ordering based on degree of a vertex as a pre-processing step.

A number of triangle counting algorithm implementations are available as part of graph processing run-times. PowerGraph [**55**] discusses a distributed-memory parallel triangle counting algorithm implemented based on gather-apply-scatter primitives. [**112**] discusses a triangle counting algorithm for distributed (external) memory. The algorithm is based on the asynchronous visitor queue approach presented in the paper and the algorithm is similar to PSP algorithm we discussed above. An extension of this work is presented at the static graph challenge [**111**].

[**9**] presents a Matrix based parallel triangle counting algorithm. To minimize the communication the paper also introduces a new matrix algebra primitive: *masked matrix multiplication* and uses Bloom filters [**17**] to minimize the overhead of communication. Several other linear algebra based triangle counting algorithms are discussed in [**137**], [**67**] and [**89**] (shared-memory). [**28**] and [**128**] discuss how triangle counting can be implemented on top of Map-Reduce frameworks. [**128**], Algorithm 3, finds triangles from the least degree vertex. The underlying idea behind [**128**], Algorithm 3 is similar to degree based vertex partitioning. However, our method is based on set intersection between adjacencies while [**128**] is based on a map reduce paradigm that transforms sets of tuples in multiple steps, including generating "special" sets with markers ("$"). Further, algorithms in [**128**] assumes the vertex degree of an adjacent vertex can be accessed as an attribute of the vertex. This requires additional space to store degree (or adjacencies) as a attribute. The algorithms presented in this paper does not assume remote vertex degree is readily available. [**110**]

discuss an another Map-Reduce triangle counting algorithm that avoids the redundant calculations using a classification method.

Many of the shared-memory triangle counting approaches discussed above are not directly applicable for hybrid memory systems due to high communication overhead. Further, the distributed triangle counting versions does not explore the optimizations that can be achieved using techniques such as vertex blocking, aggregation and involves heavy pre-processing methods.

2.2.6. Minimum Spanning Tree. There are three primary sequential algorithms to find MST of a given graph $G = (V, E)$. They are Prim's Algorithm, Kruskal Algorithm and Boruvka's Algorithm. Prim's algorithm [**31**, Chapter 23] starts with an arbitrary root vertex and grows the tree until it covers all the vertices. At each stage the algorithm adds an edge that is safe to form the minimum spanning tree; therefore Prim's algorithm is a *greedy algorithm*. Prim's algorithm operates similar to Dijkstra's algorithm to find shortest paths. Kruskal algorithm [**31**, Chapter 23] initially treats each vertex as a tree that is, it starts with a forest. Then, trees are connected using the minimum weight edge from one tree to another. Kruskal algorithm orders edges by weight and starts processing from the smallest weight edge. Neither Prim's algorithm nor Kruskal algorithm provides much data parallelism. Boruvka's Algorithm [**106**] provides more parallelim compared to either of the others. As in Kruskal's algorithm, Boruvka's algorithm also starts with a forest. But instead of iterating through ordered edges, Boruvka's algorithm finds the minimum weight edge between two components. If the algorithm finds such an edge, components are connected using the found edge. An implementation of Boruvka's algorithm uses *disjoint union* data structure. Most of the available PRAM algorithms are designed based on Boruvka's algorithm. Gallager, Humblet and Spira [**48**] introduced a distributed MST algorithm that is based on Boruvka's algorithm. This algorithm is also called *GHS*. The GHS algorithm is widely used to resolve distributed MST problem.

2.2.7. Strongly Connected Components. *Strongly Connected Component* is a subgraph of a directed graph where every vertex is reachable from every other vertex in the subgraph. Strongly Connected Components (SCC) partition the original graph. Tarjan [**129**] proposed an algorithm to find Strongly Connected Components using Depth-First search (DFS). But DFS provides little support for parallel execution. Most of the parallel SCC algorithms are based on Divide & Conquer Strongly Connected Components (DCSCC) proposed by Fleischer et al. [**44**]. More algorithms similar to DCSCC are discussed in [**122**] and [**96**]. Recently Amato et al. [**131**] proposed two Strongly Connected Components (SCC) algorithms based on DCSCC that perform equally well on all types of graphs. Also, there are algorithms that use matrix multiplication to find transitive closure (which is same as calculating SCC) of the graph [**11**,**120**]. All those algorithms are designed as iterative algorithms. Further graph algorithms that are based on DCSCC uses subgraph computations and set operations. To the best of our knowledge there is not a state-driven parallel graph algorithm to solve SCC problem.

2.2.8. Graph Coloring. The problem of *Graph Coloring* involves labeling vertices of a graph in such a way that no two adjacent vertices are labeled with the same color. There are many PRAM algorithms (see [**50**,**118**,**141**]) and few distributed memory parallel algorithms for graph coloring. Bozdağ et al. [**18**] discuss a framework to implement existing parallel greedy coloring algorithms in a distributed setting. Their framework is designed based on BSP paradigm. Another BSP style distributed memory parallel implementation on top of Hadoop is discussed in [**49**]. A distributed probabilistic algorithm is discussed in [**62**]. [**119**] discusses implementing coloring algorithm based on Luby's MIS on top of Pregel like systems. An efficient distributed memory parallel algorithm is discussed by Jones-Plassman in [**72**]. The AGM algorithm we propose is conceptually the same as the Jones-Plassman coloring heuristic with chaotic ordering. Alex et al. [**22**] discuss two graph coloring algorithms for shared memory. The first algorithm is a greedy iterative algorithm based on Bozdağ's algorithm and the second algorithm is the same as Jones-Plassman algorithm. The AGM processing function without ordering resembles the Jones-Plassman

algorithm. However, AGM framework can define more orderings and generate families of algorithms (For example order work by color, order work by monotonic distance from the start vertex or order work by level).

2.2.9. *k*-Core Decomposition. A *k-core* of a graph $G = (V, E)$ is a maximal connected subgraph of G where all the vertices have a degree of at least k. A k-core of a graph is obtained by recursively removing all vertices of degree smaller than k. Vertices are said to have *coreness k* if they belong to the k-core but not to the (k+1)-core. Vertices that belong to k-core also belong to 1-core, 2-core ... (k-1)-core. The problem *k-core decomposition* is to find the maximum coreness of every vertex in the input graph G. An $O(|E|)$ sequential algorithm for k-core decomposition is presented in [**12**]. A distributed memory parallel graph algorithm for k-core decomposition was first discussed in [**103**]. The algorithm proposed in [**103**] proves that information about coreness of neighbors is sufficient to decide the coreness of the current vertex. Based on this idea, algorithm first make an estimate of the coreness of vertices and then communicate that value to neighbors. At the same time a vertex receives estimates from neighbors and uses those estimates to re-compute its own estimate. This process continues until the algorithm converges. The AGM formulation for k-core decomposition is equivalent to the algorithm in [**103**] without ordering. With AGM formulation of this algorithm we can order work using estimated coreness values.

Another parallel Map-Reduce based algorithm is discussed in [**113**]. [**5**] presents another distributed algorithm designed based on [**12**]. An iterative algorithm for k-core decomposition on dynamic graphs is discussed in [**41**]. This algorithm works on partitioned data and assumes a graph can change by adding/deleting edges/vertices. Other efforts to find k-core decomposition in dynamic graphs are discussed in [**87**, **101**].

2.3. Spatial Characteristics

Related to ordering at spatial levels, there are graph processing frameworks that use spatial characteristics in the scheduler. However those are mostly implementations rather than formal models like AGM. One such framework that implement spatial level ordering

is Galois [**107**]. Though Galois functions in shared memory and is not directly applicable to distributed memory, I will discuss some of the spatial features used in Galois scheduler.

Galois is an iterator based abstraction for specifying graph algorithms. The iterator is defined over a set of work items. The body of the iterator carries the processing logic. Galois allows an application to specify an ordering and this ordering is considered when iterators are executed. Galois does not strictly follow the priority enforced, rather it attempts to do best effort ordering based on priority defined. The Galois scheduler, OBIM [**82**] is interesting because its relationship with the spatial features of the physical machine. The OBIM scheduler uses a sequence of bags where each bag corresponds to a priority order. When processing the current bucket, work may generated for earlier buckets and those tasks are scheduled preferentially over those in later bags.

Extended AGM enforces strict ordering on equivalence classes on the global level. But, partition (bag) ordering is "fuzzy" for inner nodes in the spatial hierarchy instance, similar to OBIM bags. Galois schedulers are predefined, and also, Galois does not support composing orderings as in Extended AGM.

2.3.1. Abstract Machine Models. The cost model of the AGM depends on two main parameters: 1. the underlying runtime, and 2. the input graph structure. In this subsection we discuss related work on underlying runtime models. The most common runtime used in parallel algorithm analysis is the PRAM. The analogous distributed memory runtime to PRAM is BSP. In addition, there are distributed memory models such as LogP. These models are discussed in detail in the following subsections.

2.3.2. PRAM. PRAM [**45**] is a multiprocessor model where every processor is connected to a shared memory. Every processor does its own processing and in-between processors data are shared by writing to/from the shared memory. Because of the shared memory, two or more processors can write to or read from the shared memory location at the same time. Based on how processors read/write from/to shared memory, PRAM machines are further classified as follows:

(1) Exclusive Read, Exclusive Write (EREW) : Two processors cannot read from the same shared memory location simultaneously. All processors can simultaneously read from distinct memory locations. Also, two processors cannot write to the same shared memory location simultaneously. All processors can write to distinct memory locations simultaneously.

(2) Concurrent Read, Exclusive Write (CREW) : All processors can read from any memory location simultaneously but processors are only allowed to write to distinct memory locations simultaneously.

(3) Concurrent Read, Concurrent Write (CRCW) : All memory locations can read from any memory location simultaneously and all processors can write to the same memory location simultaneously. This is the weakest model from available PRAM sub-categories.

The CRCW model has further subdivisions: 1. Priority CRCW; 2. Common CRCW; 3. Arbitrary CRCW. These subdivisions decide which value should be written to the shared memory location. In priority CRCW, processors have priority assigned and the processor with the highest priority writes to the shared memory. The common CRCW writes a value to shared memory if all the values are equal. In random CRCW, a randomly chosen value is written to the shared memory. Though there are several subdivisions of PRAM complexities, **[70]** shows that once we calculate PRAM complexity for one PRAM machine model, we can convert complexity value to other PRAM machine models.

PRAM is a simple, yet powerful model for analyzing algorithms. However, in PRAM all processors can access all memory cells in unit time. Therefore, PRAM ignores many factors that may become significant in distributed memory runtime environments (e.g., synchronization, communication cost). Therefore, PRAM complexity does not always reflect the correct cost of a distributed memory parallel algorithm.

2.3.3. BSP. BSP **[132]** is a machine model designed for distributed computations. In this model we assume following infrastructure;

(1) A set of processor-memory pairs

(2) A point to point communication network

(3) Barrier synchronization

The main unit of computation in BSP is the "superstep". In each superstep a program does local computations and then communicating local computation among processor and finally barrier synchronization. The algorithm performance is measured by counting the number of supersteps.

The BSP model is simple to program, independent of the target architecture, and the performance is predictable. Further, we can treat BSP as a generalized version of PRAM. The main disadvantage of BSP is the use of barrier synchronization. Though some believe barrier synchronizations are not significant, our results show that barrier synchronizations have a significant impact on performance of graph algorithms, especially at large scales.

2.3.4. LogP. LogP [**33**] is a more realistic distributed memory multiprocessor in which processors communicate by point to point messages. The model specifies performance characteristics of the network but does not assume the structure of the network.

The LogP model specifies following parameters:

(1) L : Latency – Delay incurred in communicating a message containing word from source to destination,

(2) o : Overhead – The length of time each processor is engaged in transmission or reception of each message,

(3) g : Gap – The minimum time interval between consecutive message transmissions or reception of each message,

(4) P : Processors – The number of processors.

Compared to BSP, the LogP model does not need a barrier synchronization; further, the LogP model takes into consideration factors such as overhead and gap. In BSP, overhead and gap parameters are not taken into consideration. However, it is shown that BSP can simulate LogP and LogP can simulate BSP.

As far as AGM is concerned, I am interested in overhead incurred in ordering, because different orderings give different performance results. However, when the overhead of

ordering is higher than the benefit of ordering we see poor performance results. Therefore, it is important to consider overhead in the computational model.

2.3.5. Graph Types. AGM algorithms are not iterative algorithms, in the sense these algorithms do not iterate through available vertices or edges. Available parallelism in AGM algorithms depends on ordering as well as on the graph structure. Therefore, we must consider the structure of the graph in the cost model. Particularly, we will study the relationship between different graph features and performance of distributed memory parallel state-driven graph algorithms. Some of the important features that affect performance of graph algorithms are graph diameter, degree distribution, number of connected components, size of the largest component. However, the impact of these features depend on the graph problem.

A broad range of graph structures have been studied. However, all of their structures are not directly explainable using mathematics. Therefore, we usually compare their structures with random graphs. Since, by the 1990s, there was a rich theory of random graphs. However, experimental studies has shown that random graphs do not resemble real world graphs. This is mainly because real-world networks are not homogeneous as we assume in random graphs. Random graphs studied up to 1990 were mostly homogeneous. Erdos & Renyi [**43**] and Gilbert [**51**] introduced random graph models around the same time. Many graph characteristics on those graphs have already well-studied. Especially Erdos & Renyi did an extensive study of random graph characteristics.

However, most of the real-world graphs has a few vertices with large number of outgoing edges. Such vertices are also called "hubs". Usually these graphs are refered as power-law or scale-free graphs. Researchers have also studied about the power-law random graphs and derived statistical conclusions about their features (See [**4**]).

There are graph generators that can generate more realistic power-law graphs. Two example generators are R-MAT [**23**] or Kronecker [**83**]. The R-MATmodel recursively assigns probabilities to partitions and based on those probabilities edges are added to the graph.

Kronecker model further generalizes graphs and generates graphs closely approximate to real world graphs interms of degree distributions and graph diameters.

3

Abstract Graph Machine

An *Abstract Graph Machine*(AGM) consists of a definition of a *Graph*, definition of a set *WorkItems*, an *initial workitem set*, a set of *states*, a *processing function* and a *strict weak ordering* relation. The Graph is the graph that will be processed by the algorithm. In addition to the standard vertex set and edge set, a graph definition also has the properties associated with edges and vertices. A *workitem* is the basic information unit that invokes the processing function. All the *workitems* generated by an algorithm are denoted using the *WorkItems*. *WorkItems* is a set, *workitem* is an element of that set and *workitems* refers to many elements of that set. The processing function consumes a *workitem* ($\in$ *WorkItems*) and may produce zero or more *workitems*. The processing function may change values associated with the states. The strict weak ordering relation orders *workitems* into a set of *ordered* (induced) *equivalence classes*. The interaction between the processing function and the ordering is graphically depicted in the Figure 3.1.

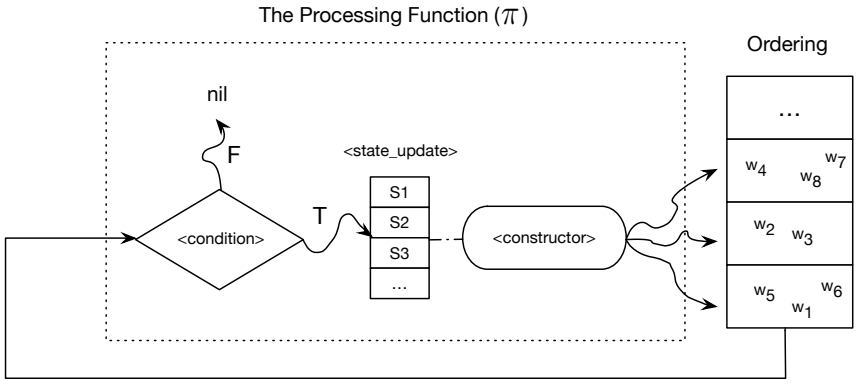


FIGURE 3.1. An overview of the Abstract Graph Machine

3.1. Model Primitives

3.1.1. Graph. A graph $G = (V, E, vmaps, emaps)$, where V is the set of vertices and $E \subseteq V \times V$, *vmaps* is a set of functions where each function is of the form $f : V \longrightarrow X$ and *emaps* is another set of functions where each function in *emaps* is in the form $f : E \longrightarrow X$. Here X represents some set of values. For example, a weighted graph in this notation is represented as $G = (V, E, \{\}, \{weight : E \longrightarrow \mathbb{R}\})$. AGM states are maintained as *mappings* (functions). The domain of the state mappings is the set V. The co-domain depends on the possible values that can be held in states. For example, in an SSSP algorithm the state mapping is $distance : V \longrightarrow \mathbb{R}$.

3.1.2. WorkItems. An AGM graph algorithm associates a state(s) with every vertex (or edge) in the graph. A change in the state associated with a vertex (or edge) may require a state change to states associated with adjacent vertices or incident edges. Such a requirement is indicated to adjacent vertices/incident edges through a *work item* (*workitem*). A *workitem* carries information about the vertex/edge to be acted upon and a corresponding set of values. The values may carry changes related to states or they may be used in defining the ordering. The AGM model denotes a *workitem* ($\in$ *WorkItems*) as a *tuple*. A tuple's first element stores a vertex/edge and the rest of the positions store values. As mentioned earlier, values may carry state changes/states or some value used in ordering. To access tuple elements, the model uses the *bracket operator*; e.g., if $w \in$ *WorkItems* and if $w = <v, p_0, p_1 \ldots, p_n>$then $w[0]$ = v and $w[1] = p_0$ and $w[2] = p_1$, etc.,. If a particular type of value is used for ordering, we say that value type is an *ordering attribute*. For example, the Dijkstra's SSSP algorithm contains a vertex and a distance in a *workitem*. The distance value will be used in updating the state associated with the vertex. A K-Level Asynchronous (KLA)-SSSP algorithm *workitem* has a vertex, a distance and a level. In this *workitem* definition, the level is an ordering attribute, but the role of the distance is same as Dijkstra's SSSP *workitem*. The *workitem* values are populated by the processing function. The size of the tuple (i.e., the number of additional elements) is determined by the states in the algorithm and the ordering attributes used in the AGM formulation.

3.1.2.1. *Processing Function.* A processing function (π) takes a *workitem* as an argument and may produce more *workitems* based on the logic defined inside the π. Mathematically π is declared as $\pi : WorkItems \longrightarrow 2^{WorkItems}$. The body of the processing function consists of logic to update states and generate new *workitems*. Prior to the state update logic there is a condition that evaluates the applicability of current *workitem* to a state change.

In the mode a π contains a set of *statement (St)s*. A statement contains a *condition* based on input *workitem* ($C : WorkItems \longrightarrow \{True, False\}$), an *update to states* ($U : WorkItems \longrightarrow \{True, False\}$) information on how an *output workitem* (w_{new}) should be constructed ($N : WorkItems \longrightarrow 2^{WorkItems}$). In an St the condition (C) is evaluated first; if condition evaluates to $True$, U is evaluated, and if both C and U are evaluated to true, then N is invoked. The function C says for a given *workitem* whether current St is applicable and the U evaluates to $True$ if states are changed when processing the input *workitem*. States are not explicit parameters to processing functions. The AGM model treats states as *side effects*. Therefore, when executing U we need to be extra cautious if updates can take place concurrently. We discuss this in detail under *data distribution* (Section 4.2). The difference between U and C is that U may have side effects when it updates states, but C does not create any side effects. The processing function is formally defined in Definition 1.

DEFINITION 1. $\pi : WorkItems \longrightarrow 2^{WorkItems}$

$$\pi(w) = \bigcup_{s_i \in St} s_i(w)$$

where; $s_i : WorkItems \longrightarrow 2^{WorkItems}$

$$s_i(w) = \begin{cases} \{w_{new} | w_{new} \in N(w) \\ \quad \textit{if C (w) is True \& U(w) is True}\} \\ \{\} \textit{else} \end{cases}$$

The definition of the processing function for a given graph algorithm involves defining statements. In other words, we need to define, for each statement, the condition (C), state

update (U), and the *workitem* constructor (N). For brevity, we use the following notation to represent the processing function:

NOTATION 1. $\pi : WorkItems \longrightarrow 2^{WorkItems}$

$$\pi(w) = \begin{cases} \{w_1 | w_1 \in < N_1(w) >, \\ \quad < U_1(w) >, \\ \quad < C_1(w) > \} \cup \\ \{w_2 | w_2 \in < N_2(w) >, \\ \quad < U_2(w) >, \\ \quad < C_2(w) > \} \cup \\ \quad \vdots \\ \{w_n | w_n \in < N_n(w) >, \\ \quad < U_n(w) >, \\ \quad < C_n(w) > \} \end{cases}$$

In Notation 1, w_i is the new *workitem* generated from N_i. As discussed previously, U_i and C_i represent state update and the condition. Total work generated by w is the union of work generated by all statements.

The processing function first executes the work in the initial *workitem* set. When processing the *workitems* in the initial *workitems* set, more work can be generated. The newly generated work is ordered and fed back again to processing functions. The vertex in every *workitem* is reachable from at least one *workitem*'s vertex in the initial *WorkItems*. In other words, the order how vertex states are changed can be represented using a Directed Acyclic Graph (DAG). Sources of the DAG are the vertices encapsulated in the initial *workitems* set (Figure 3.2). The processing function logic should be defined in such a way that each execution of a *workitem* converges the algorithm state/s. This behaviour is important to assure the termination of an AGM algorithm.

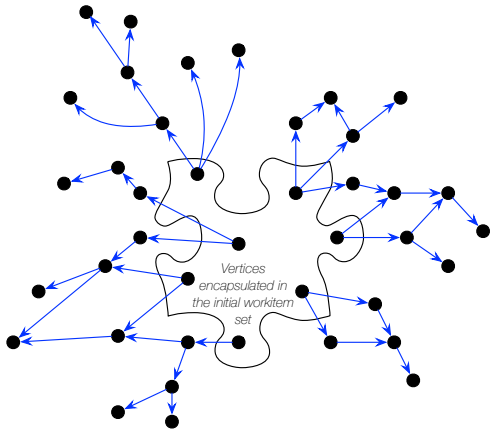


FIGURE 3.2. DAGs created during an AGM algorithm execution. Sources are elements in the initial *WorkItems*.

3.1.3. The Strict Weak Ordering Relation. The strict weak ordering relation (denoted by $<_{wis}$) must satisfy the following properties:

(1) For all $w \in \mathit{WorkItems}$, $w \not<_{wis} w$.

(2) For all $w_1, w_2 \in \mathit{WorkItems}$, if $w_1 <_{wis} w_2$, then $w_2 \not<_{wis} w_1$.

(3) For all $w_1, w_2, w_3 \in \mathit{WorkItems}$, if $w_1 <_{wis} w_2$ and $w_2 <_{wis} w_3$, then $w_1 <_{wis} w_3$.

(4) For all $w_1, w_2, w_3 \in \mathit{WorkItems}$, if w_1 is not comparable with w_2 and w_2 is not comparable with w_3, then w_1 is not comparable with w_3.

Properties 1 and 2 state that the strict weak ordering relation is **not** *reflexive* and is *anti-symmetric*. Property 3 denotes the *transitivity* of the "comparable *workitems*", and Property 4 states that transitivity is preserved among non-comparable elements in the *workitem* set. These properties give rise to an *equivalence* (i.e. non-comparable *workitems* belong to the same equivalence class) relation defined on *WorkItems*; hence, the strict weak ordering relation partitions the complete *WorkItems*. For example, in Figure 3.3, *workitems* in "A" are not comparable to each other, but a *workitem* in "A" and a *workitem* in "B" are comparable using the strict weak ordering relation.

Since *workitems* in different equivalence classes are comparable, the strict weak ordering relation defined on the set *WorkItems* induces an ordering on generated equivalence

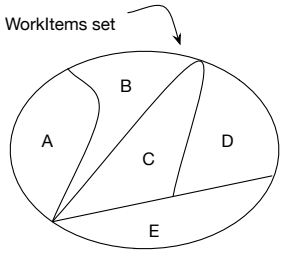


FIGURE 3.3. Partitioning *WorkItems* into equivalence classes.

classes. In general, there are several ways to define the induced ordering relation (denoted $<_{WIS}$). For work presented in this thesis, we use the definition given in the Definition 2.

DEFINITION 2. *$<_{WIS}$ is a binary relation defined on $2^{WorkItems}$, such that if $W_1, W_2 \in 2^{WorkItems}$ then; $W_1 \leq_{WIS} W_2$ iff*
forall $w_1 \in W_1$ and forall $w_2 \in W_2; w_1 <_{wis} w_2$.

3.1.4. The AGM. Having defined all supporting concepts, we now give the definition of an AGM in Definition 3.

DEFINITION 3. *An* Abstract Graph Machine*(AGM) is a 6-tuple (G, WorkItems, Q, π, $<_{wis}$, S), where*

(1) *$G = (V, E, vmaps, emaps)$ is the input graph,*
(2) *$WorkItems \subseteq (V \times P_0 \times P_1 \cdots \times P_n)$ where each P_i represents a state value or an ordering attribute,*
(3) *Q - Set of states represented as mappings,*
(4) *$\pi : WorkItems \longrightarrow 2^{WorkItems}$ is the processing function,*
(5) *$<_{wis}$ - Strict weak ordering relation defined on workitems,*
(6) *S ($\subseteq$ WorkItems) - Initial workitem set.*

In the following, we give the semantics of an AGM. An AGM starts execution with the *initial workitem set*. The initial *workitem* set is ordered according to the strict weak ordering relation. Next, the *workitems* within the smallest equivalence class are fed to the processing function. If the processing function generates new *workitems*, then they are separated into equivalence classes based on the strict weak ordering relation. The *workitems*

within a single equivalence class can execute the processing function in parallel. However, *workitems* in two different equivalence classes must be ordered according to the induced relation (i.e. $<_{WIS}$). When executing *workitems* in an equivalence class, AGM may generate new *workitems* for the same equivalence class or for a different equivalence class. The AGM executes *workitems* in the next equivalence class once it finishes executing all the *workitems* in the current equivalence class. The next equivalence class is the equivalence class greater than current equivalence class as per $<_{WIS}$ relation. However, if all the equivalence classes greater than current equivalence class are empty, AGM falls back to the smallest equivalence class generated. An AGM terminates when all equivalence classes are empty.

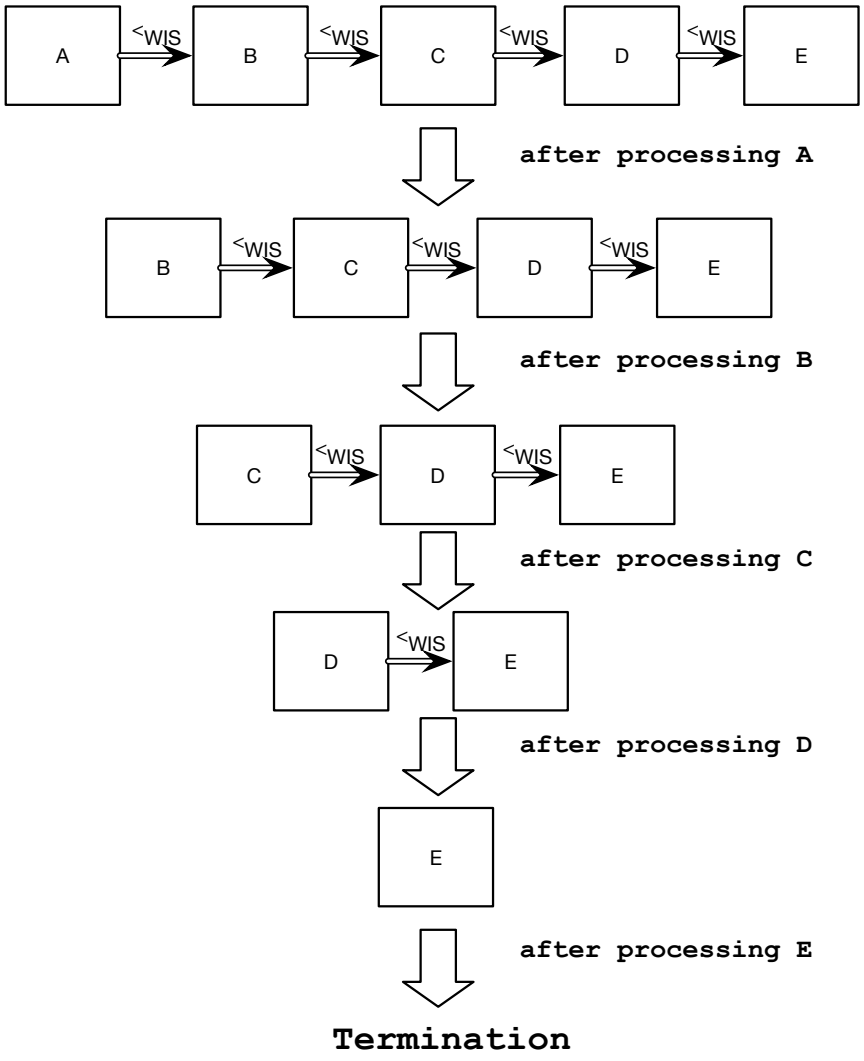


FIGURE 3.4. AGM executing equivalence classes generated in Figure 3.3.

Figure 3.4 shows how an AGM executes equivalence classes generated by the strict weak ordering relation. Figure 3.4 uses the equivalence classes generated in the example shown in Figure 3.3. Suppose the equivalence classes are ordered as follows; $A <_{WIS} B <_{WIS} C <_{WIS} <_{WIS} D <_{WIS} E$. AGM first executes *workitems* in A and when A is empty it is removed from the list of equivalence classes and moves to execute *workitems* in B. When B is empty, B is removed from the equivalence class list and moves to execute *workitems* in C, and so on. When there are no more equivalence classes to execute AGM terminates.

3.2. Termination & Correctness

Correctness of an algorithm assures that the output of the algorithm is the desired solution for the given graph problem. In an AGM algorithm, at termination, states contain the desired output and state is changed by the relevant π. Therefore, both the correctness of the algorithm and the termination of the algorithm depends on the definition of the π.

The π logic must be defined in such a way that every time a *workitem* is executed, the state is converged or unchanged. For a correctly defined processing function logic an asynchronous algorithm converges without any ordering on *workitems*; i.e., the algorithm converges with the chaotic ordering. Further it is easy to see that if algorithm converges without any ordering on *workitems*, then algorithm must converge with any ordering.

The π logic must also assures that new *workitems* are generated only when the states are changed. When algorithm states converge, new *workitems* are not generated, hence there will not be work inserted to equivalence classes and because of that AGM execution terminates.

3.3. Breadth First Search in AGM

The BFS graph application visits every vertex in the graph starting from a given source vertex. In the following, I present AGM models for level-synchronous BFS [**21**] algorithm, KLA BFS [**64**] algorithm, and parallel search based BFS algorithm.

3.3.1. Level-Synchronous BFS Algorithm. The level-synchronous breadth first search algorithm uses data structures to store the *current* and *next* vertex frontiers. Then the next container data are swapped with current after processing each level.

In the following, I model level-synchronous BFS with an AGM. The level-synchronous BFS order work by the level in the resulting BFS tree. Therefore, the ordering attribute that we are interested in is the *level*; hence I define $WorkItems^{bfs} \subseteq V \times Level$ where $Level \subseteq \mathbb{N}$. The processing function for BFS is defined in Definition 4. The state of the BFS algorithm is maintained in a map ($vlevel : V \longrightarrow Level$) that stores the level associated with each vertex. An infinite value (very large value) is associated with each vertex at the start of the

algorithm. Then the infinite value is changed as the algorithm traverses through the graph level by level.

DEFINITION 4. $\pi^{bfs} : WorkItems^{bfs} \longrightarrow 2^{WorkItems^{bfs}}$

$$\pi^{bfs}(w) = \begin{cases} \{w_k | w_k \in < w_n | w_n[0] \in neighbors(w[0]) \text{ and} \\ w_n[1] \longleftarrow w[1] + 1 >, \\ < vlevel(w[0]) \longleftarrow w[1] >, \\ < (if\ vlevel(w[0]) < \infty) > \} \end{cases}$$

The strict weak ordering relation arranges *workitems* based on the level. If two *workitems* have different levels they belong to different equivalence classes, and if they have the same level, they belong to the same equivalence class. The strict weak ordering relation for level-synchronous BFS is given in Definition 5, and Proposition 1 gives the AGM formulation.

DEFINITION 5. *$<_{ls}$ is a binary relation defined on $WorkItems^{bfs}$ as follows: Let $w_1, w_2 \in WorkItems^{bfs}$, then; $w_1 <_{ls} w_2$ iff $w_1[1] < w_2[1]$.*

PROPOSITION 1. *level-synchronous BFS Algorithm is an instance of an AGM where;*

(1) *$G = (V, E, vmaps = \{\}, emaps = \{\})$ is the input graph,*

(2) *$WorkItems = WorkItems^{bfs}$,*

(3) *$Q = \{ vlevel \}$ is the state mapping and initially $\forall i \in V, vlevel(i) = \infty$,*

(4) *$\pi = \pi^{bfs}$,*

(5) *The strict weak ordering relation $<_{wis} = <_{ls}$,*

(6) *$S = \{<v_s, 0>\}$ where $v_s \in V$ and v_s is the source vertex.*

3.3.2. KLA BFS Algorithm. The KLA BFS algorithm and level-synchronous BFS algorithm differs in terms of the way they order *workitems*. The level-synchronous BFS algorithm orders work by a single level while the KLA algorithm can go up to $K(\in \mathbb{N})$ levels. Therefore, the AGM formulation for KLA BFS is same as the AGM given in Proposition 1 except the strict weak relation ($<_{wis}$) is given by Definition 6.

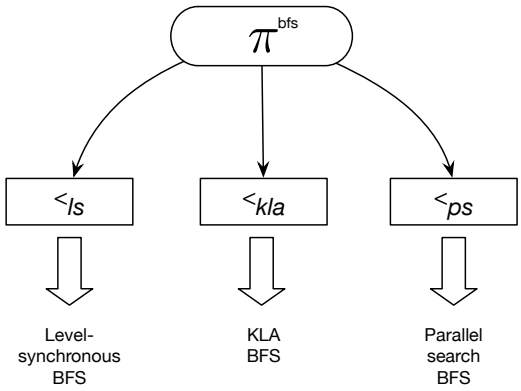


FIGURE 3.5. Summary of AGMs for BFS algorithms.

DEFINITION 6. *$<_{kla}$ is a binary relation defined on WorkItems^{bfs} as follows: Let $w_1, w_2 \in$ WorkItems^{bfs}, then; $w_1 <_{kla} w_2$ iff $\lfloor w_1[1]/K \rfloor < \lfloor w_2[1]/K \rfloor$.*

3.3.3. Parallel Search BFS Algorithm : The parallel search based BFS algorithm does not perform ordering on *workitems* and the algorithm freely traverse through un-visited vertices. As in KLA BFS, for parallel search BFS the AGM formulation given in Proposition 1 is valid except for its strict weak ordering relation. The strict weak relation for parallel search BFS says no two *workitems* are related. Therefore, the appropriate strict weak ordering relation is given in Definition 7.

DEFINITION 7. *$<_{ps}$ is a binary relation defined on WorkItems^{bfs} as follows: For any $w_1, w_2 \in$ WorkItems^{bfs}, $w_1 \not<_{ps} w_2$ nor $w_2 \not<_{ps} w_1$.*

Figure 3.5 summarizes BFS algorithms discussed above. All the algorithms discussed can be expressed as a processing function (π^{bfs}) and an ordering. Every different ordering generates a new algorithm. Further, we can generate new algorithms by applying different orderings.

3.4. Summary

The AGM model expresses a graph algorithm as a function that encapsulate logic to generate and consume work units and a strict weak ordering relation that orders those

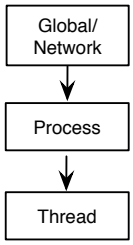
work units. The termination and the correctness of the algorithm only depends on processing function logic and ordering only affects how fast an algorithm converges.

4

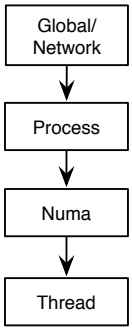
Extended Abstract Graph Machine

AGM does not distinguish whether an algorithm is shared memory or distributed memory. Whether a state-driven graph algorithm is shared memory or distributed memory is decided based on the type of the spatial memory available to an algorithm. In a system, spatial memory is organized as a hierarchy. Architecture dependent algorithms can be derived by imposing orderings at different spatial levels. We extend AGM to specify *spatial* orderings for a given state-driven graph algorithm and for a given architecture with a spatial memory hierarchy.

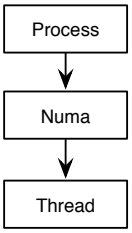
4.1. Memory Hierarchy



(A) Distributed memory program with uniform memory access in each process.



(B) Distributed memory program with non-uniform memory access in each process.



(C) Shared memory program with non-uniform memory access.

FIGURE 4.1. Spatial hierarchies of three different systems.

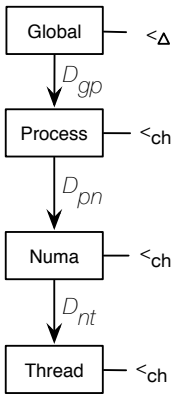
Figure 4.1 shows three different spatial memory hierarchies available in three different systems. Figure 4.1a and Figure 4.1b show memory hierarchies available in two different distributed memory systems. The *Global/Network* memory is the memory available to any process by sending/receiving messages over the network. The memory available to a local process is denoted using *Process* and memory accessible to a single thread is depicted using *Thread*. Figure 4.1b additionally has *Numa* that represents memory accessible to a particular numa domain. The last figure (Figure 4.1c) shows the memory hierarchy of a shared memory system. Note this shared memory system does not have the Global/Network memory level.

An *extended AGM* orders *workitems* in the root level of the memory hierarchy according to the ordering defined in the corresponding AGM. Further, EAGM defines ordering to be performed at other levels of the spatial hierarchy. When *workitems* propagate from higher levels in the memory hierarchy to lower levels, they need to be distributed. An EAGM also defines how to distribute *workitems* when they are sent from a higher memory level to a lower memory level in the spatial hierarchy. A *distribution function* (D) decides how *workitems* in a higher memory level are distributed to a lower memory level. In summary, an EAGM consists of following:

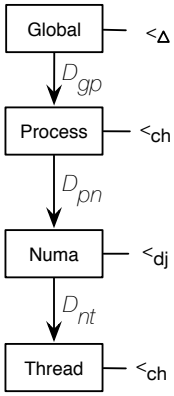
(1) A spatial hierarchy of an architecture,

(2) Orderings annotated to each level in the memory hierarchy,

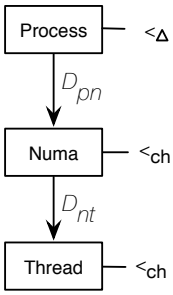
(3) Distribution functions attached to every down arrow in the spatial hierarchy.



(A) Distributed memory, Δ-Stepping EAGM.



(B) Distributed memory, Δ ordering at root and dijkstra's ordering at numa level EAGM.



(C) Shared memory, Δ-Stepping EAGM.

FIGURE 4.2. Three different EAGMs.

Figure 4.2 shows three different EAGMs for SSSP application. The first two are distributed EAGMs and the first EAGM is the distributed Δ-Stepping algorithm. The second EAGM (Figure 4.2b) performs ordering similar to distributed Δ-Stepping except that it performs a different ordering at the numa level. At the numa level, the EAGM in Figure 4.2b performs ordering according to $<_{dj}$. Definitions for strict weak ordering relations used in Figure 4.2 are given in Section 5. The last figure (Figure 4.2c) shows an EAGM corresponding to the shared memory Δ-Stepping algorithm.

In addition to the strict weak orderings these EAGMs also have a distribution function attached to each arrow from a higher level in the memory hierarchy to a lower level. In general, distribution function is declared as $D : \mathit{WorkItems} \longrightarrow \mathbb{N}$. Given a *workitem* in a higher memory level, the function D returns an id that can be used to identify the lower level memory locality. The *workitem* is routed to the memory locality identified by the id. For example, given a *workitem* the function D_{gp} returns an id of the process (also called *rank*) where the *workitem* should be routed.

Figure 4.3 shows how EAGM *workitem* distribution is taking place. The EAGM has Δ ordering at the root and Dijkstra's ordering at the numa level. The *workitems* in the

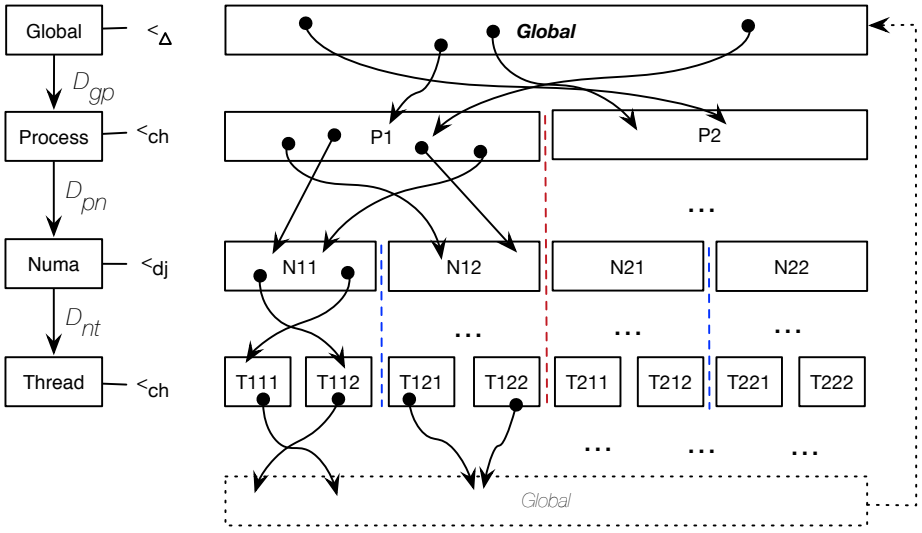


FIGURE 4.3. EAGM Spatial Hierarchy. At the global level *workitems* are ordered according to $<_\Delta$ and at process level and thread level there is no ordering, but at the numa level *workitems* are ordered according to Dijkstra's relation.

Global spatial level is divided between two processes (using D_{gp}) and each process divides *workitems* among two numa domains (using D_{pn}). Then, each numa domain divides *workitems* between two threads using D_{nt}. Typically, distributed algorithms define deterministic distribution functions to separate *workitems* between participating processes and distribution functions to route *workitems* between in-node spatial levels are random. In the following, we discuss two commonly used distribution function implementations.

4.2. Data Distribution

The data distributions define D functions in an EAGM. There are two commonly used distributions:

(1) 1-D distribution,

(2) Edge list distribution.

4.2.0.1. *1-D Distribution.* A 1-D distribution distributes vertices equally among participating processes. A distributed shared memory implementation of an AGM algorithm can implement 1-D distribution in one of two ways. It can either distribute vertices among participating processes (but without assuming ownership of a vertex to a thread, in which

case we need to take additional actions to make sure execution is consistent), or it can first distribute vertices among participating nodes and then further distribute vertices in a processor among available numa domains or threads. For the latter approach we do not need additional locks/local atomics when updating states. However, the latter approach may suffer from greater load in-balance than the initial approach.

4.2.0.2. *Edge List Distribution.* Edge list distribution is more suitable when an AGM algorithm uses an edge to route a *workitem*. In this distribution edges are distributed among participating processes, and, as in 1-D distribution, edges can be further distributed among participating nodes.

4.3. Spatial Ordering

Ordering in terms of a property or an attribute reduces the amount of redundant work in AGM algorithms. For example, from the SSSP algorithms discussed in the previous section, Dijkstra's algorithm performs the best ordering, in that it does the minimum amount of redundant work. However, the overhead of ordering in Dijkstra's algorithm is significant in a parallel distributed run-time due to the frequent synchronization. In other words, when the AGM instance generates more equivalence classes, the global synchronization overhead increases. The other SSSP algorithms discussed above reduce overhead of ordering by chunking *workitems* into larger equivalence classes. This reduces the number of equivalence classes generated and hence increases the available parallelism. When AGMs are mapped to an implementation, the ordering defined by the AGM is applied to the highest memory level. we have designed EAGM to apply ordering at lower spatial levels of a given architecture. Because of this, EAGMs reduce the amount of redundant work, but without affecting the ordering defined by the corresponding AGM. This way the corresponding AGM can generate larger equivalence classes to increase the parallelism and to reduce global synchronization overhead while using ordering at lower spatial levels to reduce the redundant work.

At lower spatial levels, *workitems* are ordered according to the strict weak relation attached to them. As in the AGM, the strict weak ordering relation separates *workitems* reaching lower spatial levels into equivalence classes. The orderings attached to the lower spatial levels are performed on *workitems* available to the memory in the relevant spatial level. Therefore, the orderings attached to lower spatial levels are more *relaxed* than the ordering attached to the root. By default the EAGM's spatial hierarchy assumes Chaotic (i.e. no ordering) ordering, but specific orderings can be enforced.

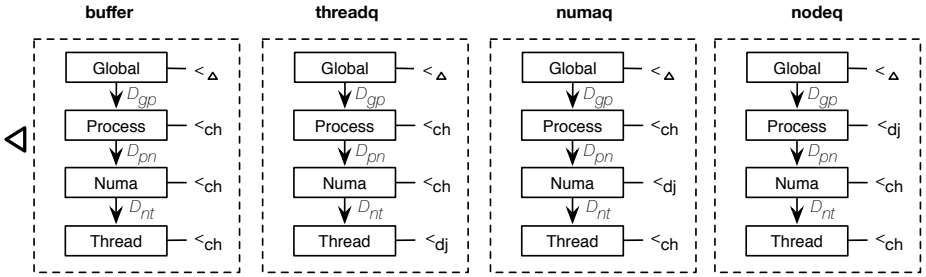
There are two ways to process *workitems* in lower spatial levels: 1. Always process non-empty smaller equivalence classes then process higher equivalence classes, 2. Process all the equivalence classes according to the induced ordering and visit again the smallest equivalence class to check whether there is more work. Do the same until there are no more *workitems* in equivalence classes. My implementations use the first approach since it is more effective for label correcting applications such as SSSP.

4.3.1. Applications. While EAGM approach is applicable to all AGM algorithms, we use SSSP as a case study to show how architecture dependent algorithms can be generated using the EAGM. Of the SSSP algorithms discussed in Section 5, fine-grained spatial ordering is effective for AGMs defined for Δ-stepping, KLA and Chaotic. By considering the spatial hierarchy used in Figure 4.1b, we apply Dijkstra's strict weak ordering relation (Definition 10) to spatial hierarchy levels Process, Numa and Thread to derive EAGMs (Figure 4.4). Each EAGM generates a variation of the main algorithm defined by its corresponding AGM.

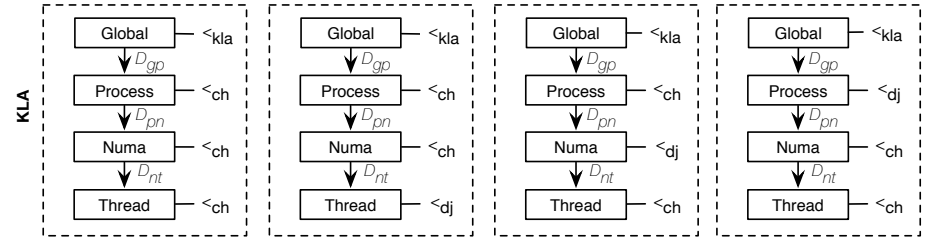
Figure 4.4a shows EAGM variations derived for Δ-stepping algorithm. Figure 4.4a-(i), applies $<_{dj}$ ordering to Thread level and Figure 4.4a-(ii) applies $<_{dj}$ ordering to Numa level and Figure 4.4a-(iii) applies ordering to Process level. EAGMs for KLA and Chaotic are derived in the same way.

4.4. Summary

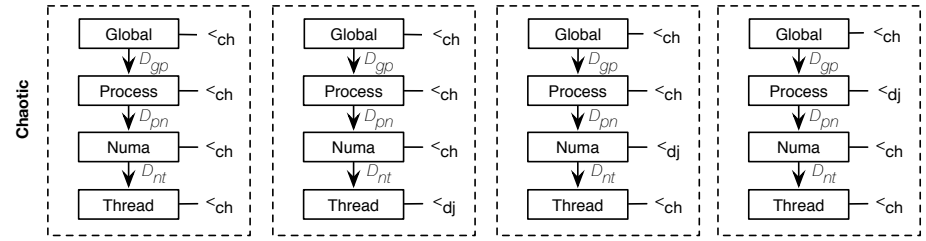
AGM is an abstract model that can express parallelism in graph algorithms. However, AGM does not specify how an AGM algorithm can be mapped to the memory hierarchy



(A) EAGMs of Δ-stepping AGM



(B) EAGMs of KLA AGM



(C) EAGMs of Chaotic AGM

FIGURE 4.4. Thread ordered, NUMA ordered and Process ordered EAGMs for Δ-stepping, KLA and Chaotic AGMs.

of a system. The Extended AGM maps the algorithm expressed in AGM to memory. The memory of a system is defined as a hierarchy, EAGM specifies further orderings at lower spatial levels of an architecture. The EAGM also specifies how *workitems* are distributed when a *workitem* is propagating from one processing function to another.

5

Families of Graph Algorithms: SSSP Case Study

Single-Source Shortest Paths (SSSP) is a well-studied graph problem. Examples of SSSP algorithms include the original Dijkstra's algorithm and the parallel Δ-Stepping and K-Level Asynchronous algorithms. In this chapter, we use AGM model to show that all these algorithms share a common logic and differ from one another by the order in which they perform work. We use the AGM model to thoroughly analyze the family of algorithms that arises from the common logic. We start with the basic algorithm without any ordering (*Chaotic*), and then we derive the existing and new algorithms by methodically exploring semantic and spatial ordering of work. Our experimental results show that new derived algorithms show better performance than the existing distributed memory parallel algorithms, especially at higher scales.

5.1. Introduction

Given a graph problem, how many ways can it be solved in? In this chapter, we consider the seemingly simple problem of *single-source shortest paths* (SSSP), where the task is to find the shortest path from a source vertex s to every other vertex in the graph. A number of sequential algorithms exist. The well-known Dijkstra's algorithm [**34**] is "work optimal',' where vertices are ordered in a priority queue based on their distance from the source s, and every edge is traversed only once. Work optimality, however, comes at a cost of limited parallelism and extensive synchronization. Subsequent development concentrated on relaxing the strict ordering of the Dijkstra algorithm to make more work available in parallel at the cost of some "wasted work" that has to be invalidated and repeated.

For example, the Δ-Stepping [**100**] algorithm groups vertices into Δ-sized *buckets*, based on their distances from the source s, giving an approximation of Dijkstra ordering. Vertices in a bucket are processed in parallel, and picking an appropriate Δ ensures the right balance between parallelism and wasted work. The *K-Level Asynchronous* [**64**] algorithm is similar, but it uses topological distances instead of shortest path distances from the source s to order work into buckets[1].

TABLE 5.1. Orderings in SSSP algorithms.

Algorithm	**Ordering**
Dijkstra's	Global priority queue
Δ-Stepping	Global distance equivalence classes defined by Δ
KLA	Global topological distance equivalence classes defined by k
Chaotic	None

ALGORITHM 2. The SSSP relax function

```
1: Input: Task (v,d), distances D
2: if d < D(v) then
3:   D(v) ← d
4:   ∀v_n ∈ neighbors(G,v) :
5:     Task(v_n, d_v + weight(v,v_n))
6: end if
```

In both Δ-Stepping and K-Level Asynchronous, processing of the buckets inserts implicit synchronization points, since processing of a bucket cannot begin until all previous buckets are finished. The *Chaotic SSSP* does away with synchronization altogether by processing all the vertices in parallel in an arbitrary order, resulting in maximum available parallelism at the cost of more wasted work.

Despite the variety of algorithms, analysis reveals that they are all based on the same core logic of *relaxation*, as shown in algorithm 2. Relaxation takes as the input a vertex-distance pair and a distance map (D), and produces more vertex-distance pairs if the distance was improved. These newly produced pairs are further relaxed, and the algorithms differ by how these relaxations are ordered (table 5.1). We methodically investigate this similarity between the seemingly different SSSP algorithms. To do that, we model the algorithms using the AGM

[1]K-Level Asynchronous with single-hop buckets is equivalent to the Bellman-Ford algorithm [**14**].

We present AGM models for all the algorithms listed in table 5.1, and we show that they change by the way in which they order work. Then we show that new algorithms can be developed by methodically discovering new orderings. We then use EAGM, to develop variations of algorithms presented in table 5.1 with additional ordering at different spatial levels of architecture such as node (process), numa (non-uniform memory access) region, and thread, resulting in *nine* different SSSP algorithms. We compare the weak scaling performance of the new algorithms with existing distributed memory parallel algorithms and also with the SSSP algorithm in PowerGraph [**55**] and in Parallel Boost Graph Library (Parallel BGL) [**40**] for a performance base line. Our results show that some of the new variations of SSSP algorithms perform better than the well-known algorithms, especially at large scales.

5.2. SSSP Algorithms in AGM

In this section, we present AGM models for algorithms discussed in table 5.1. To specify these models, we need to provide AGM elements from Definition 3. First, we provide the input graph, the *WorkItems*, the set of states, the processing function, and the initial *workitem* set. Then, we show that adding different orderings to the AGM models, we get existing distributed SSSP algorithms.

The input graph for the SSSP problem is a weighted graph: $G = (V, E, vmaps = \{\}, emaps = \{weight\})$. The basic *workitem* includes a vertex and its distance and *WorkItems* for SSSP is defined as $WorkItems^{sssp} \subseteq (V \times Distance)$. The basic *workitem* is extended by additional ordering attributes when necessary (e.g., in K-Level Asynchronous). The set of states includes a single mapping *distance* for storing the distance from the source vertex. The *distance* mapping is defined as $distance : V \longrightarrow \mathbb{R}$. The processing function for SSSP changes the distance state if the input *workitem*'s distance for a given vertex is less than what is already stored for that vertex in the distance map. The list of adjacent vertices of a given vertex are accessed through the $neighbors : V \longrightarrow 2^V$ function. The basic (it will be extended with additonal functionality for some algorithms) processing function for the SSSP graph problem is defined in Definition 8.

DEFINITION 8. $\pi^{sssp} : WorkItems^{sssp} \rightarrow 2^{WorkItems^{sssp}}$

$$\pi^{SSSP}(w) = \begin{cases} \{w_n | \langle w_n[0] \in neighbors(w[0]) \\ \qquad \text{and } w_n[1] = w[1] + weight(w[0], w_n[0]) \rangle, \\ \langle distance(w[0]) \longleftarrow w[1] \rangle, \\ \langle if\ w[1] < distance(w[0]) \rangle \} \end{cases}$$

The SSSP processing function (π^{sssp}) has a single statement. The statement is executed only if the input *workitem*, *w*s' distance is less than the value stored in the *distance* map for the vertex in the *workitem* ($w[0]$, the first element of the *workitem* tuple). Constructor of the statement specifies how to construct the new *workitem* w_n. The processing function defines the core logic that needs to be achieved by any SSSP algorithm. Some of the algorithms discussed in table 5.1 extend this definition because of the way they order *workitems*.

5.2.1. Chaotic SSSP. The Chaotic SSSP algorithm does not order *workitems*. Therefore, the strict weak ordering relation is defined in such a way that no two *workitems* are related (defined in Definition 9).

DEFINITION 9. $<_{ch}$: $WorkItems^{sssp} \times WorkItems^{sssp}$ *is a binary relation where* $\forall w_1, w_2 \in WorkItems^{sssp}$: $w_1 \not<_{ch} w_2$.

This relation induces only one equivalence class, and all the *workitems* in this class can be executed in parallel. The AGM model for Chaotic SSSP algorithm is given in Proposition 2. The presented AGM uses the strict weak ordering defined in Definition 9.

PROPOSITION 2. *Chaotic Algorithm is an instance of an AGM where*

(1) $G = (V, E, vmaps = \{\}, emaps = \{weight\})$ *is the input graph,*

(2) $WorkItems = WorkItems^{sssp}$,

(3) $Q = \{distance\}$ *is the state (initially* $\forall v \in V, distance(v) = \infty$*),*

(4) $\pi = \pi^{sssp}$,

(5) *Strict weak ordering relation* $<_{wis} = <_{ch}$,

(6) $S = \{<v_s, 0>\}$ *where* $v_s \in V$ *and* v_s *is the source vertex.*

5.2.2. Dijkstra's SSSP. The Dijkstra's SSSP algorithm globally orders *workitems* by their associated distances (Definition 10).

DEFINITION 10. $<_{dj}$: *WorkItems*$^{sssp} \times$ *WorkItems*^{sssp} *is a relation where* $\forall w_1, w_2 \in$ *WorkItems*^{sssp} : $w_1 <_{dj} w_2$ *iff* $w_1[1] < w_2[1]$.

The AGM formulation for Dijkstra's SSSP is same as the AGM formulation in Proposition 2 except for the strict weak ordering. In $<_{dj}$, two *workitems* belong to the same equivalence class if they have the same distance. In general, the equivalence classes generated by $<_{dj}$ are small, hence the parallelism available in Dijkstra's SSSP algorithm is limited.

5.2.3. Δ-Stepping Algorithm. Δ-Stepping **[100]** SSSP algorithm arranges vertex-distance pairs into distance *buckets*) of size $\Delta \in \mathbb{N}$ and executes buckets in order. Within a bucket, vertex-distance pairs can be executed in any order. Processing a bucket may produce extra work for the same bucket or for successive buckets. The strict weak ordering relation for Δ-stepping algorithm is given in Definition 11. As for Dijkstra's algorithm, Δ-Stepping AGM is as in Proposition 2 with ordering replaced by $<_\Delta$.

DEFINITION 11. $<_\Delta$: *WorkItems*$^{sssp} \times$ *WorkItems*^{sssp} *is a relation where* $\forall w_1, w_2 \in$ *WorkItems*^{sssp} : $w_1 <_\Delta w_2$ *iff* $\lfloor w_1[1]/\Delta \rfloor < \lfloor w_2[1]/\Delta \rfloor$.

5.2.4. K-Level Asynchronous Algorithm. The KLA paradigm **[64]** processes vertices up to k topological levels asynchronously (k can be varied). Correspondingly, the K-Level Asynchronous AGM orders *workitems* by their *level*. To do this, *workitems* include an additional *ordering attribute*. The K-Level Asynchronous *WorkItems* is defined as *WorkItems*$^{kla} \subseteq V \times$ *Distance* $\times$ *Level* where *Level* $\subseteq \mathbb{N}$. The processing function also is extended to populate the level attribute (Definition 12).

DEFINITION 12. $\pi^{kla} : WorkItems^{kla} \longrightarrow 2^{WorkItems^{kla}}$

$$\pi^{kla}(w) = \begin{cases} \{w_n | \langle w_n[0] \in neighbors(w[0]) \\ \quad \text{and } w_n[1] = w[1] + weight(w[0], w_k[0]) \\ \quad \text{and } w_n[2] = w[2] + 1\rangle, \\ \langle distance(w[0]) \longleftarrow w[1]\rangle, \\ \langle if\ w[1] < distance(w[0])\rangle\} \end{cases}$$

The *workitems* within consecutive k levels can be executed in parallel. The strict weak ordering relation for K-Level Asynchronous is given in Definition 13. The AGM for K-Level Asynchronous algorithm is as AGM in Proposition 2 except for the processing function, which is replaced with π^{kla}, and for the strict weak ordering, which is replaced with $<_{skla}$ defined in Definition 13.

DEFINITION 13. $<_{skla}$: $WorkItems^{kla} \times WorkItems^{kla}$ *is a relation where* $\forall w_1, w_2 \in WorkItems^{kla}$: $w_1 <_{skla} w_2$ *iff* $\lfloor w_1[2]/K \rfloor < \lfloor w_2[2]/K \rfloor$.

PROPOSITION 3. *K-Level Asynchronous Algorithm is an instance of AGM where:*

(1) $G = (V, E, vmaps = \{\}, emaps = \{weight\})$ *is the input graph,*

(2) $WorkItems = WorkItems^t$,

(3) $Q = \{distance\}$ *is the state mapping and initially* $\forall i \in V$, $distance(i) = \infty$,

(4) $\pi = \pi^{kla}$,

(5) *Strict weak ordering relation* $<_{wis} = <_{skla}$,

(6) $S = \{<v_s, 0, 0>\}$ *where* $v_s \in V$ *and* v_s *is the source vertex and level starts with 0.*

5.2.5. Family of SSSP Algorithms. The SSSP AGMs are summarized in fig. 5.1. Dijkstra's, Δ-Stepping , and Chaotic algorithms share the same processing function but with different orderings. Both Dijkstra's algorithm and Δ-Stepping algorithm use distance to define their strict weak orderings. K-Level Asynchronous uses levels to order *workitems*. The only difference between π^{sssp} and π^{kla} is that π^{kla} has logic to update level attribute in

TABLE 5.2. Thread ordered, Numa ordered and Process ordered EAGMs for Δ-stepping, KLA and Chaotic AGMs.

	buffer	threadq	numaq	nodeq
Δ-Stepping	$<_{\Delta(5)}$ ↓ $<_{ch}$ ↓ $<_{ch}$ ↓ $<_{ch}$	$<_{\Delta(5)}$ ↓ $<_{ch}$ ↓ $<_{ch}$ ↓ $<_{dj}$	$<_{\Delta(5)}$ ↓ $<_{ch}$ ↓ $<_{dj}$ ↓ $<_{ch}$	$<_{\Delta(5)}$ ↓ $<_{dj}$ ↓ $<_{ch}$ ↓ $<_{ch}$
K-Level Asynchronous	$<_{kla}$ ↓ $<_{ch}$ ↓ $<_{ch}$ ↓ $<_{ch}$	$<_{kla}$ ↓ $<_{ch}$ ↓ $<_{ch}$ ↓ $<_{dj}$	$<_{kla}$ ↓ $<_{ch}$ ↓ $<_{dj}$ ↓ $<_{ch}$	$<_{kla}$ ↓ $<_{dj}$ ↓ $<_{ch}$ ↓ $<_{ch}$
Chaotic	$<_{ch}$ ↓ $<_{ch}$ ↓ $<_{ch}$ ↓ $<_{ch}$	$<_{ch}$ ↓ $<_{ch}$ ↓ $<_{ch}$ ↓ $<_{dj}$	$<_{ch}$ ↓ $<_{ch}$ ↓ $<_{dj}$ ↓ $<_{ch}$	$<_{ch}$ ↓ $<_{dj}$ ↓ $<_{ch}$ ↓ $<_{ch}$

newly generated *workitems*. In fig. 5.1, we represent this with a dashed arrow to indicate that π^{kla} is an *extended* version of π^{sssp}. Because all the algorithms use the same processing function (with ordering extension for K-Level Asynchronous), they form an *algorithm family*.

5.3. SSSP EAGMs

AGMs are abstract and independent of implementation details. However, distributed graph algorithms are strongly impacted by properties of the distributed architecture they run on. To capture that impact, we introduce *extended* AGM (EAGM) that represents *spacial distribution* on a distributed memory platform. Currently, we recognize 4 hierarchical levels of

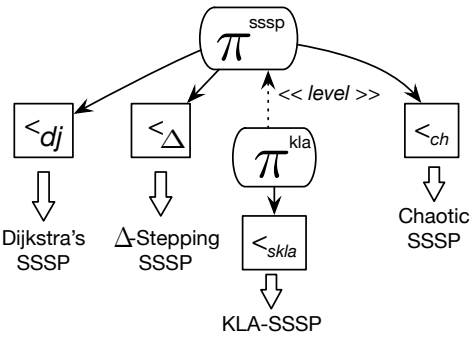


FIGURE 5.1. Summary of AGMs for SSSP algorithms.

distribution that roughly match modern distributed systems (arrows indicate inclusion):

Global $\longrightarrow$ **Process** $\longrightarrow$ **Numa** $\longrightarrow$ **Thread**

Given the spatial hierarchy, we use EAGMs to specify *spatial orderings* for AGM graph algorithms. Spatial orderings apply non-semantic ordering on *workitems* throughout the spatial hierarchy of a distributed machine. The ordering at the **Global** level is the same as in the underlying AGM, keeping the semantics of an AGM intact. Since the global ordering maintains the equivalence classes of AGM, *workitems* can be further ordered at the lower levels of the hierarchy. For example, two different EAGM spatial orderings for Δ-Stepping are $<_{\Delta(5)} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{ch}$ and $<_{\Delta(5)} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{dj}$ where each ordering corresponds to the EAGM level (the orderings are as defined in the previous section). The first spatial ordering enforces $<_{\Delta(5)}$at the global level, but leaves execution in buckets unordered ($<_{ch}$). The second spatial ordering applies Dijkstra's ordering at the **Thread** level ($<_{dj}$), which means that *workitems* at every thread are ordered in a priority queue as they reach the thread in the spatial distribution. In summary, an EAGM consists of an AGM and a spatial architecture hierarchy annotated by spatial orderings.

In table 5.2, we apply Dijkstra's strict weak ordering relation (Definition 10) to spatial hierarchy levels of Process (`nodeq`), Numa (`numaq`), and of Thread (`threadq`) to derive EAGMs for algorithms in table 5.1. The *buffer* represents the original algorithm without spatial level orderings. The table shows orderings for each combination of ordering and AGM, where the ordering chain corresponds to the archtectural hierarchy given at the beginning of this section. Each EAGM generates a variation of the main algorithm defined by its corresponding AGM. By methodical application of spatial ordering, we derive a family of SSSP algorithms. In the next section, we evaluate the performance of different EAGMs.

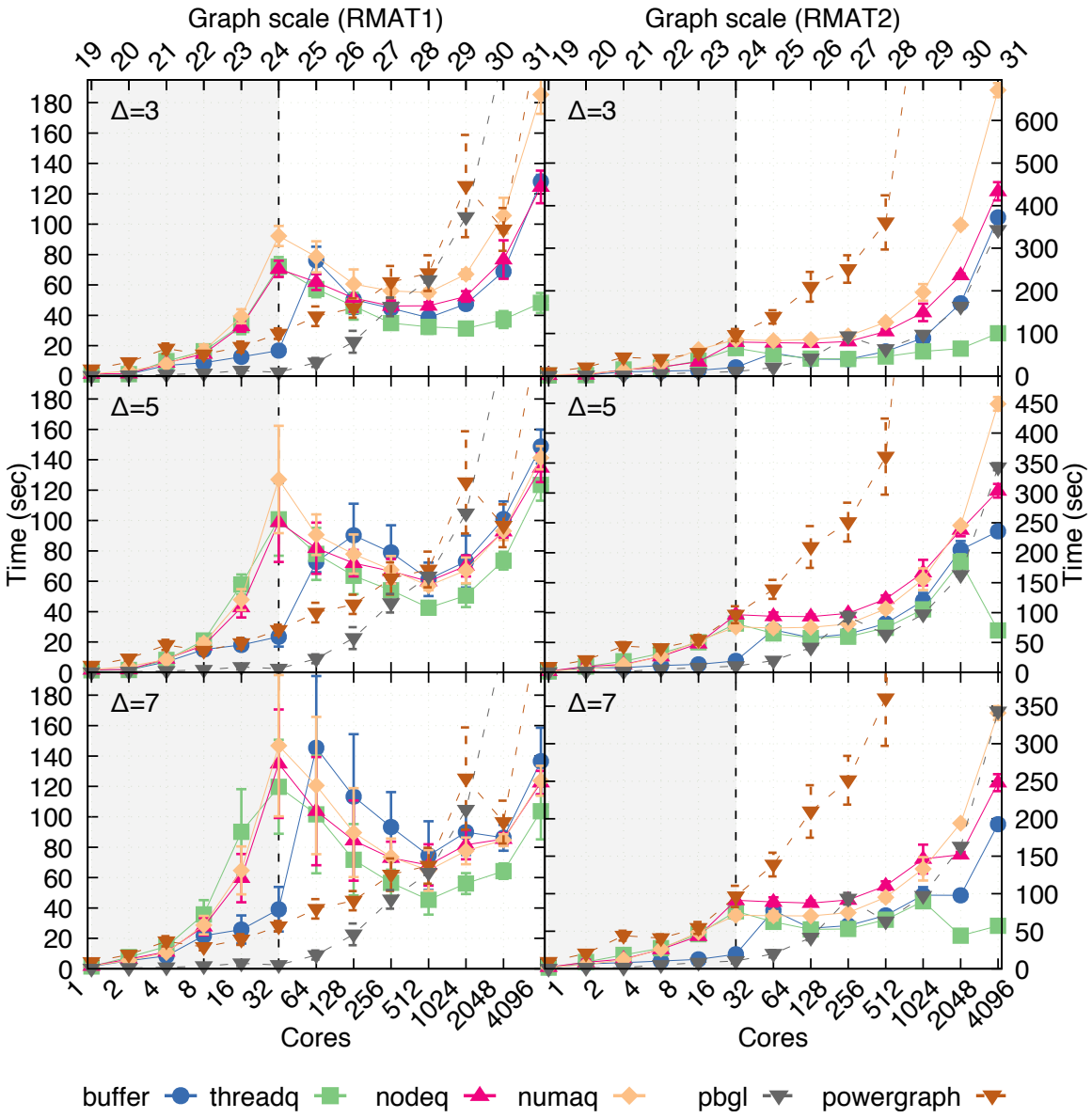


FIGURE 5.2. Timing results of Δ-stepping. Shaded region indicates single node runs.

5.4. Experiments & Results

In this section, we implement and compare the weak scaling performance of each derived EAGM in table 5.2. In addition, we also compare the performance of the EAGMs to the performance of SSSP algorithms available in two well-known graph processing frameworks PowerGraph [**55**] and Parallel BGL [**40**].

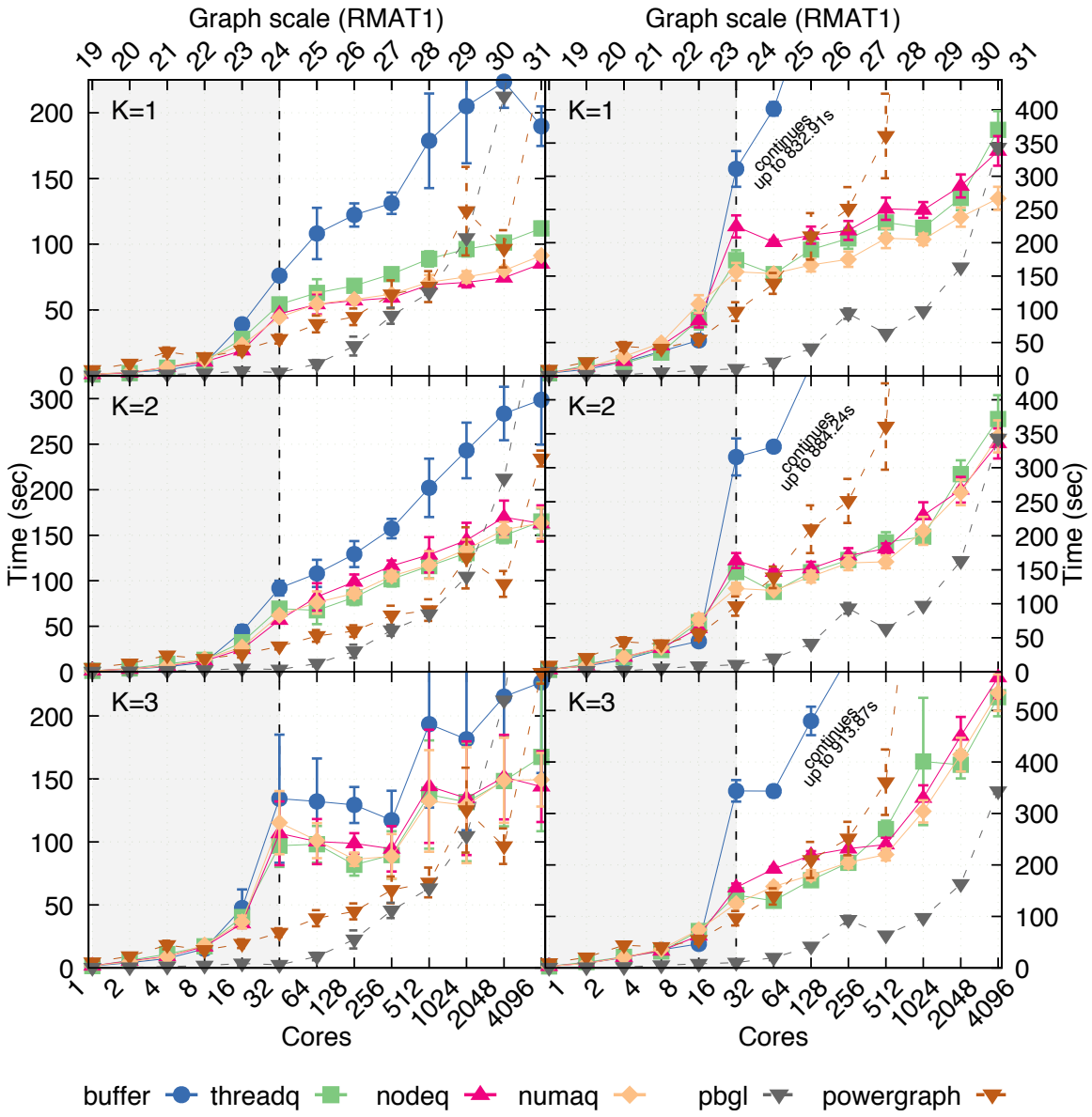


FIGURE 5.3. Timing results of KLA. Shaded region indicates single node runs. Weak scaling performance is measured on two types of synthetic *R-MAT* **[23]** graphs: RMAT1 graphs with R-MAT parameters $A = 0.57$, $B = C = 0.19, D = 0.05$ and with edge weights ranging 0-100, and RMAT2 graphs with R-MAT parameters $A = 0.5$, $B = C = 0.1, D = 0.3$ and with edge weights 0-255. All experiments were carried out on Cray XE6/XK7

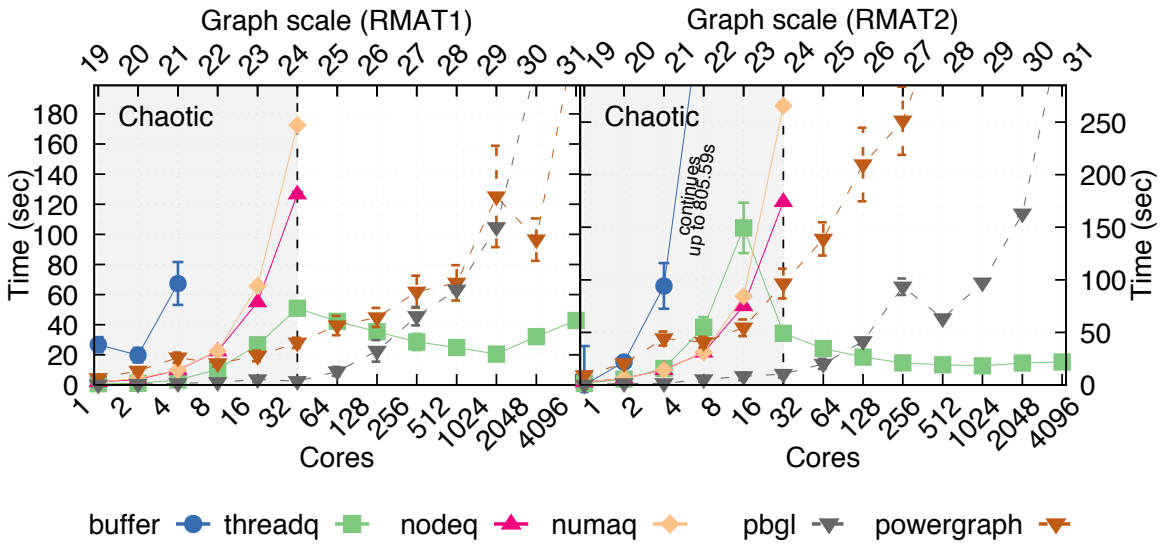


FIGURE 5.4. Timing results of the Chaotic EAGM. Shaded region indicates 1-node runs.

nodes, each with 2 AMD Opteron Abu Dhabi CPUs (for total of 32 cores), and 64 GB of memory per node (4 numa domains, 2 per CPU).

The algorithms are implemented in AM++ **[136]**, a light-weight active messaging framework. Graph vertices are equally distributed among distributed processes and in-node graph structure is stored in *compressed sparse row* format. Disjktra's orderings is implemented using concurrent priority queues at the process and the numa levels, and using standard priority queue at the thread level.

5.4.1. Scaling Results. The weak scaling results are presented in figs. 5.2 to 5.4. Experiment results for basic AGMs are represented using the `buffer` designator. As in table 5.2, EAGMs with thread-level, node-level and numa-level Dijkstra orderings are represented using `threadq`, `nodeq` and, `numaq` designators. We tested the performance of Δ-Stepping EAGMs for three delta values ($\Delta = 3, 5, 7$) and K-Level Asynchronous EAGMs with three k values ($k = 1, 2, 3$). In the following, we discuss results in detail.

5.4.1.1. *Δ-Stepping Variations.* The basic Δ-stepping (`buffer`) algorithm performs the best in-node (up to 32 cores). Since no communication is involved, the additional ordering

provided by the other implementations does not provide a sufficient benefit for its overhead. In general, the `threadq` variation is the fastest in the distributed setting for both RMAT1 and RMAT2 graph inputs. The `nodeq` and the `numaq` variations perform better with increasing deltas, but they are not competitive with the `buffer` implementation.

For RMAT1 graph inputs, PoweGraph shows better distributed performance for small scale graphs. However, for larger graph inputs, PowerGraph does not scale well. All the Δ-Stepping EAGMs outperform PowerGraph at higher scales, especially for RMAT2. The `threadq` EAGM shows better performance than PBGL on RMAT2 graphs, and for RMAT1 graphs, all EAGMs outperform PBGL.

In summary, while in-node performance is dominated by the basic Δ-Stepping algorithm (excluding PowerGraph and PBGL results), the distributed execution shows significant improvement with the `threadq` EAGM. Although the `numaq` and `nodeq` variations provide more ordering than the `threadq` variation, the overhead of the concurrent ordering reduces the performance of `numaq` and `nodeq`.

5.4.1.2. *KLA Variations.* KLA variations show different performance characteristics than Δ-stepping. For KLA, the `nodeq` and the `numaq` variations perform the best at scale, with $K = 1$. At greater K values, the performance of `threadq` is comparable to `nodeq` and `numaq`, but, in absolute terms, the performance at higher K values is worse than at $K = 1$. The `numaq` and `nodeq` provide the best potential ordering by ordering the most items. The overheads are kept at bay because at $K = 1$ all the writes to the next level's queue occur before all the reads. For higher K values, writes and reads get more mixed, and the advantage of `numaq` and `nodeq` becomes less pronounced. In KLA, for both RMAT1 and RMAT2 inputs, all EAGM variations (`threadq`, `nodeq` and `numaq`) perform better compared to the basic `buffer` variation.

For RMAT1 graph inputs, PowerGraph outperforms almost all the KLA EAGMs. However, PowerGraph execution time tends to increase with the scale, but K-Level Asynchronous EAGM variations tend to scale well with the increasing scale. For RMAT2 graph inputs, all the K-Level Asynchronous EAGM variations, except for `buffer`, outperform PowerGraph in distributed execution. However, for RMAT2, PBGL outperforms almost

all EAGMs, and `numaq` and `nodeq` tend to perform better at higher scales with $K = 1$. All the EAGMs show better performance than PBGL for RMAT1 graphs.

5.4.1.3. *Chaotic Variations.* For chaotic EAGMs, the thread-level ordering shows good performance, specially in distributed execution. For RMAT2, `threadq` weak scales almost perfectly in distributed execution. In addition, the `threadq` variation outperforms GraphLab and PBGL for both RMAT1 and RMAT2 graphs in distributed execution. Furthermore, the `threadq` Chaotic EAGM is faster than all other EAGMs in terms of absolute performance, demonstrating how the structured (E)AGM approach may result in new, highly performant algorithms.

5.5. Summary

Using the AGM abstraction, we showed that existing distributed graph algorithms; Dijkstra's SSSP, Δ-Stepping SSSP and K-Level Asynchronous has the same processing logic but with different orderings. These orderings generate different equivalence class either based on distance or based on the level. We also showed, proposed EAGM model generates more fine-grained orderings at less synchronized spatial levels. Results of our experiments showed that some of the generated algorithms perform better compared to standard distributed memory, parallel SSSP algorithms under different graph inputs.

6

Priority Based Connected Components

Connected Component computation is an important graph problem used in many applications. In this chapter, we present a priority-based asynchronous distributed memory parallel graph algorithm for computing connected components in a graph. The proposed algorithm avoids synchronization and uses priority to reduce the amount of work. Experimental results show that the proposed algorithm performs better compared to traditional Connected Components (CC) algorithms, such as Shiloach-Vishkin (SV).

6.1. The Problem

For an undirected graph $G = (V, E)$, where V and E represent vertex and edge set, respectively, we say $v, u \in V$ are in the same **connected component** if a path exists from u to v.

Sequential algorithms to solve CC problem use either BFS or DFS. While DFS algorithms are not directly parallelizable [**117**], BFS algorithms can be parallelized. However, in the worst case, BFS algorithms can perform up to $O(|V|^2)$ of work. Although there are more sophisticated parallel algorithms proposed for PRAM architecture, in practical distributed computing settings, the performance of these algorithms suffer from the overheads of synchronization and remote message communication. Further, their performance varies depending on the structure of the input graph.

6.2. The Asynchronous Algorithm

The proposed algorithm works based on a DAG. To construct the DAG, we assume that each vertex in the graph is associated with a priority. The priority assigned to a vertex is unique. We represent the priority vector with α. For $v_1, v_2 \in V$ *and* $v_1 \neq v_2$, $\alpha[v_1] < \alpha[v_2]$ *or* $\alpha[v_2] < \alpha[v_1]$. The lower the value of $\alpha[v](v \in V)$, the higher the priority. The priority vector can be easily constructed by assigning the global vertex id to vector values. However, in order to avoid any load imbalance issues in distributed execution, the vertex ids must be randomly distributed among ranks (assuming graphs is 1D distributed). Further, we assume that the algorithm maintains a state called *vcomponent*. This state has an entry per each vertex. The component state stores the priority value of the highest priority vertex reachable from the current vertex. Initially, the component state of a vertex is set to its vertex priority.

The algorithm starts with the sources of the DAG. A source is a vertex, where priority is higher than its neighbors (e.g., see vertex 1 and 12 in Figure 6.1). Source vertices send their priority values to all their neighbors (Figure 6.2). Every time a vertex receives a message from a neighbor, it checks whether the priority value of its neighbor is higher than the current component priority. If it is higher, then the current vertex changes its component value to the newly received priority value. Whenever, a vertex component value is updated, the vertex checks whether the updated component priority is greater than all of its neighbors. If so, the vertex will notify all of its neighbors of the updated priority value (Figure 6.3). However, if the vertex has a neighbor that has a higher priority than the newly updated priority, it will not send (See vertex 14 in Figure 6.3). The main objective of this algorithm is to propagate the search for the highest priority vertex in a component. When there are no more changes in the component state, the algorithm terminates (Figure 6.4).

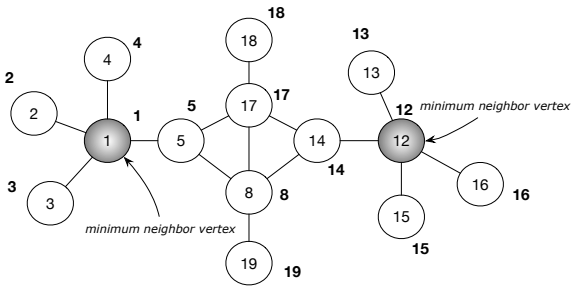


FIGURE 6.1. Initial step of the algorithm. Numbers depict the value of component state for each vertex.

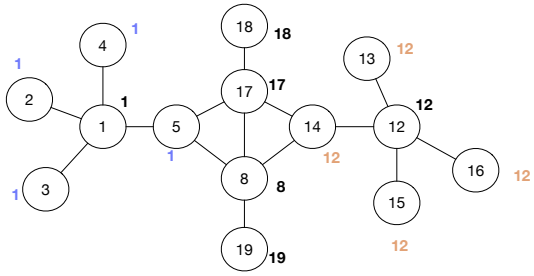


FIGURE 6.2. Algorithm step 2.

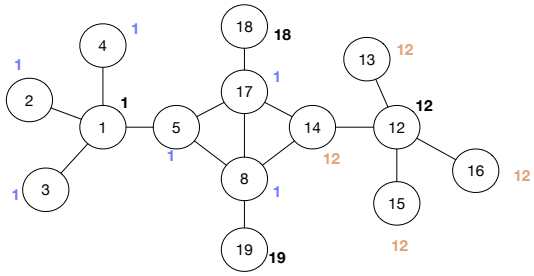


FIGURE 6.3. Algorithm step 3.

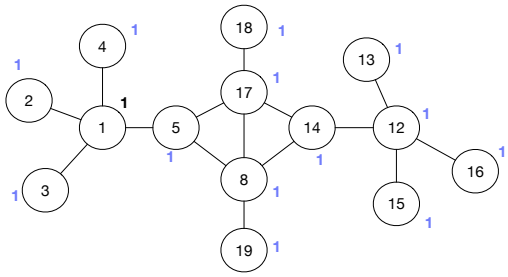


FIGURE 6.4. At the termination of the algorithm.

Algorithm 3 Distributed-Memory Parallel CC Algorithm

```
1:  procedure CC(G^local, Set:sources)
2:    for each Vertex v in sources do
3:      Send(v, vcomponent[v])
4:    end for
5:  end procedure
6:
7:  procedure RECEIVE(Vertex:v,Priority:p)
8:    if (p < vcomponent[v]) then
9:      vcomponent[v]← p
10:     setadj ← adjacencies(v, G^local)
11:     if min(setadj) > p then
12:       for each u in setadj do
13:         Send(u, vcomponent[v])
14:       end for
15:     end if
16:   end if
17: end procedure
```

The CC algorithm in Algorithm 3 assumes $\alpha[v] = v, \forall v \in V$. The main entry procedure is "CC" (Line 1 –Line 5). The algorithm assumes vertices are uniformly distributed among nodes. The G^{local} represents the sub-graph corresponding to the current *Rank*. "CC" procedure also takes the source vertex set. The function called *Send* (Line 3,Line 13) and the procedure *Receive* are related to each other. Whenever *Send* is called, it will invoke the *Receive* (Line 7–Line 17) procedure (maybe in a remote locality). The *Receive* procedure then implements the execution logic explained in the previous paragraph.

If there is more than one source vertex, there will be multiple parallel searches in the same component. In that case, the search with the higher priority source vertex will take over the search with the lower priority source vertex.

It is important to note that the proposed algorithm starts with only a few vertices (source vertices) and gradually increases the amount of work. Then, when the most number of vertices reach their saturated state (i.e., when a vertex is reached by the highest priority vertex it can reach), the amount of work gradually declines. Finally, when all vertex states are finalized, algorithm terminates.

The proposed algorithm is a *label-correcting* algorithm. i.e., the component state of a vertex is repeatedly changed until its value is equal to the source vertex with the highest priority. Also, note that the proposed algorithm is *un-ordered*: i.e. for the correctness of

the algorithm, the order states being changed do not have an effect. However, the order states being changed have an impact on the performance of the algorithm. The sooner the highest priority source vertex can update vertices in a component, the sooner the algorithm converges.

6.3. Ordering

The proposed algorithm is vertex-centric. Every time a component state associated with a vertex is changed, a message is sent to its neighbors, including the neighbor vertex and the updated component priority. We call such a message a *workitem*.

The performance of the algorithm depends on how we order *workitems*. Ordering is added in such a way that algorithm minimizes synchronization. We use "data parallel priority queues" to order *workitems*. "Data parallel priority queues" maintain a priority queue per each parallel thread. When a thread receives a *workitem*, it is added to the appropriate priority queue.

Priority-based distributed memory parallel CC algorithm is listed in Algorithm 4. The algorithm consists of four procedures. The *Initialization* procedure (Line 1–Line 5) is called at the start of the algorithm. During initializing, the component state (*vcomponent*) of every vertex is initialized to its priority (SeeLine 3). The rest of the procedures (*Run*, *HandleQueue*, and *Receive*) are called in each parallel threads.

Once a thread is spawned, the algorithm invokes *Run* procedure with the thread id. The *Run* function calculates the source vertex set for DAG execution (Line 7–Line 19). Found source vertices are pushed into the priority queue relevant to the current thread. Finally, *Run* procedure calls *HandleQueue* procedure (Line 20).

The *HandleQueue* pops *workitems* from the priority queue relevant to the current thread and executes logic similar to logic inside the *Receive* function in Algorithm 3. As in Algorithm 3, every *Send* invokes a *Receive* procedure in a remote rank or in the same rank. The *Receive* function is quite similar to the *Receive* function explained in Algorithm 3 except that *Receive* function in Algorithm 4 inserts *workitem* to the priority queue (relevant to invoking thread) if *workitem* updated the component state.

Algorithm 4 Priority based CC Algorithm

```
1: procedure INITIALIZATION(G^local = (V, E))
2:   for v ∈ V : do in parallel
3:     vcomponent ← v
4:   end for
5: end procedure
6: procedure RUN(thread:tid)
7:   for v ∈ V do in parallel
8:     v_min ← v
9:     for u ∈ neighbors(v, G^local) do
10:      if u < v_min then
11:        v_min ← u
12:        break
13:      end if
14:    end for
15:    if u == v_min then
16:      workitem w(u, u)
17:      push(pq[tid], w)
18:    end if
19:  end for
20:  HandleQueue(tid)
21: end procedure
22: procedure HANDLEQUEUE(thread:tid)
23:  while true do
24:    while !empty(pq[tid]) do
25:      w ← pop(pq[tid])
26:      v ← w.destination
27:      p ← w.priority
28:      setadj ← adjacencies(v, G^local)
29:      if min(setadj) > p then
30:        for u ∈ setadj do
31:          workitem w(u, p)
32:          Send(w)
33:        end for
34:      end if
35:    end while
36:    if terminate() then
37:      break
38:    end if
39:  end while
40: end procedure
41: procedure RECEIVE(workitem:w, thread:tid)
42:  v ← w.destination
43:  p ← w.priority
44:  if p ¡ vcomponent(v) then
45:    vcomponent(v) ← p
46:    push(pq[tid],w)
47:  end if
48: end procedure
```

6.4. Experiments & Results

We evaluate weak scaling performance of the proposed CC algorithm (*pr-cc*).

6.4.1. Implementation. We implemented the proposed algorithm on top of a MPI wrapped lightweight messaging framework called AM++ [**136**]. The graph vertices are equally distributed among participating nodes (also called *1D block distribution*). The local graph is represented using *compressed sparse row* format. In the local graph, each undirected edge is represented using two directed edges. The priorities are encoded into the vertex labels (so we do not need separate space for vertex priorities).

6.4.2. Experiment Setup. We ran our experiments on a Cray XC system that has 2 Broadwell 22-core Intel Xeon processors. Each node consists of 128 GB DDR4-2400 memory. The MPI implementation we used is Cray MPICH (version 7.4.4).

Preliminary results showed that to get the best performance results for all the algorithms, we needed to run two processes per node (Because there are two sockets per node) in MPI "thread multiple" mode.

6.4.3. Graph Input. For weak scaling experiments, we use *R-MAT* [**23**] synthetic graphs. Two types of RMAT synthetic graphs were used. They are:

- RMAT-1: Graphs based on the current Graph500 [**104**] Breadth First Search benchmark specification with R-MAT parameters $A = 0.57$, $B = C = 0.19$ and $D = 0.05$.
- RMAT-2: Graphs generated based on the proposed Graph500 [**56**] SSSP benchmark specification with R-MAT parameters $A = 0.50$, $B = C = 0.1$ and $D = 0.3$.

6.4.4. Baseline Algorithms. We compare the performance of the proposed algorithm with the Shiloach-Vishkin [**123**] algorithm. There are several distributed versions of Shiloach-Vishkin with optimizations (See [**38**]). In addition to the main Shiloach-Vishkin algorithm, we also compared the performance of our algorithm with an optimized SV algorithm. In summary, we compared with following algorithms:

(1) Shiloach-Vishkin algorithm (*sv*) – This is the original Shiloach-Vishkin algorithm

(2) Optimized Shiloach-Vishkin (*svopt*) – To reduce remote message communication this algorithm first calculates connected components in local graphs and then execute SV operations (tree hooking and short-cutting).

6.4.5. Weak Scaling Results. Weak scaling results for RMAT-1 graphs and RMAT-2 graphs are presented in Figure 6.5 and Figure 6.6.

In both cases (RMAT-1 & RMAT-2) *sv* and *svopt* algorithms perform better in shared memory execution (when the number of cores is less than 16). However, in distributed execution, the priority CC performed better. Also, the plots show that priority CC scaled well compared to the other two algorithms.

SV algorithms exchanged nearly four times the messages than the proposed algorithm. As an example, at scale 28 (16 * 32 cores), SV algorithm exchanged about 35668649973 messages and our algorithm exchanged about 8583349641 messages. Also, to process a scale 28 graph, the SV algorithm executed 18 super steps (more than 18 barriers). Due to these reasons, we argue that priority based algorithm is efficient.

Further, we see that the performance of SV algorithms with RMAT-1 input graphs is better than RMAT-2 graphs.

6.5. Connected Components in AGM

The AGM model for the proposed CC algorithm is presented in Proposition 4. For this algorithm, the *workitem* definition is similar to the *workitem* definition for SSSP. We define $WorkItemSet^{cc} \subseteq V \times \mathbb{N}$. The algorithm uses the *component* state to store highest priority value within a single component. The processing function (See Definition 14) has one statement and this statement first checks whether incoming *workitem* priority is higher than the priority stored inside *component* state. If the priority is higher, the *component* state is updated with the incoming *workitem* priority and the change is relayed to lower priority neighbors. As discussed in the previous subsection, there are multiple ways to order *workitems* for this algorithm. The "strict weak ordering relation" that does not perform ordering is presented in Definition 15. Other orderings, such as by level or by a Δ bucket

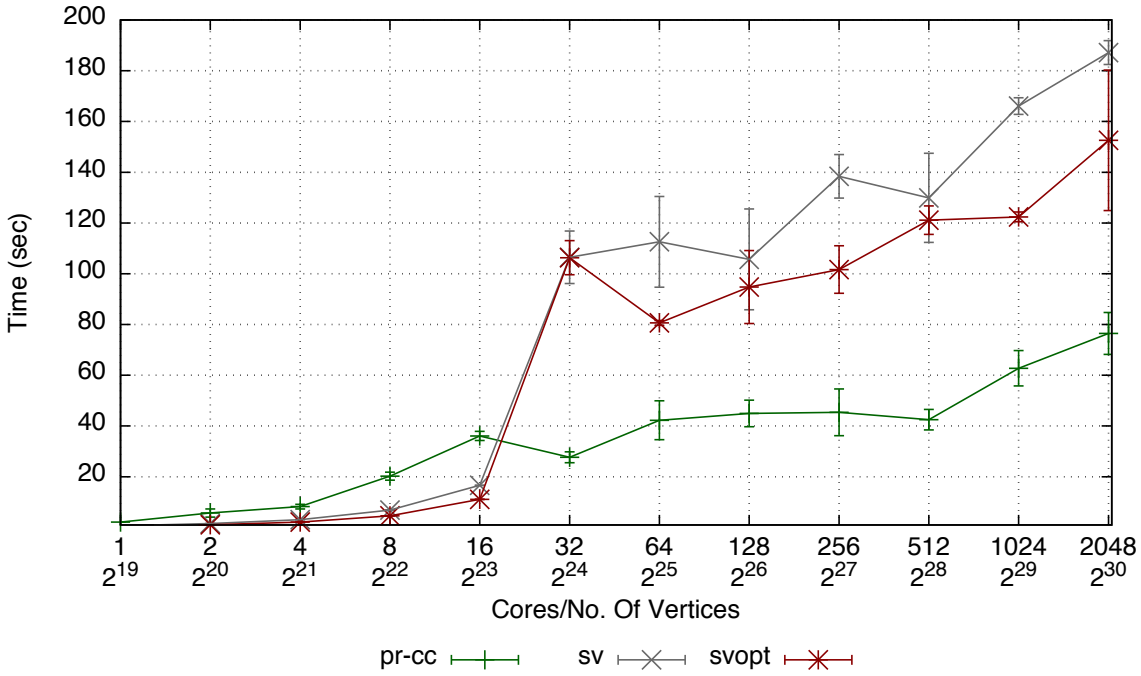


FIGURE 6.5. CC Algorithms execution time for RMAT-1 graphs.

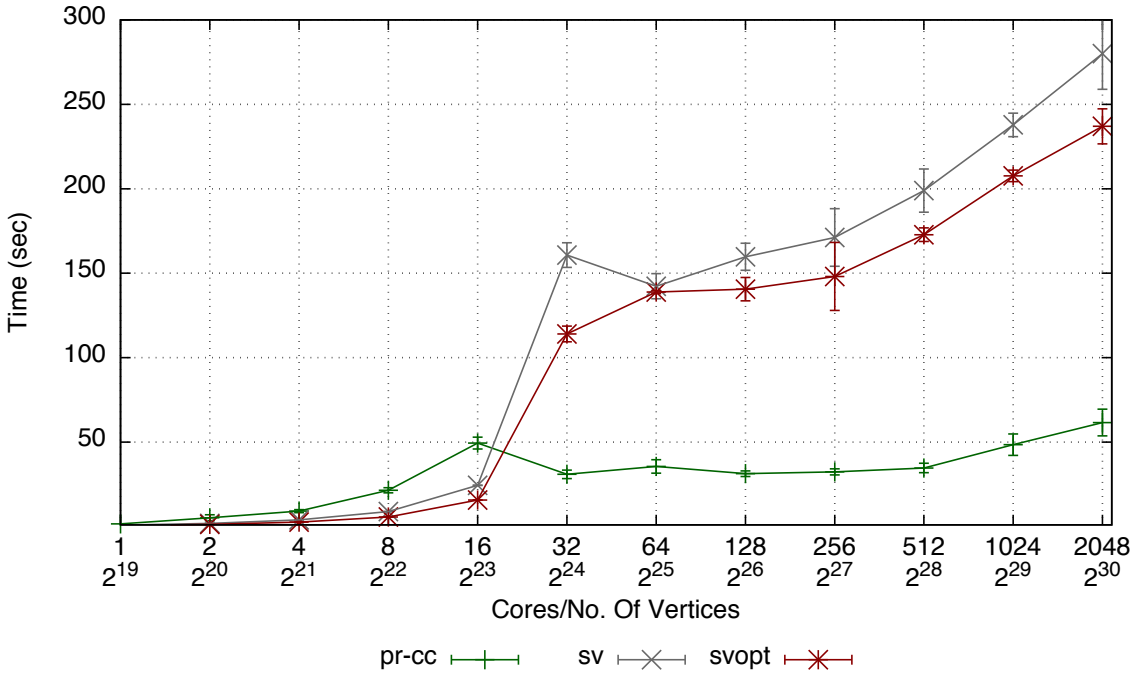


FIGURE 6.6. CC Algorithms execution time for RMAT-2 graphs.

defined on priority, are also possible. Each of these orderings instantiates a different AGM. The AGM presented in Proposition 4 uses single equivalence class, a chaotic ordering.

DEFINITION 14. $\pi^{cc} : WorkItems^{cc} \longrightarrow 2^{WorkItems^{cc}}$

$$\pi^{cc}(w) = \begin{cases} \{w_k | w_k \in < w_n[0] \in neighbors(w[0]) \; and \\ w[0] < w_n[0] \; and \\ w_n[1] \longleftarrow w[1] > \\ < component(w[0]) \longleftarrow w[1] >, \\ < if \; w[1] < component(w[0]) > \} \end{cases}$$

DEFINITION 15. *$<_{cc}$ is a binary relation defined on $WorkItems^{cc}$ where, $w_1 \not<_{cc} w_2$ nor $w_2 \not<_{cc} w_1, \forall w_1, w_2 \in WorkItems^{cc}$.*

PROPOSITION 4. *CC Chaotic Algorithm is an instance of AGM where:*

(1) $G = (V, E, vmaps = \{\rho\}, emaps = \{\})$ *is the input graph,*

(2) $WorkItems = WorkItems^{cc}$,

(3) $Q = \{component\}$ *is the state mapping and initially* $\forall i \in V, component(i) = \rho(i)$,

(4) $\pi = \pi^{cc}$,

(5) *Strict weak ordering relation* $<_{wis} = <_{cc}$,

(6) $S = \{<v, \rho(v)>\}$ *where* $v \in V$ *and* $\forall i \in neighbors(v)$, $\rho(v) > \rho(i)$.

6.6. Summary

This chapter presented a priority-based connected components algorithm. The algorithm without priority is quite simple, and priority and thread level ordering helps to reduce the amount of work while avoiding synchronization. The algorithm we presented is asynchronous, label-correcting and un-ordered. The proposed algorithm performs better in distributed execution as it minimizes synchronization and reduces remote messages.

7

Luby's Maximal Independent Set

The Maximal Independent Set (MIS) graph problem arises in many applications such as computer vision, information theory, molecular biology, and process scheduling. The growing scale of MIS problems suggests the use of distributed-memory hardware as a cost-effective approach to providing necessary compute and memory resources. Luby proposed four randomized algorithms to solve the MIS problem. All those algorithms are designed focusing on shared-memory machines and are analyzed using the PRAM model. These algorithms do not have direct efficient distributed-memory implementations. In this chapter, we extend two of Luby's seminal MIS algorithms, "Luby(A)" and "Luby(B)," to distributed-memory execution, and we evaluate their performance. We compare our results with the "Filtered MIS" implementation in the Combinatorial BLAS library for two types of synthetic graph inputs.

7.1. Introduction

Let $G = (V, E)$ be a graph where V represents the set of vertices and E represents the set of edges in the graph. An *independent set* in G is a set of vertices in a graph such that no two vertices in the set are adjacent. The largest independent sets (there may be more than one) are called the *maximum independent sets*. Since finding a maximum independent set is NP-hard, most applications settle for finding a *maximal* independent set. A MIS of a graph is an independent set that is not a subset of any other independent set (see Figure 7.1). Finding a MIS is an important graph problem used in many applications, including computer vision, coding theory, molecular biology and process scheduling. Although

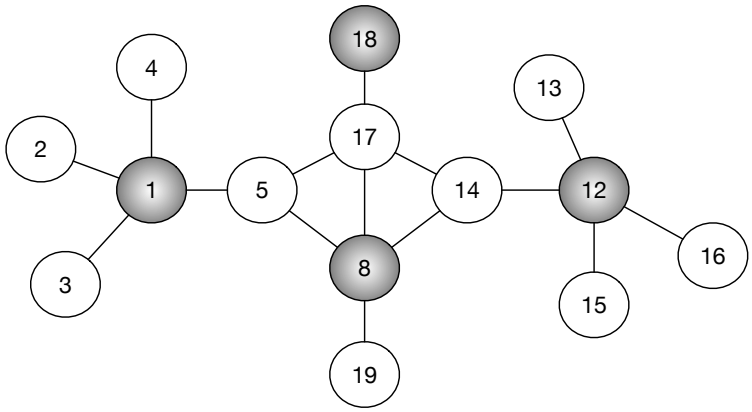


FIGURE 7.1. The gray nodes show a maximal independent set of this graph.

efficient MIS algorithms are well-known [**30**], the increasing scale of data-intensive applications suggests the use of distributed-memory hardware (clusters), which in turn requires distributed-memory algorithms.

Luby's Monte Carlo [**91**] MIS algorithms are often used for parallel MIS implementations. Luby MIS algorithms are designed focusing on shared memory machines and analyzed using the PRAM model. Luby's algorithms do not immediately lend itself to efficient distributed memory parallel algorithms due to overhead incurred by synchronization and distributed subgraph computations. In this chapter, we present distributed versions of Luby's Monte Carlo algorithms (Algorithm A and Algorithm B) that minimize these overheads. Furthermore, we derive a variation of Luby(A) that avoids computing random numbers in every iteration. All presented algorithms are implemented in the AM++ runtime [**136**] and their performance is evaluated. Our results show that the proposed algorithms scale well in distributed settings. We also compare our results with the *FilteredMIS* implementation in the Combinatorial BLAS library [**20**], and we show that our implementations are several times faster compared to FilteredMIS algorithm.

7.2. Luby's Algorithms

Luby's algorithms are the most widely used parallel algorithms for finding a MIS in shared memory. In his original publication, Luby discussed a *general iterative scheme* and four particular variations based on it. The general iterative scheme is listed inAlgorithm 5.

Algorithm 5 General Iterative Scheme in Luby MIS

Input: Graph $G = (V, E)$
Output: Maximal Independent Set S_{mis}

1: $S_{mis} \leftarrow \emptyset$
2: $S_{is} \leftarrow \emptyset$ ▷ initializing the independent set
3: $G_s(V_s \leftarrow V, E_s \leftarrow E)$
4: **while** $G_s \neq \emptyset$ **do**
5: $S_{is} \leftarrow$ Select an independent set from G_s
6: $S_{mis} \leftarrow S_{mis} \cup S_{is}$
7: $V_r \leftarrow S_{is} \cup \{$neighbors of vertices in $S_{is}\}$
8: $E_r \leftarrow \{$edges incident on vertices in$V_r\}$
9: $G_s \leftarrow G_s(V_s \leftarrow (V_s - V_r), E_s \leftarrow (E_s - E_r))$
10: **end while**

In every iteration, the general iterative scheme selects a non-empty independent set and merges it to the output (S_{mis}). Then, the selected independent set and its neighbors are removed from the input graph, and the resulting subgraph is fed into the scheme for the next iteration (Line 7–Line 9). This process is repeated until the resulting subgraph is empty. In every iteration, the general iterative scheme generates a new independent set. Luby proved that the union of all those independent sets is a maximal independent set.

To select an independent set from a subgraph in an iteration, Luby proposed two Monte Carlo algorithms: *Select A* and *Select B*. Select B is further enhanced to create two more variations, *Select C* and *Select D*. All four of those variations use randomization to calculate an independent set. Select algorithms A, B, and C are non-deterministic, while Select D is deterministic. In this chapter, we focus on Select algorithms A and B (since C and D are variations of B). Select A and Select B algorithms are summarized in Table 7.1.

Select A	**Select B**
1. Assume all the vertices in the subgraph are independent	1. Assume vertices that satisfy the *coin* random variable test are independent
2. Assign random values to vertices in V_s	2. Calculate the independent set based on the degree distribution of the subgraph
3. Calculate the independent set based on assigned random values	

TABLE 7.1. Independent set selection criteria for Select A and Select B algorithms

Select A is the simplest of the algorithms. It considers all the vertices (V_s) in the subgraph to be in an independent set. Then, it assigns a random number, r ($1 \leq r \leq |V_s|^4$) to

each vertex in the subgraph. Then, for every edge in the subgraph (edges in E_s), Select A removes the vertex in the edge that has the greater random value.

Unlike Select A, Select B does not consider all vertices in the subgraph to be in the independent set in an iteration. Instead, Select B uses a random variable (*coin*) to decide whether a vertex in the subgraph should be selected to be in an independent set. The value of the coin is determined based on a probability distribution defined using degree distribution of the subgraph. More precisely, if $d(v)$ is the degree associated with a vertex $v \in V'$, then $coin(v) = 1$ with probability $1/2d(v)$. If $d(v) = 0$, then $coin(v)$ is always 1. For more details about Algorithm B, we refer the reader to Luby's original publication in **[91]**.

7.2.1. Luby's Algorithms in Distributed Memory. The Luby algorithms do not lend themselves directly to efficient distributed-memory parallel implementations. Luby's algorithms are designed focusing on shared memory machines and are analyzed using the PRAM model. In the PRAM model, all processors need to synchronize after reading from the shared memory and also before writing to the shared memory. A natural way to extend a shared-memory Luby algorithm to distributed memory is to use the BSP approach. In BSP **[133]**, shared memory operations can be converted to compute, communication and barrier synchronization phases. However, this approach results in many barrier synchronization phases.

Another issue is that the "general iterative scheme" (algorithm 5) depends on subgraph computations. That is, in every iteration, the algorithm constructs a new subgraph the by removing vertices and edges of the independent set calculated in the current iteration from the graph. Constructing a subgraph in every iteration is inefficient in distributed memory as it involves communication and synchronization even if the subgraph is maintained implicitly through vertex masking.

In addition, the Select A algorithm requires a new choice of random numbers in every iteration. The range of the numbers depends on the number of vertices remaining in the subgraph. Therefore, the random number generation requires a reduction over the number

of vertices in the subgraph and a barrier in every iteration. Furthermore, random number generation incurs a significant computational overhead.

In the next section, we discuss how we extend Luby's algorithms to distributed execution while avoiding the drawbacks discussed above. In the proposed algorithm, the overhead of barrier synchronization phases are minimized by overlapping computation and communication. The subgraph computation is achieved through vertex filtering. However, vertex filtering cripples the ability to iterate over the graph data structure in parallel. Therefore, in our implementation, we use a parallel data structure. We also present a variation of Select A algorithm that avoids random number generation in each iteration and uses random numbers generated initially.

7.3. Distributed Memory Parallel Luby Algorithms

The proposed distributed-memory parallel Luby algorithms use a 1D distribution to distribute the graph vertices among participating ranks. Every rank gets a subset of vertices and an edge subset relevant to the vertices. Within a rank, a vertex subset and its associated edge subset is represented using a local graph representation ($G^{local} = (V^l, E^l)$). A vertex is "owned" by a rank and vertices owned by different ranks communicate by passing messages. Message passing communication between ranks is designed based on BSP processing, but with overlapped communication and computation for improved efficiency.

7.3.1. Distributed General Iterative Scheme. The distributed general iterative scheme (algorithm 6) requires subgraph computation. Though explicit subgraph computation may be practical in sequential and shared-memory parallel environments, it is inefficient in distributed memory due to overheads of creating and distributing a new subgraph at every iteration. Equivalent distributed subgraph computation functionality can be achieved with vertex filtering (i.e., apply a filtering predicate to indicate whether a vertex is to be considered in the current computation). Although with vertex filtering more edges then strictly necessary are traversed in every iteration, the increased parallel efficiency outweighs the cost of the unnecessary traversals.

Algorithm 6 Distributed General Iterative Scheme

```
1:  procedure LUBYITERATE(G^local, select^fn)
2:    buffer ← {}
3:    delete ← {}
4:    while there are NIL vertices in G do
5:      select^fn(&buffer, &delete)
6:      epoch {
7:        for each Vertex v in buffer in parallel do
8:          if v is not in delete then
9:            mis[v] ← IN
10:           for each u in adjacencies(v, G^local) do
11:             Send(u, OUT)
12:           end for
13:         end if
14:       end for
15:     }
16:   end while
17: end procedure
18:
19: procedure Receive(v : Vertex, s : State)
20:   mis[v] ← s
21: end procedure
```

To alleviate the overhead of subgraph computations in distributed memory, we use two data structures:

(1) An *append buffer* (*buffer*) – for efficient parallel access; and

(2) A *set structure* (*delete* set).

These two data structures are created by the general iterative scheme and are passed into a specific Select (A or B) algorithm. The Select algorithm is responsible for populating the append buffer and delete set. When the Select algorithm decides a vertex is a candidate to be in the MIS, it adds the vertex to the buffer. Next, after buffer contains the initial MIS candidate vertex set, all vertices that have a neighbor with a lesser random value assigned to it, are placed in the delete set. The general iterative scheme traverses the append buffer in parallel and checks whether a vertex is in the delete set; if the vertex is *not* present in the delete set, then the vertex is added to the result MIS. To reduce the contention when operating on the delete set we implement delete set as a collection of sets, where each thread maintains a set local to the thread. Insertion to a delete set is local to the

invoking thread. When querying an element from the delete set, we first check whether the element resides in the thread-local set, and then we search for the element in sets belonging to other threads. During the search phase the sets are only read, so they can be safely shared between threads. The two-step design (buffer and then delete set) limits contention between threads to the low-overhead insert contention on the append buffer.

During computation, a vertex can be in one of three states (stored in a property map)

(1) *IN*– vertex is in MIS;

(2) *OUT*– vertex is not in MIS; and

(3) *NIL*– vertex is not yet processed.

Initially, all the vertices are in state *NIL*. When the algorithm terminates, all vertex states are changed either to *IN* or *OUT*.

Whether a vertex state should be changed from *NIL* to *IN* or *NIL* to *OUT* is decided within the general iterative scheme (*LubyIterate* in Algorithm 6). First, the general iterative scheme invokes the appropriate "Select" algorithm $select^{fn}$ ($Select^{A}$ or $Select^{B}$). The specific select algorithm is responsible for populating the append buffer and the delete set (Line 5). The general iterative scheme iterates through the append buffer in parallel and checks whether a vertex is present in the delete set (Lines 7 to 14). If a vertex is *not* in the delete set then that vertex state is updated to *IN* (Line 9). When a vertex state is changed to *IN* state, all its neighbors' states are to changed to *OUT* state (11). The *Send* operation determines to which rank the message should be sent based on the destination vertex and the graph distribution. Messages sent through *Send* are received in the *Receive* (Lines 19 to 21) function. Traversing through vertices in the append buffer and updating vertex states takes place within a single super-step (i.e., within a single *epoch*).

Algorithm 7 Distributed Select^A

```
1: procedure SELECT^A(G^local, ref abuffer, ref deleteset)
2:   localcount ← 0
3:   globalcount ← 0
4:   for each Vertex v in G^local in parallel do
5:     if mis[v] == NIL then
6:       localcount ← (localcount + 1)
7:     end if
8:   end for
9:   globalcount ← Reduce(localcount,SUM)
10:  random ← RandomDist(1, globalcount^4, Seed())
11: /*Assign random values to vertices in NIL state*/
12:  for each Vertex v in G^local in parallel do
13:    if mis[v] == NIL then
14:      π[v] ← random.Value()
15:      abuffer.add(v)
16:    end if
17:  end for
18: /*Use random values to remove conflicting vertices*/
19:  epoch {
20:    for each Vertex i in abuffer in parallel do
21:      for each j in adjacencies(v, G^local) do
22:        if u belongs to G^local then
23:          if π[i] >= π[j] then
24:            deleteset.add(i)
25:          else
26:            deleteset.add(j)
27:          end if
28:        else
29:          Send(j, i, π[i], COMPARE)
30:        end if
31:      end for
32:    end for
33:  }
34: end procedure
35: /*Every Send call invokes a Receive*/
```

```
36: procedure Receive(j:Vertex, i:Vertex, irnd:Real, act:Action)
37:   if act == COMPARE then
38:     if irnd >= π[j] then
39:       Send(i, j, π[j], REMOVE)
40:     else
41:       deleteset.add(j)
42:     end if
43:   else
44:     if act == REMOVE then
45:       deleteset.add(j)
46:     end if
47:   end if
48: end procedure
```

7.3.2. Distributed Select A. *Select A* (Luby(A)) algorithm takes a local graph representation, an append buffer, and a delete set. Select A algorithm is listed in Algorithm 7. G^{local} is the local graph representation, *abuffer* represents the append buffer and *deleteset* represents the delete set. Select A algorithm first calculates the number of vertices in *NIL* state using a *global reduction* (Lines 4 to 9). Then, for each vertex in *NIL* state, a random value, k ($1 < k < globalcount^4$), is assigned (Lines 12 to 17). When generating random values, a combination of rank id and thread id is used to generate a unique random value seed for every thread ($Seed()$ on Line Cref10). Random values are stored in a property map; a rank only stores random values for vertices in its local graph. While assigning a random value to each local vertex in a *NIL* state, Select A algorithm inserts those vertices to the append buffer (Line 15).

In the next phase, the algorithm traverses through vertices in the append buffer in parallel and inserts adjacencies with higher random values that are in *NIL* state to the delete set (Lines 20 to 32). If the adjacent vertex does not belong to G^{local}, then a message is sent to the appropriate locality (Line 29), including the source vertex id i and its random value. The rank that owns the destination vertex j checks whether the received random value is less than the random value of j. If so, j is added to the delete set, otherwise a message is sent back to i to add i to the delete set. The first message is denoted using the action *COMPARE* and the second action is represented using the action *REMOVE*. Every

Send call corresponds to a *Receive* function (Lines 36 to 48) invocation. At the end of the execution of epoch in Lines 19 to 33, vertices in the append buffer but not in the delete set represent an independent set.

7.3.3. Select AV (A variation of Select A). The Select A algorithm generates random numbers in every iteration. Random number generation is computationally expensive, also, to generate random numbers we need to calculate the total subgraph vertex set size (across all distributed ranks) which incurs additional communication overhead due to distributed reduction and barrier synchronization.

Select AV (Luby(AV)) is almost same as Select A, except that, we do not generate random numbers to calculate an independent set. Instead, we use graph representation vertex IDs to break the symmetry and calculate an independent set. In other words, $\pi[i] = i \, \forall i \in V_s$, where V_s represents the vertex set of a subgraph. This method depends on the distribution of vertex identifiers in the graph data structure, but it works well with our representation which randomly permutes vertices before the algorithm begins.

Algorithm 8 Distributed Select^B

```
1: procedure Select^B(G^local, ref abuffer, ref deleteset)
2:   degree ← {} /*Calculate vertex degrees relative to the subgraph*/
3:   epoch {
4:     for each Vertex v in G^local in parallel do
5:       if mis[v] == NIL then
6:         for each u in adjacencies(v, G^local) do
7:           if u belongs to G^local then
8:             if mis[u] == NIL then
9:               degree[v] ← (degree[v] + 1)
10:            end if
11:          else
12:            Send^1(u, v, ISNIL)
13:          end if
14:        end for
15:      end if
16:    end for
17:  }
18: /*Build independent set based on coin value*/
19:   for each Vertex v in G^local in parallel do
20:     if coin(v, degree[v]) == 1 then
21:       abuffer.add(v)
22:     end if
23:   end for /*Remove conflicting vertices*/
24:   epoch {
25:     for each Vertex i in abuffer in parallel do
26:       for each j in adjacencies(v, G^local) do
27:         if j belongs to G^local then
28:           if j is in abuffer then
29:             if degree[i] ⇐ degree[j] then
30:               deleteset.add(i)
31:             else
32:               deleteset.add(j)
33:             end if
34:           end if
35:         else
36:           Send(j, i, degree[i], COMPARE)
37:         end if
38:       end for
39:     end for
40:   }
41: end procedure
```

```
42: /*Every Send[1] call invokes a Receive[1]*/
43: procedure Receive[1](u:Vertex, v:Vertex, act:Action)
44:   if act == ISNIL then
45:     if mis[u] == NIL then
46:       Send[1](v,u,NILTRUE)
47:     end if
48:   else
49:     if act == NILTRUE then
50:       degree[u] ← (degree[u] + 1)
51:     end if
52:   end if
53: end procedure
54: /*Every Send call invokes a Receive*/
55: procedure Receive(j:Vertex, i:Vertex, irnd:Real, act:Action)
56:   /*Same as the Receive procedure in Algorithm 7*/.
57: end procedure
```

7.3.4. Distributed Select B. *Select B* (or Luby(B)) algorithm does not add all the vertices in *NIL* state to the append buffer, instead it only adds a subset of vertices in *NIL* state. The subset is calculated based on the random variable *coin*. The random variable coin has two values 0 and 1 that are assigned based degree distribution of vertices. Therefore, Select B algorithm first calculates the degree of each vertex relative to the subgraph. Then, the algorithm selects subset vertices in *NIL* state for an independent set based on the coin value. Afterwards, the algorithm checks in parallel if any adjacent vertices were selected. If so, the algorithm uses degrees of vertices to resolve the conflict and remove any non-independent vertices. Algorithm pseudocode for Select B algorithm is presented inAlgorithm 8.

Calculating vertex degrees in the current subgraph requires communicating with remote ranks. Lines 4 to 16 show the degree calculation. If an adjacent vertex is not in current locality, the algorithm sends a message to the remote locality to check the status of the adjacent vertex (LineLine 12), using the *ISNIL* action to query the status of the adjacent vertex. These messages invoke the $Receive^1$ procedure on the remote locality (Lines 43 to 53). If the adjacent vertex is in the *NIL* state, the remote rank sends back a reply *NILTRUE*.

Lines 19 to 23 show how Select B algorithm invokes *coin*, a random variable, to select a subset of vertices as a candidate for an independent set. The coin function takes a vertex and its degree to decide whether the random variable value is 1 or 0. If the coin function returns 1, the vertex is added to the append buffer.

After adding a subset of vertices in the subgraph to the append buffer, Select B algorithm iterates through the content in the append buffer in parallel. If a vertex in the append buffer has an adjacent vertex that is also in the append buffer, then the vertex with the smaller degree is removed. The vertex that has a lower degree is added to the delete set (Lines 25 to 39). If the adjacent vertex is in a remote locality a message is sent (Line-Line 36). The receive code to handle messages sent (LineLine 36) is similar to the *Receive* procedure in Select A (Lines 36 to 48, in Algorithm 7). Like in Select A, when Select B finishes executing the epoch in Lines 24 to 40, vertices in the append buffer but not in the delete set represents an independent set.

7.4. Experiments & Results

7.4.1. Implementation. The proposed algorithms are implemented on top of an active messaging framework AM++ **[136]**, using *pthreads* for in-node threading.

Graph vertices are equally distributed among participating nodes (1D block distribution). The local graph is represented using compressed sparse row (CSR) format. Every undirected edge is represented using two directed edges.

Algorithm implementations do not require pre-processing of inputs and can deal with parallel edges and self-loops (common artifacts in synthetic inputs). Whenever there is code that iterates through adjacencies of a vertex, the algorithm inserts adjacent vertices to a local set. The body of the loop is executed only if the adjacent vertex is not present in the set (See the code below).

```
1: ...
2: adjacentvertices ← {}
3: for each u in adjacencies(v, G^local) do
4:   if (u! = v) then                          ▷ /*Exclude self-loops*/
```

```
5:     if u not in adjacentvertices then                ▷ /*Exclude parallel edges*/
6:       adjacentvertices.insert(u)
7:       ...
8:     end if
9:   end if
10: end for
```

7.4.2. Experimental Setup. We ran our experiments on a Cray XC system with 2 Broadwell 22-core Intel Xeon processors and 128 GB DDR4-2400 memory per node. For scaling results, we used only up to 32 cores per node to provide uniform scaling. We used Cray MPICH MPI (version 7.4.4) and GCC 6.3.0. We used two processes per node (one per NUMA domain), and we used MPI in thread-multiple mode.

7.4.3. Graph Input. We evaluate the MIS algorithms in-terms of *weak scaling* and *strong scaling*. We use *RMAT* **[23]** synthetic graphs. Two types of RMAT synthetic graphs are used. They are:

- RMAT-1: Graphs based on the current Graph500 **[104]** Breadth First Search benchmark specification with RMAT parameters $A = 0.57$, $B = C = 0.19$ and $D = 0.05$.
- RMAT-2: Graphs generated based on the proposed Graph500 **[56]** SSSP benchmark specification with RMAT parameters $A = 0.50$, $B = C = 0.1$ and $D = 0.3$.

Both types of graphs have 16 undirected edges per vertex. Strong scaling experiments were carried on RMAT-1 and RMAT-2 scale 25 graphs (largest possible before memory exhaustion).

7.4.4. Weak Scaling Results. For weak scaling we compare our implementation to FilteredMIS **[19]** implementation in the CombBLAS **[20]** library. The FilteredMIS algorithm runs Luby(A) with edge filtering. However, implementation presented in this chapter does not perform any edge filtering. We show FilteredMIS results with 0% and 50% edge filtering where no edges and half of the edges are ignored, respectively. The more edges are ignored, the better FilteredMIS performs.

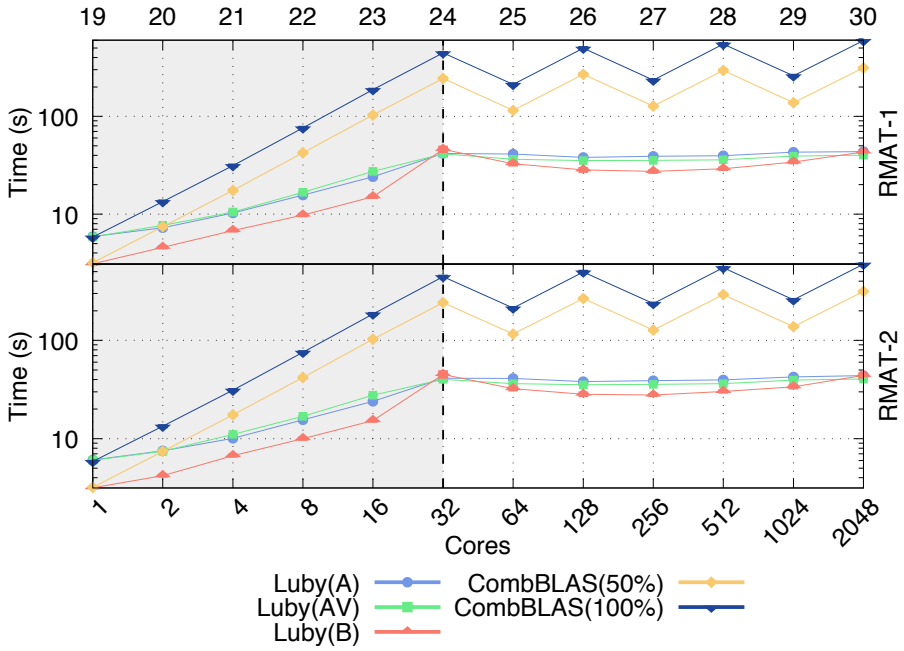


FIGURE 7.2. Weak scaling results of MIS algorithms for RMAT graphs, including FilteredMIS. The shaded area shows the shared memory execution.

Figure 7.2 shows weak-scaling results for of Luby(A), Luby(AV), Luby(B), and FilteredMIS for RMAT-1 and RMAT-2 graph inputs. For both graph inputs distributed Luby algorithms presented in this chapter outperform CombBLAS,FilteredMIS (for both 50% and 100% edge filtering).

Results from distributed execution of FilteredMIS show a zig-zag pattern (when cores > 32). The CombBLAS version we use only supports a square number of tasks; therefore, when executing on a non-square number of nodes (2, 8, 32) we used two *tasks* per node to make the execution on a square number of processes. When the number of tasks per node is two, FilteredMIS execution time decreases and when the number of tasks per node is 1, the execution time increases. We enabled multi-threading in CombBLAS so the tasks can take advantage of multiple cores. We observed that CombBLAS performed the worst with one task per core (no multi-threading).

As per Figure 7.2, Luby(B) performs better in shared memory (when, cores < 32) for both graph inputs. Unlike Luby(A), Luby(B) does not consider all vertices in the subgraph to be in the initial approximation to the independent set. Luby(B) has a choice step, where

it calculates a subset from the subgraph vertices based on the probability distribution (See Section 7.2, "coin" function). Since Luby(B) selects subset from the subgraph vertices, Luby(B) is able to calculate an independent set faster than Luby(A) in an iteration. On the other hand, Luby(A) does not have a choice step and it considers all the vertices in the subgraph as a candidate for an independent set. The data statistics we collected shows that Luby(A) spends most of its time in calculating an independent set in the first iteration, especially in shared-memory execution. For example, the statistics in Table 7.2 are collected for scale 20 RMAT-1 graph on 2 cores. Other scales behave in the same way.

	Luby(A)	Luby(B)
Exec. Time (sec.)	7.13	4.17
Time for 0th iteration (sec.)	6	0.01
No.of Iterations	5	20
Vertices in 0th iteration	1048576	517394
Deleted Set Sz.	917623	2043

TABLE 7.2. Runtime statistics for Luby(A) and Luby(B) on RMAT-1, Scale 20 graph on 2 cores.

Luby(A), however, converges much faster than Luby(B). As shown in Table 7.2, Luby(A) takes 6 iterations to terminate while Luby(B) takes 20 iterations. When the number of iterations are higher, the overhead of global synchronization also increases. At scale 24 (when cores = 32) we see a sudden increase in Luby(B)'s runtime. This is because at scale 24 execution runs in 2 processes and the overhead of communication become significant. The performance of Luby(B) is more affected at scale 24 than the performance of Luby(A). Since the number of iterations are higher in Luby(B), synchronization overhead is more important in Luby(B) than in Luby(A).

The difference between Luby(A) and Luby(AV) are not prominent. Luby(AV) is able to achieve a slight improvement over Luby(A) in distributed execution. This is mainly because Luby(AV) avoid the need to calculate vertex set size using a global reduction and also avoids the need to generate random numbers.

All the Luby algorithms (A, AV, B) presented in this chapter do not show perfect weak scaling performance in shared memory due to the contention created on data structures

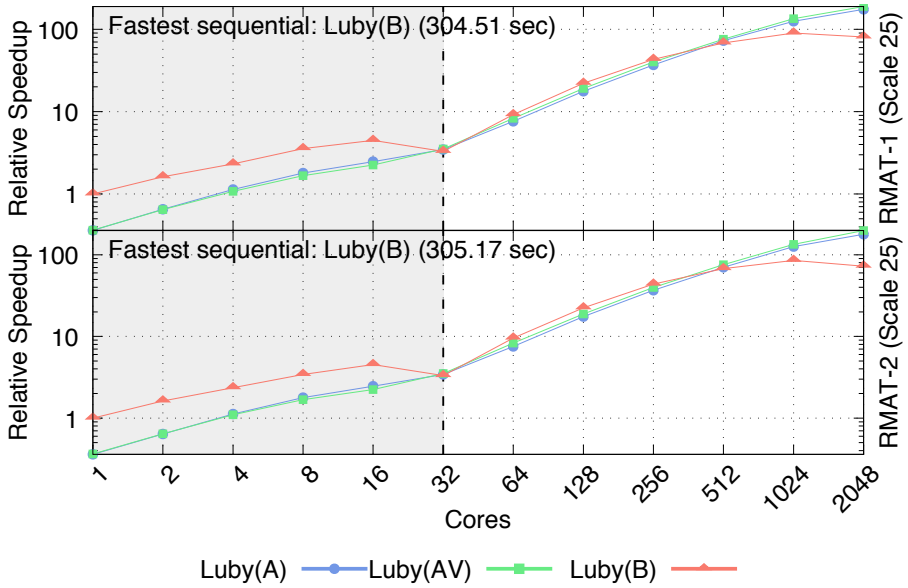


FIGURE 7.3. Strong scaling results of MIS algorithms for RMAT-1 and RMAT-2, Scale 25 graph inputs. Shaded region shows the shared memory execution.

with the increasing number of threads. However, we see good weak scaling in distributed memory for all Luby (A, AV, B) algorithms.

7.4.5. Strong Scaling Results. For strong scaling experiments, we ran MIS algorithms on RMAT-1 and RMAT-2 scale 25 graphs. To have better understanding about how algorithms scale relative to each other, we measured Relative Speedup, $= \frac{T_{ref\_1}}{T_n}$ i.e., the ratio of the execution time of the fastest sequential algorithm, T_{ref_1} and the parallel execution time on n processing elements, T_n.

Figure 7.3 shows the strong scaling results of MIS algorithms presented in this chapter for the graph inputs discussed above. Due to synchronization overhead discussed in the context of weak-scaling results, Luby(B) shows better speedup in shared memory, but in distributed memory Luby(B) speedup drops at higher scales due to higher synchronization overhead.

7.5. Summary

Most of the existing research on MIS focuses on theoretical analysis than practical implementation. Further, the few practical implementations mostly implement Luby's MIS

algorithms. Luby's MIS does not immediately extend as efficient distributed-memory parallel algorithm due to synchronization overheads and subgraph computation overheads.

In this chapter we presented distributed versions of parallel Luby's algorithms. The algorithms we propose minimize the synchronization overhead by overlapping communication and computation and minimizes the subgraph computation overhead using vertex filtering and by maintaining parallel data structures. Our results show that the algorithms we present are several times faster than existing MIS algorithms.

8

FIX MIS

The growing scale of graph data suggests the use of distributed memory hardware as a cost-effective approach to providing necessary compute and memory resources. Existing distributed memory parallel MIS algorithms rely on synchronous communication and use techniques such as subgraph computations. In this chapter, we present an asynchronous distributed-memory parallel graph algorithm that relies on a virtual directed acyclic graph (DAG) that is created during the algorithm execution. We introduce two additional algorithms that save computations by ordering generated work. The first algorithm applies ordering globally to reduce computations, and the second algorithm applies ordering locally at the level of threads to minimize the synchronization overhead. We use two different implementations of Luby's algorithm variants as baseline to compare the performance of the presented algorithms:

(1) vertex-centric Luby A and Luby B implementations, and

(2) the CombBLAS linear-algebra Luby A implementation.

Results show that proposed algorithms outperform both implementations of Luby algorithms, especially in distributed execution. Furthermore, we show that for low-diameter graphs the algorithm that applies global ordering scales better than other algorithms and for high diameter graphs the original asynchronous algorithm and thread-level ordering algorithm show better performance.

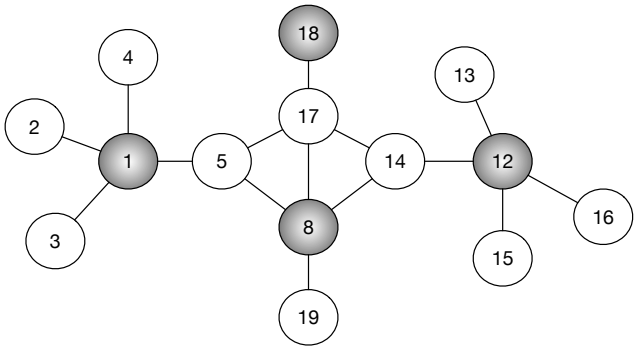


FIGURE 8.1. The gray nodes show a maximal independent set of this graph.

8.1. Introduction

Existing research on parallel algorithms for the MIS problem focuses primarily on theoretical analyses (most often in shared memory settings), and this provides incomplete insight into real-world performance. The available distributed-memory parallel algorithms for MIS are based on Luby's randomized MIS [**91**] algorithms. Luby's algorithms were developed primarily focusing on shared memory; they use techniques that do not scale to distributed-memory. In this chapter we propose an asynchronous distributed-memory parallel MIS algorithm based on a DAG induced on the input graph, using unique randomly assigned vertex identifiers. *FIX*, the algorithm we propose, can further reduce the number of computations by arranging the computations in a particular order. We present two different orderings of FIX, FIX-Bucket and FIX-PQ, and we experimentally evaluate their performance against two different implementations of distributed-memory parallel Luby's algorithms.

The FIX algorithm we present in this chapter is an asynchronous distributed-memory parallel algorithm. It maintains a state for every vertex, indicating whether the vertex is in MIS (FIX1) or not in MIS (FIX0). FIX creates a virtual DAG based on randomly assigned vertex identifiers. Processing starts from the sources of this DAG, and state changes are propagated towards the sinks of the DAG. We show that the algorithm terminates, and that at termination vertices that have the state FIX1 form an MIS.

The correctness of the FIX algorithm does not depend on the order of changes to vertex states, and the execution can proceed in any of the correct orders (the algorithm execution

is not deterministic, but the result is). We derive two algorithms from the basic FIX algorithm by applying the following orderings:

(1) Order work based on where it originates – First process work originating from vertices in FIX1 and then process work originating from vertices in FIX0 (*FIX-Bucket* algorithm);

(2) Order work based on state and then order work based on the monotonic distance from the vertex that started the work (source of the DAG). In other words, the work that is in FIX1 is processed immediately and the work that is not in FIX1 is processed based on the distance from the source. Vertices that have smaller distance from the source gets priority over vertices that have higher distance from the source when the work is in FIX0 state (*FIX-PQ* algorithm).

We show that above two orderings reduce the number of computations relative to the original FIX algorithm.

The performance of the proposed FIX algorithm (including the two ordering variations) is evaluated and compared with Luby's algorithms for both weak scaling and strong scaling. Direct implementations of the original Luby algorithms are inefficient in distributed-memory runtimes, mainly because of subgraph computation, random number generation, and synchronization. **[73]** presented an efficient implementation of two of Luby's seminal algorithms with overlapping computation and communication. In addition, CombBLAS **[20]** has an implementation of Luby's algorithm that is implemented using linear algebra primitives. We use these two implementations as our baselines to compare the performance of the FIX algorithms.

Our results include two types of synthetic graph inputs to evaluate the weak scaling performance and two synthetic graphs and two natural graphs (one with a small diameter and one with a higher diameter) to evaluate the strong scaling performance. Results show that the three FIX algorithms outperform Luby's algorithms implemented in **[73]** and CombBLAS **[20]**. Furthermore, we show that for low-diameter connected graphs, the

FIX-Bucket algorithm has better performance than FIX and FIX-PQ, and for higher diameter graphs, the FIX and FIX-PQ algorithms have better performance than FIX-Bucket algorithm.

In summary, the main contributions of this chapter are:

(1) Development and characterization of three distributed-memory parallel MIS algorithms;

(2) Comparison of weak scaling results to two different implementations of distributed Luby algorithms;

(3) Analysis of results showing that for low-diameter graphs FIX-Bucket outperforms other algorithms and that for high-diameter graphs the FIX or the FIX-PQ algorithm outperforms other algorithms.

8.2. FIX Algorithm

The FIX algorithm begins by generating a directed acyclic graph (DAG) on the input graph G. First, every vertex is assigned a unique random identifier. Every undirected edge (v,u) is given a direction based on the identifiers of v and u, where the vertex with the greater identifier becomes a *successor*, and the vertex with the lesser identifier becomes a *predecessor* (see fig. 8.2). The vertices without predecessors are the *sources* of the induced DAG, and the vertices without successors are the *sinks*.

The FIX algorithm is shown in algorithm 9. FIX has three main subroutines: *Initialize*, *Begin*, and *Receive*. Every *Send* call in the algorithm invokes the *Receive* subroutine. The FIX algorithm maintains a state per each vertex. The state is represented using the maps *mis* and *count*, where the *mis* map represents the membership in MIS (FIX0 indicates not in MIS, and FIX1 indicates in MIS), and the *count* map represents the number of predecessors with state FIX1. In *Initialize*, the *mis* state of every vertex is initialized to *UNFIX*, indicating that the membership of the vertex in the MIS is not decided yet, and the *count* state is initialized to 0, indicating that no predecessors are in MIS. The FIX algorithm first adds source vertices to the MIS (lines 1 to 8) by changing their state to FIX1. Upon changing the state of a source, a message is sent to its successors to notify them of the change.

The initial messages sent from *Begin* are handled by *Receive*. The *Receive* routine is a *message handler* that is implicitly invoked per every *Send* call. If a message comes from a vertex with a state of FIX1, the state of the receiving vertex u is immediately changed to FIX0, and all successors of v are notified of this change (lines 1 and 5). If a message comes from a vertex in the FIX0 state, the count of predecessors with FIX0 state is incremented. If all predecessors have the FIX0 state (line 8), then the vertex joins the MIS. Its state is set to FIX1, and all of its neighbors are notified. The algorithm traverses the DAG induced on the input graph G by following the predecessor and successor links, and it terminates when the states of all sinks are set to FIX0 or FIX1, or, equivalently, when there are no vertices left in the state UNFIX.

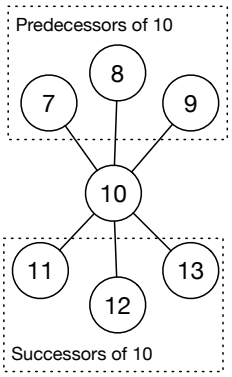


FIGURE 8.2. Successors and predecessors of a vertex.

At termination, vertices with state FIX1 form an MIS. To prove this, we first need to show that at termination, all vertices must either be in FIX1 or FIX0 state. Suppose there is a vertex, v, in the UNFIX state at termination. The vertex v can be in any one of the following 4 situations within the induced DAG:

(1) v has neither predecessors nor successors;

(2) v has predecessors but not successors;

(3) v has no predecessors but it has successors; or

(4) v has both predecessors and successors.

In cases 1 and 3 the *Begin* routine promotes the state of v to FIX1 (line 3 in algorithm 9). In cases 2 and 4, if v is in UNFIX state, then, there is no predecessor with state FIX1 because of the message sent on line 11 in algorithm 9. A message sent in line 11 calls the *Receive* procedure and it assures successor of a FIX1 vertex is in FIX0 (lines 1 and 2). Then, either all the predecessors must be in the FIX0 state, or there is at least one predecessor that is in the UNFIX state. If all the predecessors are in FIX0, v must be in FIX1 (lines 8 and 9 in algorithm 9). Since we assume v is in the UNFIX state, there must be at least one predecessor in the UNFIX state. Let this predecessor of v be u. We can make a similar argument

Algorithm 9 FIX Algorithm

Initialize $G = (V, E)$:

```
1: for each v in V do
2:   mis[v] ← UNFIX, count[v] ← 0
3: end for
```

Begin $G = (V, E)$:

```
1: for each v in V do
2:   if v is a source then
3:     mis[v] ← FIX1
4:     for each u in Successors of v do
5:       Send(u, FIX1)
6:     end for
7:   end if
8: end for
```

Receive u, vstate:

```
1: if vstate == FIX1 then
2:   mis[u] ← FIX0
3:   for each u_s in Successors of u do
4:     Send(u_s, FIX0)
5:   end for
6: else
7:   count[u] ← count[u] + 1
8:   if count[u] == |Predecessors| then
9:     mis[u] ← FIX1
10:    for each u_s in Successors of u do
11:      Send(u_s, FIX1)
12:    end for
13:  end if
14: end if
```

for u and can conclude that u has a predecessor in the UNFIX state. We continue the argument until we reach a vertex without predecessors (since the DAG is finite and acyclic, we will reach a source level vertex). However, by cases 1 and 3, we know that a vertex without predecessors will be promoted to FIX1. Therefore, the sources of the DAG (they are also predecessors) cannot be in the UNFIX state. Therefore, our assumption that there is a vertex in UNFIX state after termination causes a contradiction, and we can conclude that every vertex is in either FIX0 or FIX1 after algorithm termination.

Furthermore, none of the vertices that are in the state FIX0 can be changed to FIX1 because a vertex's state transitions to FIX0 if, and only if, it has a neighbor that is in FIX1. Therefore, the set of vertices in the FIX1 state cannot be expanded any further. Hence, a set of vertices that is in the FIX1 state is an MIS.

Compared to Luby's algorithms, the FIX algorithm avoids the overhead of random number generation in every iteration by assigning a random permutation of identifiers to vertices. In addition, FIX also avoids the need to maintain a subgraph for every iteration. Maintaining a subgraph requires $O(n)$ space; hence, compared to Luby's algorithms, FIX is space efficient.

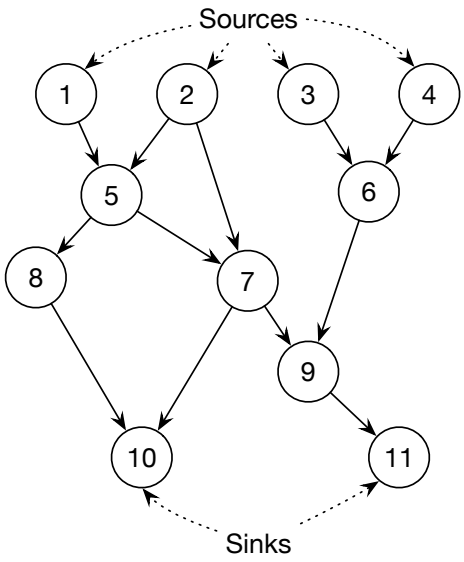


FIGURE 8.3. The virtual DAG created based on predecessors and successors.

8.2.1. Distributed FIX. The distributed implementation of FIX divides vertices equally among participating processes and stores those vertices and their adjacencies locally. G^{local} represents the local subgraph. Every rank runs some number of threads, and distributed messages are exchanged between ranks within *epochs*.

An epoch is a code region where distributed programs can exchange messages. A global barrier is executed at the start of an epoch and at the end of an epoch. Computation and communication can be overlapped within epochs.

Our distributed implementation is listed in algorithm 11. The implementation follows the single program, multiple data (SPMD) paradigm, in the sense that every rank runs the same program. Every rank maintains two per-vertex counter maps of counters (the counters are indexed by the vertex identifier). The first counter, *predecessor count* (*predcount*), counts the number of predecessors per each vertex and this is calculated at the initialization of the algorithm (line 19). The second counter, *predecessor completed count* (*predcc*) keeps track of the number of predecessors that were moved to FIX0 state. This counter is updated during algorithm execution. These two counters are used to identify whether all the predecessors of a vertex have been moved to FIX0 state. For a given vertex, v, $predcount[v] = predcc[v]$ means that all the predecessors of the vertex v are in FIX0, and v can be set to FIX1.

The algorithm execution starts with the *FIXMIS* (lines 16 to 27) procedure. Inside an epoch, each rank goes through vertices in the local graph in a parallel thread and changes the state of vertices that have a zero *predcount*. When a vertex's state is changed, the vertex notifies all of its successors (line 22). The notification implicitly invokes the *Receive* procedure, as in algorithm 9. The *Receive* function first checks whether the destination vertex's state is already changed to either FIX1 or FIX0 (Line 29). If not, it checks whether the source vertex's state has been changed from UNFIX to FIX1. If so, the receiving vertex's state is changed to FIX0, and the receiving vertex notifies, in turn, all of its successors (Lines 32 to 38). If the source vertex's state is changed from UNFIX to FIX0, the receiving vertex increments its *predcc* counter. Then, if all of the receiving vertex's neighbors with higher priorities are transferred to FIX0 state (i.e., if predcc = predcount), the receiving vertex is promoted to FIX1 state (lines 28 to 52).

In our particular implementation of the FIX algorithm, we did not use a thread-level distribution of vertices. Therefore, counters and MIS states can be updated by two threads at the same time. To avoid race conditions we use atomic operations.

As is, the algorithm 11 suffers from load imbalance within threads in shared memory execution. The algorithm processes each vertex in parallel (line 18), and some of the vertices, that threads process are not sources (i.e., they do not satisfy the condition $predcount[v] == 0$, Line 19). Because of that, those threads do little work compared to threads that process sources.

To balance the load between threads, we pre-calculate the source vertices and insert them into an append buffer, and *FIXMIS* procedure iterates over the sources instead of all vertices in the graph. This way we make sure that every thread processes about the same number of source vertices. The modified distributed MIS with better thread level load balance is given in algorithm 10 (lines 5 to 9 and line 18).

Algorithm 10 Modified FIX for better load balance between threads

```
1: ...
2: procedure INITIALIZE(G^local)
3:   for each Vertex v in G^local in parallel do
4:     ...
5:     for each u in adjacencies(v, G^local) do
6:       if (u < v) then
7:         predcount[v] ← predcount[v] + 1
8:       end if
9:     end for
10:    if predcount[v] == 0 then
11:      appendbuffer.insert(v)
12:    end if
13:  end for
14: end procedure
15:
16: procedure FIXMIS(G^local)
17:   ...
18:   for each Vertex v in appendbuffer in parallel thread do
19:     ...
20:   end for
21:   ...
22: end procedure
23: ...
```

Algorithm 11 Distributed Memory Parallel FIX Algorithm

```
1: predcount ← {0...0}
2: predcc ← {0...0}
3: procedure INITIALIZE(G^local)
4:   for each Vertex v in G^local in parallel do
5:     mis[v] ← UNFIX
6:     predcount[v] ← 0
7:     predcc[v] ← 0
8:     for each u in adjacencies(v, G^local) do
9:       if (u < v) then
10:        predcount[v] ← predcount[v] + 1
11:      end if
12:    end for
13:  end for
14: end procedure
15:
16: procedure FIXMIS(G^local)
17:   epoch {
18:     for each Vertex v in G^local in parallel thread do
19:       if (predcount[v] == 0) then
20:         mis[v] ← FIX1
21:         for each u in adjacencies(v, G^local) do
22:           Send(u, v, mis[v])
23:         end for
24:       end if
25:     end for
26:   }
27: end procedure
```

```
28: procedure RECEIVE(destv, srcv, srcstate)
29:   if mis[destv]! = UNFIX then
30:     return
31:   end if
32:   if srcstate == FIX1 then
33:     mis[destv] ← FIX0
34:     for each u in adjacencies(destv, G^local) do
35:       if (u > destv) then
36:         Send(u, destv, mis[destv])
37:       end if
38:     end for
39:   else                                              ▷ srcstate=FIX0
40:     if (srcv < destv) then
41:       predcc[destv] ← (predcc[destv] + 1)
42:     end if
43:     if predcc[destv] == predcount[destv] then
44:       mis[destv] ← FIX1
45:       for each u in adjacencies(destv, G^local) do
46:         if (u > destv) then
47:           Send(u, destv, mis[destv])
48:         end if
49:       end for
50:     end if
51:   end if
52: end procedure
```

8.3. Ordering in FIX

The FIX algorithm discussed above is an unordered algorithm. Therefore, the order in which the states are updated does not affect the correctness of the algorithm, but with certain ordering schemes we can reduce the amount of computations in FIX.

The FIX algorithm decides a vertex is in the MIS (FIX1) if all of its predecessors are not in MIS. Therefore, the sooner a vertex finds out that it is in MIS, the more computations it can avoid. For example, in Figure 8.4, the vertex v has four predecessors; out of those four, one predecessor is in state FIX1. If v gets a state change from the vertex in FIX1 first, v will not have to execute the logic relevant to increase predcc count and the "if condition" that compares predcc and predcount. If the thread level vertex distribution is random, the

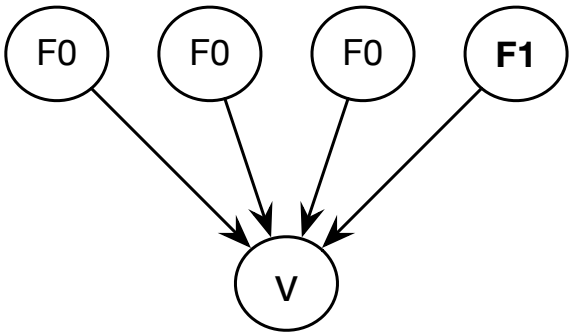


FIGURE 8.4. DAGs created by FIX algorithm execution for the graph input in 8.1.

predcc count will be a atomic variable and processing FIX1 predecessors first will considerably reduce the contention when processing a large graph, especially in shared memory execution. However, if we process FIX0 predecessors before FIX1, the computation we did for FIX0 state changes have no effect on the final outcome.

We came up with two forms of orderings that reduce computations as described above. The first approach orders the execution on the state. That is, messages generated are separated into two *buckets*. The first bucket contains the work originated from vertices whose states transferred to FIX1. The second bucket contains the work originated from vertices who transferred their states to FIX0. How execution proceeds in this ordering is depicted in Figure 8.5. Implicitly, the bucket ordering traverse the DAG in a level synchronous fashion. We call this algorithm FIX-Bucket and it is discussed in detail in Section 8.3.1.

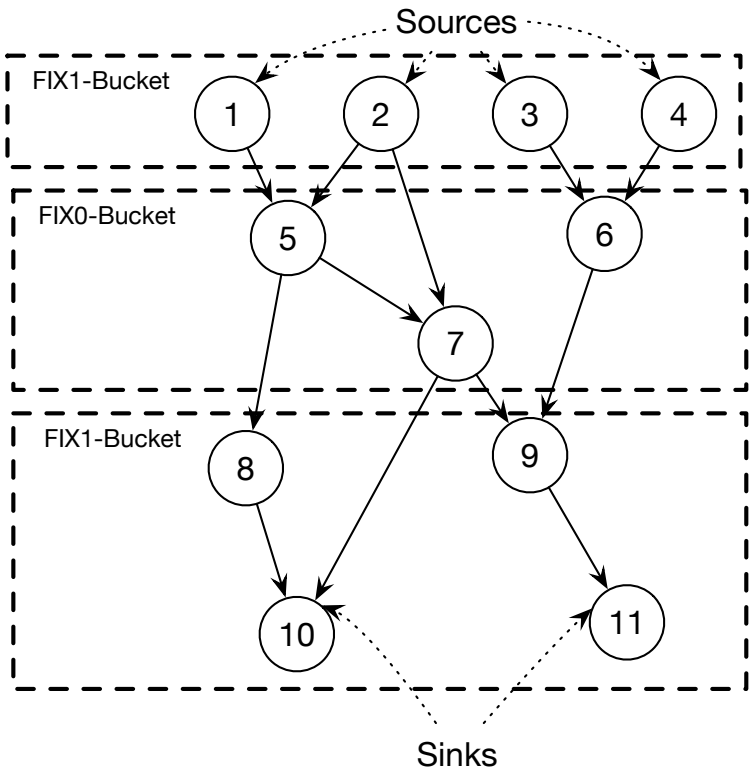


FIGURE 8.5. How DAG is executed in FIX-Bucket ordering.

The above ordering involves a global barrier to be executed in every level. The second ordering we propose (FIX-PQ), skips global synchronization and performs local ordering on work. The work generated is ordered based on the state of the source vertex as well as on the distance from the originating source (in the DAG). The work with FIX1 state is immediately processed and work that has FIX0 state is ordered by distance from the relevant source. When the distance is higher the priority is also high. The purpose of distance ordering is to propagate state changes deeper into the DAG.

Both the FIX-Bucket and FIX-PQ algorithms require fewer computations than the original FIX algorithm. Table 8.1 shows the number of computations saved by each algorithm relative to the original FIX algorithm. The "Skipped" denotes the number of times the *Receive* function is invoked even though there is no state change or increase in predcc count. The "Called" denotes the number of times the *Receive* function was invoked and there was a state change or predcc counter was increased.

	FIX	FIX-PQ	FIX-Bucket
Skipped	84208408 (65.10%)	119592110 (92.46%)	112771147 (87.16%)
Called	45130908 (34.89%)	9747206 (7.53%)	16613655 (12.84%)
Total	129339316	129339316	129339316

TABLE 8.1. Saved computations for FIX-Bucket and FIX-PQ algorithms, relative to FIX. Results are for scale 23, RMAT-1 (refer Section 8.4.3 for details) graph with 4 cores running in parallel.

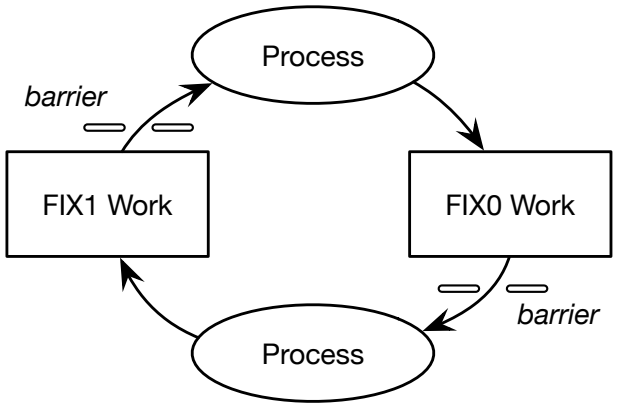


FIGURE 8.6. An overview of the FIX-Bucket algorithm.

As per the statistics in Table 8.1, the FIX-PQ algorithm saves the most amount of work. This is primarily because FIX-PQ is ordered using both distance and state and so able to propagate FIX1 state changes to successors faster than FIX-Bucket is able to.

In the following section we discuss the FIX-Bucket and the FIX-PQ algorithms in detail.

8.3.1. FIX-Bucket Algorithm. The *FIX-Bucket* algorithm maintains two distributed containers, called *buckets* (Figure 8.6). Locally, a bucket is implemented as an append buffer, but before processing the append buffer all ranks must globally synchronize. In other words an append buffer is processed within an epoch. The first container, which we call *fix0bucket*, stores all the vertices where state is transferred to FIX0. The second container, *fix1bucket*, stores all the vertices where state is changed to FIX1.

The FIX-Bucket algorithm is listed in Algorithm 12. The initialization code is the same as the initialization procedure in Algorithm 11. The FIX-Bucket algorithm starts in the

same way as the FIX algorithm but at the end of the *FIXBucketMIS* (Lines 9 to 21) procedure, *FIXBucketMIS* calls the *HandleBuckets* procedure. In every rank, the *HandleBuckets*(Lines 23 to 46) procedure iterates through each bucket in parallel threads and sends state changes to successors. However, at a given time, all the ranks iterate through only one bucket. Therefore, unlike in the FIX algorithm, in the FIX-Bucket algorithm there are no messages originating from FIX0 vertices when processing *fix1buckets*. Also there are no messages originating from FIX1 vertices when processing *fix0buckets*. The *Receive* function handles incoming messages and populates them to appropriate buckets.

Algorithm 12 Distributed Memory Parallel FIX-Bucket Algorithm

```
1: predcount ← {0...0}
2: predcc ← {0...0}
3: fix0bucket ← {}
4: fix1bucket ← {}
5: procedure INITIALIZE(G^local)
6:   /*Same as the Initialization procedure in Algorithm 11*/.
7: end procedure
8:
9: procedure FIXBUCKETMIS(G^local)
10:   epoch {
11:     for each Vertex v in G^local in parallel do
12:       if (predcount[v] == 0) then
13:         mis[v] ← FIX1
14:         for each u in adjacencies(v, G^local) do
15:           Send(u, v, mis[v])
16:         end for
17:       end if
18:     end for
19:   }
20:   HandleBuckets()
21: end procedure
22:
23: procedure HANDLEBUCKETS(void)
24:   while fix0bucket not empty and fix1bucket not empty do
25:     /*Handle FIX0 bucket.*/
26:     epoch {
27:       for each Vertex v in fix0bucket in parallel do
28:         for each u in adjacencies(v, G^local) do
29:           if (u > v) then
30:             Send(u, v, mis[v])
31:           end if
32:         end for
33:       end for
34:     }
35:     /*Handle FIX1 bucket.*/
36:     epoch {
37:       for each Vertex v in fix1bucket in parallel thread do
38:         for each u in adjacencies(v, G^local) do
39:           if (u > v) then
40:             Send(u, v, mis[v])
41:           end if
42:         end for
43:       end for
44:     }
45:   end while
46: end procedure
```

```
47: procedure RECEIVE(destv, srcv, srcstate)
48:   if mis[destv]! = UNFIX then
49:     return
50:   end if
51:   if srcstate == FIX1 then
52:     mis[destv] ← FIX0
53:     fix0bucket → push(destv)
54:   else                                              ▷ srcstate=FIX0
55:     if (srcv < destv) then
56:       predcc[destv] ← (predcc[destv] + 1)
57:     end if
58:     if predcc[destv] == predcount[destv] then
59:       mis[destv] ← FIX1
60:       fix1bucket → push(destv)
61:     end if
62:   end if
63: end procedure
```

8.3.2. FIX-PQ Algorithm. The execution time of the FIX algorithm depends on the maximum height of the virtual DAG (e.g., Figure 8.3). The longest path of the DAG is important, especially when deciding whether a vertex should be transferred to FIX1 state, because a vertex's state can only be updated to FIX1 if all predecessors of the vertex are in FIX0 state. Processing work generated by FIX1 vertices is straightforward since neighbors of FIX1 must be transferred to FIX0 irrespective of the number of predecessors. Furthermore, notifying successors about a state change of a vertex to FIX1 helps to save computations. Therefore, the FIX-PQ algorithm processes FIX1 work immediately and orders work generated by FIX0 vertices based on the distance from a source. The ordering is applied at the thread level to avoid the overhead of synchronization.

The FIX-PQ algorithm is listed in Algorithm 13. The algorithm keeps an array of priority queues and the size of the array is equal to the number of threads (Line 3). The function *getnumthreads* returns the number of threads the algorithm is executing. The *Initialize* procedure is the same as the *Initialize* procedure in Algorithm 11. The *FIXPQMIS* procedure (Lines 8 to 20) adds source vertices to the MIS and notifies state changes. In the same epoch the algorithm calls the function *HandlePQs* (Line 18). Also, algorithm uses *workitem* structure to encapsulate destination vertex, source vertex, source state and distance.

For each shared memory thread, the *HandlePQs*(Lines 22 to 40) procedure pop work from the priority queue and process it. The priority queue only contains work related to FIX0 predecessors, therefore, the processing logic updates predcc count and checks whether the destination vertex can be promoted to FIX1. The *HandlePQs* procedure is executed until there is work available in the system. The function *terminate()* returns *True* when the termination detection detects that there is no more work to be processed.

The *Receive* function (Lines 41 to 52) processes work items originating from FIX1 vertices and work items originating from FIX0 vertices are added to the priority queue for the current thread.

Note that *HandlePQs*' procedure is invoked within the epoch of the *FIXPQMIS* procedure. Therefore, Algorithm 13 executes asynchronously, but work is ordered at the thread level.

Algorithm 13 Distributed Memory Parallel FIX-PQ Algorithm

```
1: predcount ← {0...0}
2: predcc ← {0...0}
3: pqs[getnumthreads()]
4: procedure INITIALIZE(G^local)
5:   /*Same as the Initialization procedure in Algorithm 11*/.
6: end procedure
7:
8: procedure FIXPQMIS(G^local)
9:   epoch {
10:    for each Vertex v in G^local in parallel do
11:      if (predcount[v] == 0) then
12:        mis[v] ← FIX1
13:        for each u in adjacencies(v, G^local) do
14:          Send(u, v, mis[v])
15:        end for
16:      end if
17:    end for
18:    HandlePQs()
19:  }
20: end procedure
21:
22: procedure HANDLEPQS(void)
23:   while terminate() is False do
24:     while pqs[getthreadid()] not empty do
25:       workitem wi ← pqs[getthreadid()].pop()
26:       if (wi.srcv < wi.destv) then
27:         predcc[wi.destv] ← (predcc[wi.destv] + 1)
28:       end if
29:       if predcc[wi.destv] == predcount[wi.destv] then
30:         mis[wi.destv] ← FIX1
31:         for each u in adjacencies(wi.destv, G^local) do
32:           if (u > wi.destv) then
33:             workitem wn(u, wi.destv, mis[wi.destv], (wi.dist+1))
34:             Send(wn)
35:           end if
36:         end for
37:       end if
38:     end while
39:   end while
40: end procedure
```

```
41: procedure RECEIVE(wi : workitem)
42:   if wi.srcstate == FIX1 then
43:     mis[wi.destv] ← FIX0
44:     for each u in adjacencies(wi.destv, G^local) do
45:       if (u > wi.destv) then
46:         Send(u, wi.destv, mis[wi.destv])
47:       end if
48:     end for
49:   else                                          ▷ wi.srcstate=FIX0
50:     pqs[getthreadid()] → push(wi)
51:   end if
52: end procedure
```

8.4. Implementation & Experiments

8.4.1. Implementation. The proposed algorithms are implemented on top of an MPI-wrapped, lightweight, active messaging framework called AM++ **[136]**. Graph vertices are equally distributed among participating nodes (*1D block distribution*). The local graph is represented using a *compressed sparse row* format. In the local graph each undirected edge is represented using two directed edges.

Algorithm implementations are resilient to parallel edges and self-loops. Both parallel edges and self-loops are handled within algorithms. Whenever there is code that iterates through adjacencies of a vertex, the algorithm inserts adjacent vertices to a local set. The body of the loop is executed only if the adjacent vertex is not present in the set.

8.4.2. Experiment Setup. We ran our experiments on a Cray XC system that has 2 Broadwell 22-core Intel Xeon processors. Our experiments only used up to 16 cores to uniformly double the problem size and to double the number of processors in weak scaling. Each node consists of 128 GB DDR4-2400 memory. The MPI implementation we used is Cray MPICH (version 7.4.4).

Preliminary results show that to get the best performance results for all the algorithms, we need to run two processes per node (because there are two sockets per node) in MPI thread multiple mode.

Graph	Vertices	Edges
Friendster	65608366	1806067135
US Road	23947347	58333344
RMAT-1(26)(rmat1)	67108864	1073741824
RMAT-2(26)(rmat2)	67108864	1073741824

TABLE 8.2. Graph inputs and their attributes used in strong scaling experiments

8.4.3. Graph Input. We evaluate the MIS algorithms in terms of *strong scaling* and *weak scaling*. For weak scaling experiments, we use *R-MAT* [**23**] synthetic graphs. Two types of RMAT synthetic graphs are used. They are:

- RMAT-1: Graphs based on the current Graph500 [**104**] Breadth First Search benchmark specification with R-MAT parameters $A = 0.57$, $B = C = 0.19$ and $D = 0.05$.
- RMAT-2: Graphs generated based on the proposed Graph500 [**56**] SSSP benchmark specification with R-MAT parameters $A = 0.50$, $B = C = 0.1$ and $D = 0.3$.

Strong scaling experiments were carried out on the graphs listed in table 8.2.

8.5. Results

The weak scaling results of the proposed algorithms are compared against two different implementations of Luby algorithms : 1. Luby(A) and Luby(B) discussed in [**73**] and; 2. Luby(A) in CombBLAS library. Section 8.5.1 discusses these results. In the results, when we use *FIX** we refer to FIX, FIX-Bucket and FIX-PQ algorithms collectively.

8.5.1. Weak Scaling Results.

8.5.1.1. *Comparison with Vertex-Centric Luby Algorithms.* Weak scaling results of FIX* algorithms are presented in Figure 8.7 on RMAT-1 and RMAT-2 graph inputs. Both in shared memory and in distributed memory FIX* algorithms outperform Luby(A). Luby(B) shows better performance for few initial scales in shared memory and as the execution moves to distributed memory, FIX* algorithms supersede the performance of Luby(B). We were unable to collect Luby(A) results for scale 32 graphs at 64 nodes due to an "out of memory" error.

Ordering helps to improve the performance of FIX* algorithms by reducing the required number of computations (See Section 8.3). In shared memory, we see that FIX-PQ

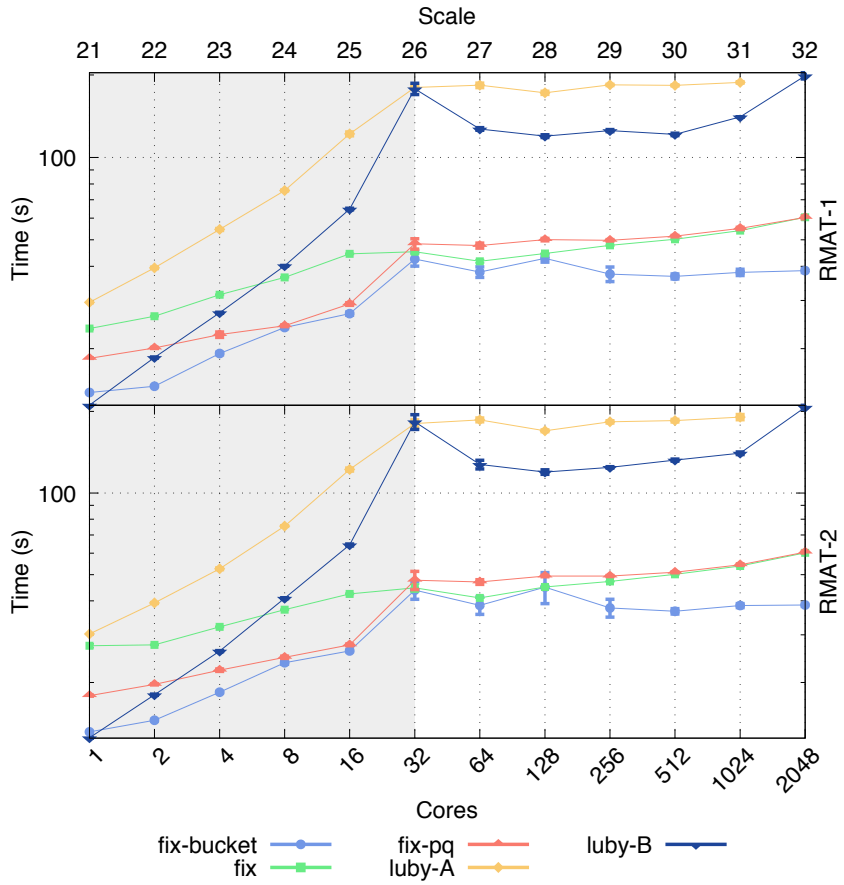


FIGURE 8.7. FIX* & Luby algorithms weak scaling results for RMAT-1 and RMAT-2 graphs. Shaded region shows the shared memory execution.

shows better performance than FIX. In shared memory FIX-PQ is able to avoid more computations than FIX algorithm, but, when the execution becomes distributed, performance improvement in FIX-PQ is not prominent compared to FIX. As the execution becomes distributed, the compute / communication ratio decreases and more time is spent on communication. Therefore, the performance improvement gained by reducing computation is small relative to the much higher overhead cost of distributing execution.

The FIX-Bucket algorithm shows better weak scaling results in distributed execution. Starting from the sources of the DAG, The FIX-Bucket algorithm progress by processing vertices in each level. This way, algorithm assures that predecessors are always processed and hence it is able to avoid most of the redundant computations. However, the FIX-Bucket algorithm requires a barrier synchronization after processing each level in the

graph. Overhead of this barrier synchronization is not as significant as RMAT graphs generally have fewer synchronization levels. Also, RMAT graphs are well-connected; therefore, there is enough work to keep all processors busy.

8.5.1.2. *Comparison with CombBLAS Luby Algorithms.* The FilteredMIS [**19**] algorithm runs Luby(A) with edge filtering. However, implementations presented in this chapter do not perform any edge filtering. We show FilteredMIS results with 0% and 50% edge filtering where no edges and half of the edges are ignored, respectively. The more edges are ignored, the better FilteredMIS performs.

Distributed execution of FilteredMIS results show a zig-zag pattern (when cores > 32). The CombBLAS version we use only supports a square number of tasks; therefore, when executing on a non-square number of nodes (2, 8, 32) we used two *tasks* per node to make the execution on a square number of processes. When the number of tasks per node is two, FilteredMIS execution time decreases and when the number of tasks per node is 1, the execution time increases.

8.5.2. Strong Scaling Results. For strong scaling experiments, we ran MIS algorithms on graphs listed in table 8.2 over 1–1024 cores. To have better understanding about how algorithms scale relative to each other, we measured Relative Speedup, $= \frac{T_{ref\_1}}{T_n}$ i.e., the ratio of the execution time of the fastest sequential algorithm, T_{ref_1} and the parallel execution time on n processing elements, T_n.

8.5.2.1. *Low diameter graphs.* Strong scaling results of FIX* algorithms and Luby's algorithms on RMAT graphs are shown in Figure 8.9. For both RMAT-1 and RMAT-2 graphs we see better speedup in FIX* algorithms than Luby's algorithms. We see some drop in speedup when algorithm execution moves from shared memory to distributed memory. When execution reaches 2048 cores, the Luby B algorithm speedup decrease due to overhead of barrier synchronization. Further, we see an unusual scaling behavior of Luby(B) on the Friendstr network both in shared memory and for some cores in distributed memory. However, we validated our results. FIX* algorithms show better scaling behavior compared to both Luby algorithms.

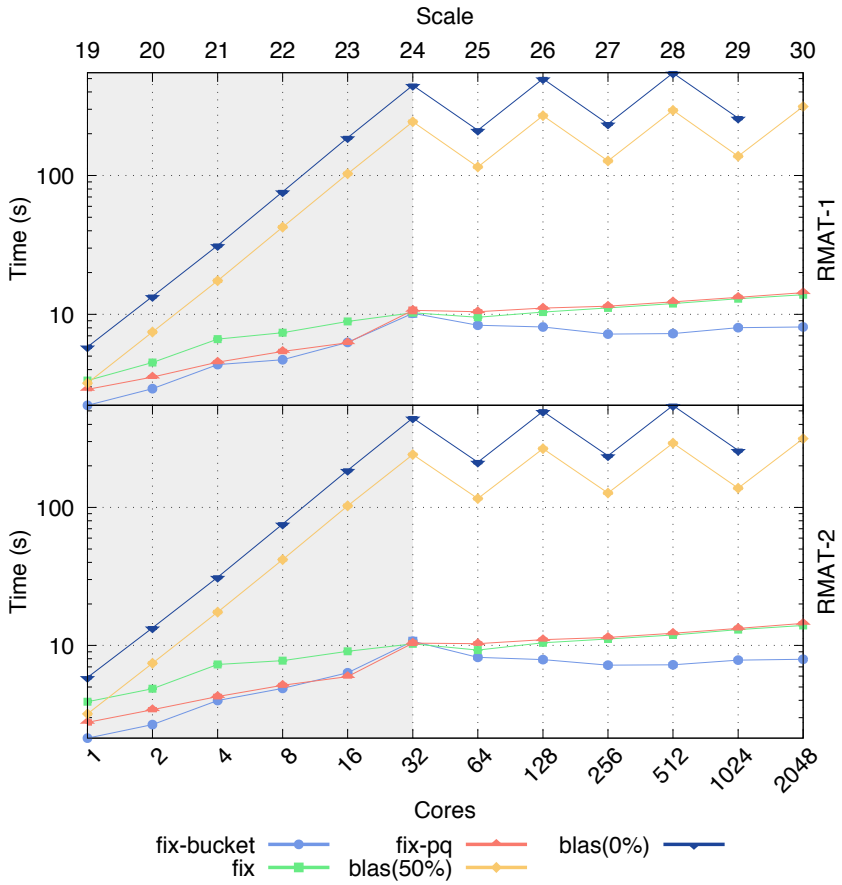


FIGURE 8.8. FIX* & CombBLAS FilteredMIS algorithms, weak scaling results for RMAT-1 and RMAT-2 graphs. Shaded region shows the shared memory execution.

8.5.2.2. *Higher diameter graphs.* In general, both RMAT-1 and RMAT-2 generate low diameter graphs (diameter 10-50, the diameter is estimated using the approximate diameter algorithm) relative to road networks. Further, Friendster graph's diameter is 32 [**84**]. Compared to the diameters of those graphs, road networks have larger diameters. Figure 8.10 shows strong scaling results for US road networks. The approximate diameter of a US road network is 850 [**84**].

As per Figure 8.10, both FIX and FIX-PQ show sound, strong scaling results for US road networks. Due to large diameter in road networks, the FIX-Bucket algorithm executes many synchronization phases (≈850 barriers). Because of the overhead of barrier synchronization FIX and FIX-PQ outperform FIX-Bucket. We see the FIX algorithm achieving

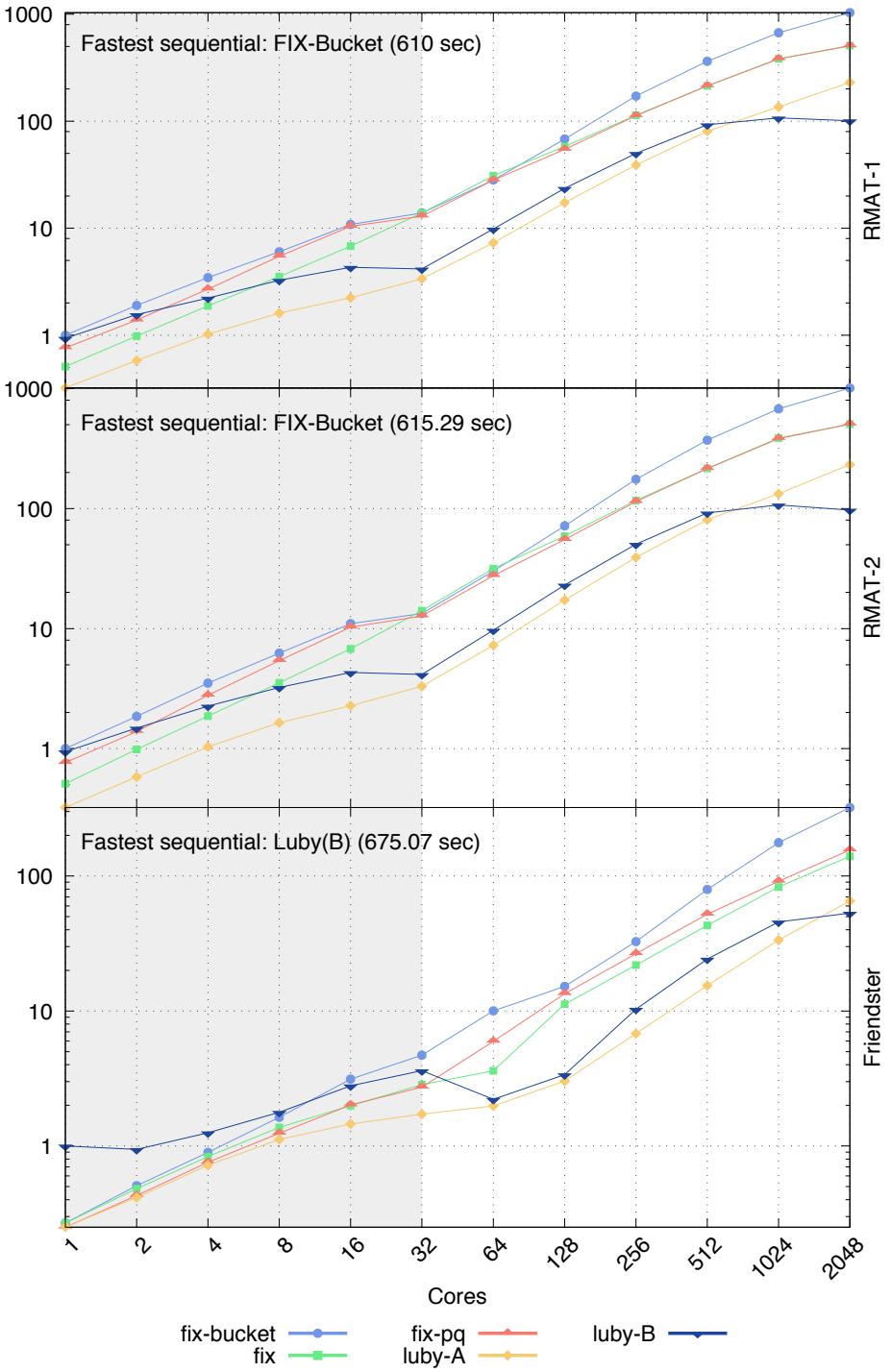


FIGURE 8.9. Strong scaling results of FIX* algorithms and vertex centric Luby's algorithms

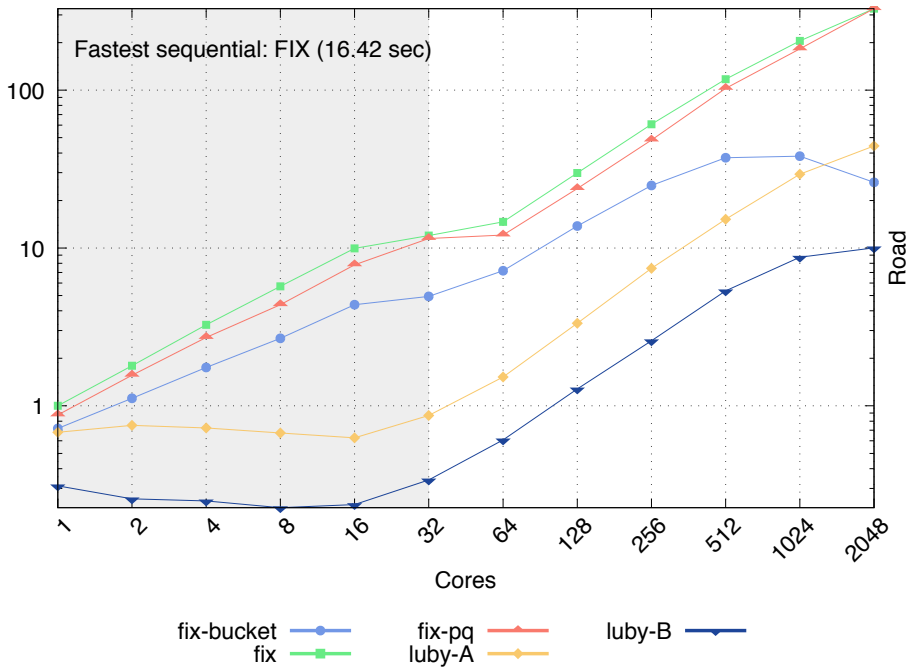


FIGURE 8.10. Strong scaling results of FIX* algorithms and vertex centric Luby's algorithms.

slightly better speedup compared to FIX-PQ. Unlike the RMAT and the Friendstr graphs, road networks is not a well- connected graph and it has a higher-diameter compared to RMAT and Friendstr. Therefore, the number of saved computations in FIX-PQ in road network is less and it is unable to gain performance over the FIX algorithm.

8.6. MIS in AGM

The proposed MIS algorithm maintains a state per each vertex. The state determines whether a vertex is in MIS or not in MIS. Therefore, if a state change occurs in a vertex, it needs to notify low priority neighbors. I define $WorkItems^{mis} \subseteq Vertex \times Vertex \times State$. The first vertex value defines the destination, the second vertex value defines the source, and the state value defines whether the source vertex is in MIS or not. If state is 1 (FIX1), then the source vertex is in MIS, and if state is 0 (FIX0) , then the source vertex is not in MIS. The AGM model for the proposed algorithm uses two states: *mis* state stores the state of each vertex and *waitcnt* stores the number of higher priority neighbors that are in FIX0 state. Further, I assume the input graph is $G = (V, E, vmaps = \{wait\}, emaps = \{\})$. Here, the

wait mapping stores the number of higher priority neighbors of a vertex and this can be calculated as a pre-processing step. The processing function for the proposed algorithm consists of three statements. The first statement is enabled if the receiving *workitem*'s source vertex is in the MIS. The second statement is enabled if the receiving *workitem*'s source vertex is not in MIS and if the value of *wait* for the destination vertex is less than the *waitcnt* for the destination vertex. The last statement is enabled if the source vertex is not in MIS and if all higher priority neighbors of the destination vertex are also in FIX0 state.

As discussed above, there are several ways to order *workitems*. A straight forward way to order *workitems* is not to relate any *workitems* (See Definition 16). Such a relation creates a Chaotic ordering of *workitems*. I name the algorithm made out of Chaotic ordering as *FIX*. In Definition 16, I divide *workitems* into two equivalence classes. The first equivalence class contains the FIX1 *workitems* and the second equivalence class contains the FIX0 *workitems* (*FIX-Bucket*).

DEFINITION 16. *$<_{ch}$ is a binary relation defined on $WorkItems^{mis}$ where, $w_1 \not<_{ch} w_2$ nor $w_2 \not<_{ch} w_2$ for any $w_1, w_2 \in WorkItems^{mis}$.*

DEFINITION 17. *$<_{fixbk}$ is a binary relation defined on $WorkItems^{mis}$ where, $w_1 <_{fixbk} w_2$ if $w_1[2] < w_2[2]$.*

DEFINITION 18. $\pi^{mis} : WorkItems^{mis} \longrightarrow 2^{WorkItems^{mis}}$

$$\pi^{mis}(w) = \begin{cases} \#Statement\ 1 \\ \{w_k | w_k \in < w_n[0] \in neighbors(w[0])\ and \\ \ w_n[0] > w[0]\ and\ w_n[1] \longleftarrow w[0]\ and \\ \ w[2] \longleftarrow mis(w[0]) \\ \quad and\ w_n[2] = w[2]+1 >, \\ \ < mis(w[0]) \longleftarrow FIX0 >, \\ \quad < if\ FIX1 == w[2] > \} \cup \\ \#Statement\ 2 \\ \{w_k | w_k \in < \{\} >, \\ \quad < waitcnt(w[0]) \longleftarrow (waitcnt(w[0])+1) >, \\ \quad < if\ ((FIX0 == w[2]) \&\& \\ \quad\ wait(w[0]) > waitcnt(w[0])) > \} \cup \\ \#Statement\ 3 \\ \{w_k | w_k \in < w_n[0] \in neighbors(w[0])\ and \\ \ w_n[0] > w[0]\ and\ w_n[1] \longleftarrow w[0]\ and \\ \ w[2] \longleftarrow mis(w[0]) \\ \quad and\ w_n[2] = w[2]+1 >, \\ \ < mis(w[0]) \longleftarrow FIX1 >, \\ \quad < if\ ((FIX0 == w[2]) \&\& \\ \quad\ wait(w[0]) == (waitcnt(w[0])-1)) > \} \end{cases}$$

PROPOSITION 5. *MIS-FIX Algorithm is an instance of AGM where:*

(1) $G = (V, E, vmaps = \{wait, \rho\}, emaps = \{\})$ *is the input graph,*

(2) $WorkItems = WorkItems^{mis}$,

(3) $Q = \{mis, waitcnt\}$ *is the state mapping and initially* $\forall i \in V, mis(i) = UNFIX \,\&\, waitcnt(i) = \alpha$ *where* $\alpha = count(\{w | w \in neighbors(i) \,\&\, \rho(w) > \rho(i)\})$,

(4) $\pi = \pi^{mis}$,

(5) *Strict weak ordering relation* $<_{wis} = <_{fixbk}$,

(6) $S = \{<v, v, FIX1>\}$ *where* $v \in V$ *and* $\forall i \in neighbors(v),\ \rho(v) > \rho(i)$.

8.7. Summary

Maximal Independent Set (MIS) is a well-studied graph problem, and there are parallel algorithms to solve it. However, most of those algorithms show poor performance in distributed settings due to synchronous communication, subgraph construction, and random number generation.

In this chapter, we presented three MIS algorithms suitable for distributed memory parallel execution. The FIX algorithm is an asynchronous algorithm designed for distributed execution and FIX-Bucket and FIX-PQ algorithms make use of ordering to reduce computations. While FIX-Bucket performs ordering at a global level, the FIX-PQ algorithm performs ordering at the thread level to avoid global synchronization.

Weak scaling results on RMAT graphs show that FIX* algorithms outperform both vertex-centric Luby implementations as well as CombBLAS FilteredMIS algorithm. Weak scaling results and strong scaling results (except road network) show that the FIX-Bucket algorithm performs well for low-diameter graphs. For higher-diameter graphs FIX and FIX-PQ outperform other algorithms.

9

Orderings in Triangle Counting

Triangles are the most basic non-trivial subgraphs. Triangle counting is used in a number of different applications, including social network mining, cyber security, and spam detection. In general, triangle counting algorithms are readily parallelizable, but when implemented in distributed, shared-memory, their performance is poor due to high communication, imbalance of work, and the difficulty of exploiting locality available in shared memory. In this chapter, we discuss four different (but related) triangle counting algorithms and how their performance can be improved in distributed, shared-memory by reducing in-node load imbalance, improving cache utilization, minimizing network overhead, and minimizing algorithmic work. We generalize the four different triangle counting algorithms into a common framework and show that for all four algorithms the in-node load imbalance can be minimized while utilizing caches by partitioning work into blocks of vertices, the network overhead can be minimized by aggregation of blocks of work, and algorithm work can be reduced by partitioning vertex neighbors by degree.

We experimentally evaluate the weak and the strong scaling performance of the proposed algorithms with two types of synthetic graph inputs and three real-world graph inputs. We also compare the performance of our implementations with the distributed, shared-memory triangle counting algorithms available in PowerGraph-GraphLab and show that our proposed algorithms outperform those algorithms, both in terms of space and time.

9.1. Introduction

For a given graph $G = (V, E)$, triangle counting involves finding structures that have three vertices connected to each other by an edge. Triangle counting is used in many applications. For example, triangles are used to assess the content quality of social networks [**135**], to detect web spam [**13**] and to uncover thematic structures of the web [**36**]. With the increasing size of data sets, the graphs generated for those applications are growing larger and may not fit into a single machine memory.

Algorithm 14 Parallel Node Iterator Triangle Counting Algorithm

NodeIterator $G^{\text{local}} = (V, E)$:

```
1: for each v in V parallel do
2:   for each u ∈ pred(v) do
3:     for each w ∈ succ(v) do
4:       if u ∈ pred(w) then
5:         TC = TC + 1
6:       end if
7:     end for
8:   end for
9: end for
```

Hybrid, Single Program, Multiple Data (SPMD) approaches that represent distributed, shared-memory runtimes are a popular way to deal with these large graphs. *MPI+OpenMP* or *MPI+PThreads* are two such examples. When extending triangle counting for such distributed, shared-memory runtimes we face several challenges: 1. in-node load imbalance, 2. poor cache utilization, and, 3. higher message communication. Further, certain preprocessing optimizations applicable for shared-memory parallel triangle counting algorithms become cumbersome and inefficient to apply when the processing graph is distributed. In this chapter, we demonstrate step-by-step how we developed four related triangle counting algorithms that are scalable in distributed, shared-memory runtimes.

Most of the existing parallel triangle counting algorithms are variations of the sequential triangle counting algorithm: *Node-Iterator* [**121**]. Node-Iterator partitions neighbors of a vertex into two sets. The set $pred(v)$ is defined as the set of neighbors of v that are less than v and set $succ(v)$ is defined as the set of neighbors of v that are greater than v. The set

$pred(v)$ is called *predecessors* of v and the set $succ(v)$ is called *successors* of v. The parallel version of this algorithm (Listed in Algorithm 14) iterates over vertices and checks whether a vertex in predecessor set and a vertex in successor set makes an edge; if they do, the number of triangles is incremented.

A straightforward extension of this algorithm for distributed, shared-memory parallel execution, is to distribute vertices among different nodes and process every vertex in a separate parallel thread. This approach creates load imbalance between parallel threads when processing a skewed graph as threads that process hub vertices take more time compared to the threads that process vertices with fewer edges. In distributed execution, the algorithm needs to send each (v,u) pair to the node that owns vertex w to check whether u is in the predecessor set of w. When the number of distributed nodes is increased, the number of messages the algorithm needs to send also increases, as the successor vertex set of w can be distributed among the nodes. Therefore, when the number of nodes is increased algorithm spends more time on communication and less time on computation.

By processing every wedge (a possible triangle) in a separate parallel thread, algorithm can balance the load. However, processing every wedge in a separate parallel thread reduces the cache utilization and also increases the number of messages in distributed execution. A common optimization to reduce the number of processing wedges is to order vertices by their degree (e.g., [**109**]). Applying this optimization as it is for distributed, shared-memory triangle counting is challenging. Permuting vertex ids in a distributed setting requires sorting vertices by the degree, assigning new ids and exchanging those new ids with all other adjacencies in remote ranks and then, rebuilding the graph. This preprocessing step consumes time and is cumbersome to implement in a distributed setting.

In this chapter, we show how we handled each of the above discussed challenges. First, we show that Node-Iterator is one of four ways to do triangle counting and we generalize those four algorithms using two common abstractions (Section 9.2). All the previously discussed challenges are applicable to all four algorithms. To balance the overhead of load imbalance, cache utilization and to reduce the latency of each of the small messages, we

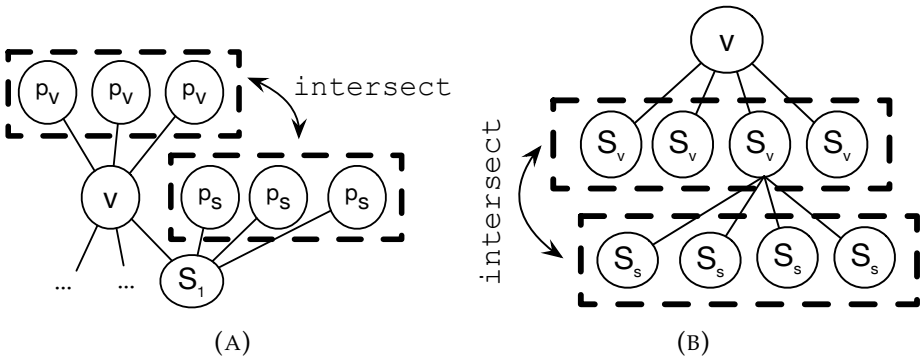
propose blocking neighbors – this is described in Section 9.4. Further, many real world graphs tend to have a small number of vertices with a high degree and a large number of vertices with a small degree. Therefore, we saw that latency incurred during distributed execution was still high with vertex blocks. Section 9.4.3 shows how latency of small blocks can be further reduced using block aggregation. Finally, we propose four triangle counting algorithms with two super-steps in each. In the first super-step, the algorithms partition neighbors by their degree, and in the second super-step the algorithms calculate the number of triangles. Partitioning neighbors by degree avoids the need for distributed sorting and graph re-construction.

Much of the existing work focuses on shared-memory, distributed-memory, or external memory. In this chapter we consider hybrid runtime models and the problems that one encounters when implementing triangle counting in such hybrid models. Some of the techniques we use are not new in separation (e.g., message aggregation), we use them in novel combinations to achieve better scaling. Selecting features and showing how to use them in combination to achieve better performing distributed, shared-memory parallel triangle counting algorithms is the main contribution of this chapter.

The performances of our proposed algorithms are evaluated on weak-scaling and strong-scaling (Section 9.6). For weak scaling we use two types of synthetic graphs and for strong scaling we use two types of synthetic graphs and three large real-world networks from [**79**]. Results show that the proposed algorithms scale well. In addition, we also compare our results with another distributed, shared-memory triangle counting implementation: PowerGraph-GraphLab [**55**] triangle counting algorithms and show that our algorithms outperform that triangle counting algorithms.

9.2. Triangle Counting

The parallel Node-Iterator algorithm discussed in Section 9.1 partitions neighbors of every vertex into two. To partition the neighbors, the algorithm uses vertex ids. For every vertex, v, the algorithm intersects the predecessor set of v with v's successor's predecessors (See Figure 9.1a). The size of the intersection is accumulated into the number of triangles.



(A) (B)

FIGURE 9.1. Set intersection in PSP and SSS algorithms.

We call this algorithm PSP. Mathematically, this triangle counting algorithm is expressed in Equation (1). In Equation (1), *TC* is the total number of triangles counted, and, as explained above, every vertex goes through its successors and intersects the vertex's predecessors with each successor's predecessors.

$$TC = \Sigma_{v \in V} \Sigma_{s \in succ(v)} |pred(v) \cap pred(s)| \tag{1}$$

The same number of triangles can be counted by switching *pred* with *succ* and *succ* with *pred* in Equation (1). In other words, instead of intersecting a vertex's predecessor set with successor's predecessor set we can intersect the vertex's successors with the predecessor's successor sets. We call this algorithm *successor, predecessor's successor* (SPS) and it is expressed in Equation (2).

$$TC = \Sigma_{v \in V} \Sigma_{p \in pred(v)} |succ(v) \cap succ(p)| \tag{2}$$

Another approach to count triangles is to intersect the successor set of a vertex with its successor's successors. Figure 9.1b depicts the algorithm. We call this algorithm SSS, and the mathematical equation for this algorithm is given in Equation (3).

$$TC = \Sigma_{v \in V} \Sigma_{s \in succ(v)} |succ(v) \cap succ(s)| \tag{3}$$

Algorithm	set_1	set_2	set_3
PSP	*succ*	*pred*	*pred*
SPS	*pred*	*succ*	*succ*
SSS	*succ*	*succ*	*succ*
PPP	*pred*	*pred*	*pred*

TABLE 9.1. set_1, set_2 and set_3 parameter values for each triangle counting algorithm.

In Equation (4), we switch succ with pred, as we did in Equation (1). The resulting equation is given in Equation (4). The algorithm represented in Equation (4) goes through predecessors of every vertex and intersects with predecessor's predecessor set (Named as *predecessor, predecessor's predecessor* (PPP)).

$$TC = \Sigma_{v \in V} \Sigma_{s \in pred(v)} |pred(v) \cap pred(s)| \tag{4}$$

An important observation of these algorithms is that they all involve three sets per each vertex. A generalized equation for triangle counting is given in Equation (5). The general form has a predecessor set or a successor set of the iterating vertex (set_1) to iterate, a predecessor set or a successor set of the iterating vertex (set_2), and a predecessor set or a successor set corresponding to a vertex in the iterating set (set_3). Table 9.1 summarizes the algorithms with appropriate values for set_1, set_2, and set_3.

$$TC = \Sigma_{v \in V} \Sigma_{s \in set_1(v)} |set_2(v) \cap set_3(s)| \tag{5}$$

9.3. Distributed, Shared-Memory Triangle Counting

As we saw in the previous section, the main operation in triangle counting is the intersection between two sets. In a distributed environment all the elements of these two sets may not reside in the same locality. In such situations we need to collect all the elements in both sets into one locality and perform the set intersection.

Algorithm 15 PSP Triangle Counting Algorithm

PSP $G^{\text{local}} = (V, E)$:

```
1: epoch {
2:   for each v in V in a parallel thread do
3:     for each s ∈ succ(v) do
4:       Send pred(v) & s to owner of s
5:     end for
6:   end for
7: }
```

Receive $pred(v), s$:

```
1: TC += pred(v) ∩ pred(s)
```

9.3.1. Graph Distribution & Data Structures.

Algorithm 16 Generalized Triangle Counting Algorithm

TC $G^{\text{local}} = (V, E)$:

```
1: epoch {
2:   for each v in V in a parallel thread do
3:     for each s ∈ set1(v) do
4:       Send set2(v) & s to owner of s
5:     end for
6:   end for
7: }
```

Receive $set_2(v), s$:

```
1: TC += set2(v) ∩ set3(s)
```

In our implementations, vertices are distributed equally among the participating nodes (*1D distribution*). A vertex id is represented using 64 bit integers. The first 48 bits represent the local vertex id and the second 16 bits represent the node id a vertex belongs to. Every rank contains a set of vertices and a set of edges corresponding to the vertices in the vertex set. Vertices and edges local to a rank are stored in *compressed sparse row* (CSR) format. This is shown in Figure 9.2. In Figure 9.2, every rank is separated using a vertical dash line and "Rn" represents rank n. We find the rank which a vertex belongs to by looking at the second 16 bits of the global vertex id. The CSR format consists of two arrays: 1. Indices array – this array stores the edges; and, 2. Indices pointer array – this array stores edge ranges belonging to every vertex. As an example, adjacencies for vertex v_k in rank R0 are found in the indices pointer array from position "a" to position "b".

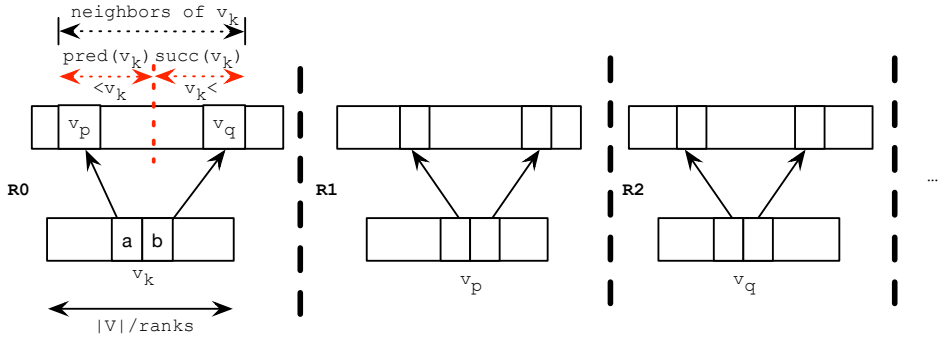


FIGURE 9.2. Separating successors and predecessors in CSR structure.

Assuming the indices range for every vertex is sorted, it is straightforward to calculate the predecessor set and successor set for every vertex. For every vertex, v_k, we need to find the position in the indices array that divides neighbors compared to v_k. Suppose this position is m, then range [a, m) denotes the predecessor set of v_k and the range [m, b) represents the successor set of v_k. In other words, vertices in range (a, m] in the indices array are less than v_k, and vertices in the range (m, b] in the indices array are greater than v_k (See R0 CSR in Figure 9.2). This way we limit the amount of additional space needed to store predecessor sets and successor sets to $O(|V|)$ (we only need to store the position m for every vertex).

However, many real-world graphs are unsorted. Still, when building the CSR structure we need to sort them by the source vertex (e.g., *Histogram Sort*). For unsorted graphs, this sorting step is extended and vertices are sorted by the source and then, by the destination. Note that this sorting step is local to a node and does not require any form of distributed communication; it takes place during the graph construction.

9.3.2. Challenges in Distributed Shared-Memory Parallel Triangle Counting. To construct an effective distributed, shared-memory parallel triangle counting algorithm, there were three primary problems we needed to solve: in our initial attempt to solve these challenges we came up with the PSP triangle counting algorithm listed in Algorithm 15. In this algorithm, every vertex processes in a separate parallel thread. An *epoch* (Line 1) is a code region that indicates explicit communication is taking place. Every vertex sends

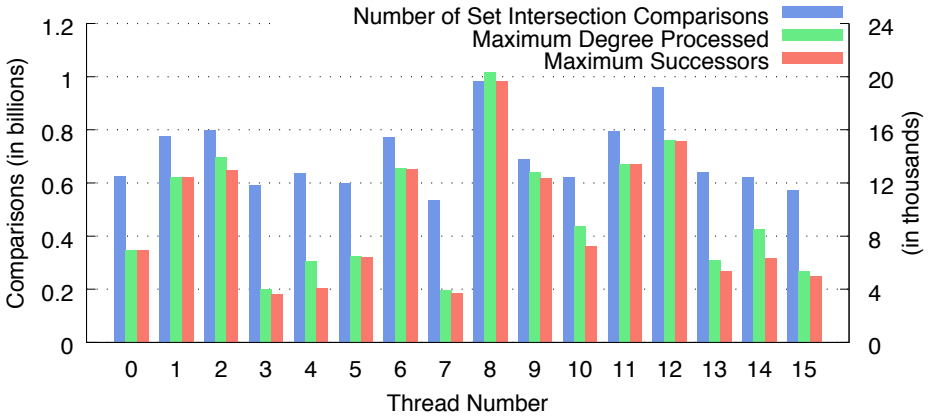


FIGURE 9.3. Number of comparisons performed in set intersection (left axis), and maximum vertex degree and number of successors (right axis) in each thread on LiveJournal social network graph with SSS triangle counting algorithm.

its predecessor set to each owner rank of its successors. Then, the rank that receives the predecessor set performs the intersection in a single thread.

By switching "pred" set with "succ" set and "succ" set with "pred" set in the above algorithm we get the SPS algorithm. In a similar way we can get the other algorithms. Algorithm 16 lists a generalized triangle counting algorithm and by replacing set_1, set_2 and set_3 according to values in Table 9.1, we get PSP, SPS, SSS and PPP. The performance of these initial implementations are poor because of:

(1) load imbalance between processing elements,

(2) redundant messages to the same rank.

For algorithms derived from Algorithm 16 (including Algorithm 15) we measured the work done by each parallel thread by counting the number of set comparisons performed by each thread. Figure 9.3 shows the set comparisons performed by each thread when processing LiveJournal social network graph **[85]** on 16 shared-memory parallel threads with the SSS algorithm. As can be seen in Figure 9.3, some threads are more utilized than others. Thread number seven performs the fewest set comparisons (i.e., 533529383) and thread number eight performs the most set comparisons.

Load imbalance is mainly caused by the uneven distribution of degrees on vertices. In Algorithm 16 every vertex is processed by a separated parallel thread. Figure 9.3 shows the

maximum degree and maximum number of successors of all the vertices processed by a thread in SSS algorithm. As can be seen in Figure 9.3 thread 8 has processed the vertex with highest degree (also the maximum number of successors) and thread seven has processed smaller degree vertices. Most of the real-world graphs are power-low graphs (i.e., they have few vertices that have a high number of neighbors and high number of vertices with fewer number of neighbors). It takes more time to perform set intersections on vertices with higher numbers of neighbors than vertices with fewer neighbors. Therefore, threads that process vertices with higher numbers of neighbors take more time than threads that process vertices with fewer numbers of neighbors.

One approach to overcome the load imbalance is to process each *open wedge* (a possible triangle) in a separate parallel thread. An open wedge is a triple : a predecessor of a vertex, a successor of a vertex and the vertex (See Figure 9.4). Now each thread processes an open wedge and the "Receive" function of Algorithm 15 needs to be modified to do a binary search on the successor's predecessor set instead of the set intersection. While this approach balances the work among parallel threads, it generates a lot of of messages. If a vertex has "n" number of predecessors and "m" number of successors, then this algorithm creates $n \times m$ number of messages per vertex. If we assume the average degree of a vertex is d, then this approach creates approximately $O(|V| \times d^2)$ messages. The algorithm also loses the ability to take advantage of better cache utilization due to binary search at the receiver's end. The accumulated runtime overhead and binary search to process the significant number of generated messages is quite high. Therefore, this approach does not show the advantage of load balance because the overhead of message communication and binary search is greater than the advantage achieved by load balancing.

9.4. Blocking and Grouping Vertices

To reduce the number of messages while balancing the load on processing threads we came up with a strategy to block vertices in predecessor sets and successor sets (the PSP algorithm). A block is a set of vertices. The block size is configurable. In the following we explain the blocking process in general for all the algorithms.

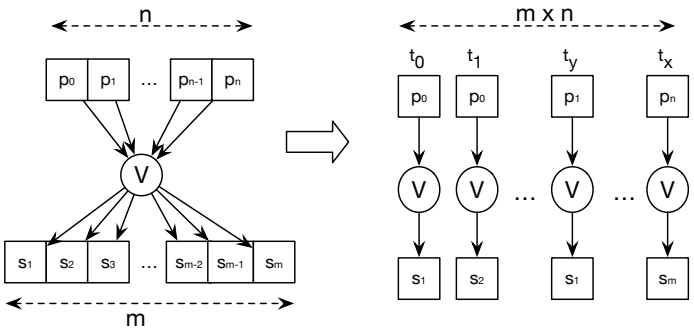


FIGURE 9.4. Every thread processes an open wedge

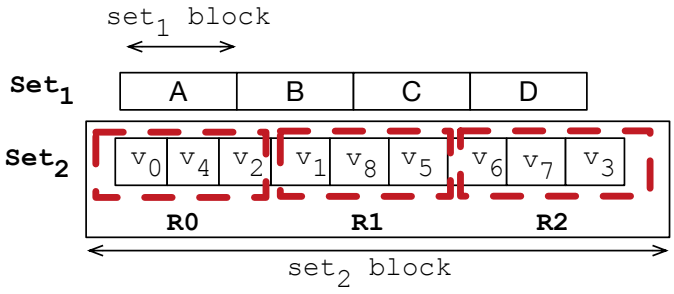


FIGURE 9.5. Blocking vertices in sets.

All the triangle counting algorithms discussed in Section 9.2 go through all the vertices in parallel and send one set of neighbors (set_1) to the owner of a vertex in an another neighbor set (set_2). For some algorithms (e.g., SSS and PPP) these two neighbor sets are the same. With blocking, instead of sending a complete chunk of neighbors to the owner of an another set, we send a subset, and this subset of vertices is called a "block". This approach reduces the number of messages relative to the approach discussed in Section 9.3.2.

Similar to set_1 we also block vertices in set_2. However, vertices in a set_2 block may not belong to the same rank. Therefore, in a set_2 block, vertices are *grouped* by their rank (See Figure 9.5 - suppose set_2 block size is 10). A distributed message will have a block from set_1 vertices, and subset of a block in set_2 vertices.

Every distributed message received by a rank is processed in a separate parallel shared-memory thread. Therefore, every distributed message will cause a thread to execute set intersections equal to the number of vertices in the group. The objective of blocking set_2 is to control the number of set intersections per thread.

The PSP algorithm with message blocking and grouping for one rank is listed in Algorithm 17. The code listed from Line 2 to Line 28 is executed for each shared-memory parallel thread. As discussed earlier, an *epoch* represents a region where the program can send/receive messages. Earlier algorithms (Algorithm 15 and Algorithm 16) process every vertex in a separate shared-memory parallel thread, but in this algorithm each set_1 block (*preblock* in Line 15) is processed in a separate parallel thread. For the PSP, algorithm, set_1 is the predecessor set of a vertex (shown in *pred(v)*) and set_2 is the successor set of a vertex (*succ(v)*). Every thread iterates over vertices (Line 3) and calculates an *offset* (Line 8). However, if a vertex does not have any predecessors or successors we do not need to proceed executing the rest of the algorithm; it will not generate any triangles (See condition in Line 4). The number of predecessor blocks and successor blocks is calculated based on predecessor block size and successor block size (Line 9 and Line 10). Then, the loop in Line 12 iterates over predecessor blocks starting from the offset. The next predecessor block in this "for" loop is selected after *nthreads* number of threads. Therefore, each thread processes a single predecessor block at a time. Once a thread selects a predecessor block, it extracts the vertices relevant to that block from pred(v) (Line 13–Line 15). The extracted predecessor block needs to be sent to every successor's rank. Line 16 iterates over successor blocks and extracts an appropriate successor block and stores in the variable *sucblock*. Instead of iterating over every vertex in this successor block, the algorithm, groups vertices in the successor block based on the vertex ownership. That is, vertices belonging to the same rank are grouped together. The function *group-by-rank* is responsible for grouping vertices based on their rank (See Line 20). The output of this function is an array where "i th" position in that array has the successor vertices for rank *i*. Then, the algorithm iterates over all rank ids and sends predecessor block and the respective successor block encapsulated in a message to the respective rank (Line 21 – Line 24).

The receiving rank gets a predecessor block and a set of successor vertices (See *Receive*, Line 2–Line 5). The receiving rank then goes through all vertices in the successor sets and for each successor set it extracts the set of vertices that are greater than or equal to the first element of the predecessor set (*Receive*, Line 3). The first element in the predecessor set

Algorithm	D1 Hit/Miss(%)	D2 Hit/Miss(%)
Basic SSS	95.0/5.0	55.0/45.0
Blocked SSS, block = 100	96.0/4.0	69.8/30.2

TABLE 9.2. Cache utilization of SSS vs. blocked SSS.

is a lower bound for this new set. Then, the algorithm performs an intersection between the predecessors and the calculated lower bound set to determine the number of triangles (*Receive*, Line 4).

Calculating the lower bound set allows us to save some comparison operations in set intersection. Calculating an upper bound is not necessary since the set intersection comparison starts from the beginning of the sets; when set intersection reaches the end of the smaller set, it terminates the set intersection operation.

The logic for the generalized blocked triangle counting algorithm is quite similar to Algorithm 17, and, presented in Algorithm 18. In the generalized version of the algorithm, the predecessor set of a vertex is replaced with set_1 and the successor set of a vertex is replaced with set_2. For some algorithms both set_1 and set_2 are the same (e.g., SSS). For those algorithms the logic is simpler because we iterate and send blocks to owners of vertices in the same set. For example, for SSS we can remove the first condition from Algorithm 17, Line 4, and we do not need to calculate *nsetbblks* as its value is the same as *nsetablks*.

9.4.1. Block Size in Shared-Memory. In shared-memory execution, smaller block sizes (set_1 block size and also set_2 block size) reduce the load imbalance between parallel threads. Figure 9.6 shows the number of comparisons performed by each thread for set intersection in the SSS algorithm with LiveJournel graph input. Compared to Figure 9.3, Figure 9.6 shows much better balancing of work among different threads. For example, in SSS implementation (Algorithm 16, with $set_1 = set_2 = set_3 = succ$) a thread processes a vertex irrespective of the number of neighbors that vertex has. In the blocked implementation (Algorithm 18), a vertex with a higher number of neighbors is processed by more than one thread (depending upon the size of the block). Therefore, with blocking work can

Algorithm 17 PSP Triangle Counting Algorithm with Blocking & Grouping

PSP $G^{\text{local}} = (V, E), predblksz, sucblksz, nthreads$:

```
1:  for each parallel thread : tid in nthreads do
2:    epoch {
3:      for each v in V do
4:        if (|pred(v)| == 0) or (|succ(v)| == 0) then
5:          continue;
6:        end if
7:
8:        offset = (v + tid) % nthreads;
9:        npredblks = (|pred(v)| + predblksz − 1) / predblksz
10:       nsucblks = (|succ(v)| + sucblksz − 1) / sucblksz
11:
12:       for (pos = offset; pos < npredblks; (pos += nthreads)) do
13:         predstart = pos * predblksz
14:         predend = min(predblksz, (|pred(v)|-predstart)) + predstart
15:         preblock = {x|x ∈ pred(v) & (predstart ≤ indexof(x) < predend)}
16:         for (sucpos = 0; sucpos < nsucblks; sucpos++) do
17:           sucstart = sucpos * sucblksz
18:           sucend = min(sucblksz, (|succ(v)|-sucstart)) + sucstart
19:           sucblock = {x|x ∈ succ(v) & (sucstart ≤ indexof(x) < sucend)}
20:           rankgroups = group-by-rank(sucblock)
21:           for each r in ranks do
22:             sucblckforrank = rankgroups[r]
23:             Send(r, preblock, sucblckforrank)
24:           end for
25:         end for
26:       end for
27:     end for
28:     }
29: end for
```

Receive $preblock, sucblckforrank$:

```
1: p = preblock[0]
2: for each s in sucblckforrank do
3:   lowerbound = {x|x ∈ pred(s) and p ≤ x}
4:   TC += |preblock ∩ lowerbound|
5: end for
```

be equally distributed among threads irrespective of the degree distribution of the input graph.

The proposed algorithms (Algorithm 18), block the predecessor set (set_1) of a vertex and the successor set (set_2) of a vertex. In the basic PSP triangle counting algorithm, a predecessor set of a vertex is sent to each of its successors. By blocking and grouping

Algorithm 18 Generalized Triangle Counting Algorithm with Blocking & Grouping

TC $G^{\text{local}} = (V, E), setablksz, setbblksz, nthreads$:

```
1:  for each parallel thread : tid in nthreads do
2:    epoch {
3:      for each v in V do
4:        if (|set1(v)| == 0) or (|set2(v)| == 0) then
5:          continue;
6:        end if
7:
8:        offset = (v + tid) % nthreads;
9:        nsetablks = (|set1(v)| + setablksz − 1) / setablksz
10:       nsetbblks = (|set2(v)| + setbblksz − 1) / setbblksz
11:
12:       for (pos = offset; pos < nsetablks; (pos += nthreads)) do
13:         setastart = pos * setablksz
14:         setaend = min(setablksz, (|set1(v)|-setastart)) + setastart
15:         preblock = {x|x ∈ set1(v) & (setastart ≤ indexof(x) < setaend)}
16:         for (setbpos = 0; setbpos < nsetbblks; setbpos++) do
17:           setbstart = setbpos * setbblksz
18:           setbend = min(setbblksz, (|set2(v)|-setbstart)) + setbstart
19:           setbblock = {x|x ∈ set2(v) & (setbstart ≤ indexof(x) < setbend)}
20:           rankgroups = group-by-rank(setbblock)
21:           for each r in ranks do
22:             setbblckforrank = rankgroups[r]
23:             Send(r, preblock, setbblckforrank)
24:           end for
25:         end for
26:       end for
27:     end for
28:   }
29: end for
```

Receive $setablock, setbblckforrank$:

```
1: p = setablock[0]
2: for each s in setbblckforrank do
3:   lowerbound = {x|x ∈ set1(s) and p ≤ x}
4:   TC += |setablock ∩ lowerbound|
5: end for
```

successors of a vertex, (set_2), we send a predecessor block together with multiple successor vertices to the same rank (Using PSP as an example). Then, the receiving rank can reuse the predecessor block to do set intersection with several predecessor sets of the successor vertices. This approach achieves better cache utilization because the predecessor block is re-used. This applies to all algorithms. Table 9.2 compares the cache utilization of the basic

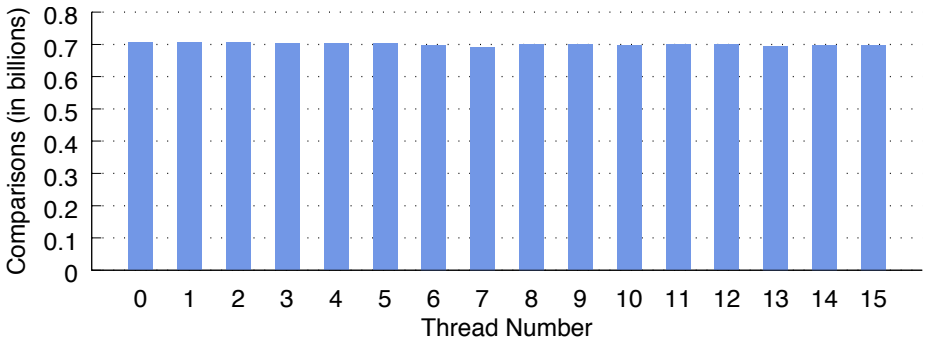


FIGURE 9.6. Number of comparisons performed in set intersection by each shared memory parallel thread, on LiveJournal social network graph with SS triangle counting algorithm. set_1 block size = set_2 block size = 100

Block Size	D1 Hit/Miss(%)	D2 Hit/Miss(%)
10	87.1/12.9	53.7/46.3
80	93.7/6.3	54.9/45.1
340	96.2/3.8	59.1/40.9

TABLE 9.3. Cache utilization of SSS algorithms with increasing block size. Values are average across the number of threads.

Block Size	Bytes Communicated
10000	16842970024
1000	17927870248
100	29910155304

TABLE 9.4. Block size vs. Number of bytes communicated. SPS algorithm with Graph500, Scale 23 graph input and experiments run on four nodes.

SSS algorithm with the blocked SSS algorithm. As can be seen in Table 9.2, the blocked SSS algorithm gets better utilization of level 2 data (D2) cache (same behaviour is seen in other algorithms). This is mainly because of the for loop in "Receive" functions. However, increasing successor block size would cause load imbalance as the number of vertices loop in increases. Therefore, by adjusting the successor (or set_2) block size we can achieve better cache utilization while minimizing the load imbalance.

Smaller block sizes achieve better balancing of work among parallel threads. However, smaller block sizes tend to reduce the cache utilization. Table 9.3 shows the cache utilization of the SSS algorithm with increasing block sizes (both set_1 and set_2 block sizes are set to value in the table). As can be seen in Table 9.3, when we increase the block size we see better cache utilization.

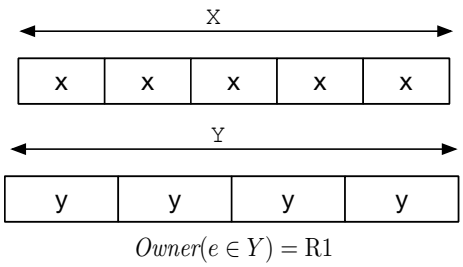


FIGURE 9.7. Block example.

9.4.2. Block Size in Distributed-Memory. In distributed execution, smaller block sizes generate more messages. To explain this better, consider the example in Figure 9.7. In this example, we are sending set X to the owner of Y. Suppose all the elements of Y belong to the same owner. In a basic algorithm (without blocking) we would send $X+Y$ number of vertices, but in the blocked version we need to send each x size block with each y size block. Not only we are sending $\lceil X/x \rceil * \lceil Y/y \rceil$ number of messages, but also smaller block sizes increase the number of bytes transferred over the network. Therefore, as per Figure 9.7, the algorithm needs to send each x block with all y blocks; this needs to repeat for each y block. Therefore, the total number vertices transferred is $(Y + x \times Y/y) \times X/x = XY(1/x + 1/y)$. As per the equation, when we decrease the block size, (x or y), the number of vertices transferred increases. Table 9.4 experimentally shows how the number of bytes communicated over the network increases with the small block sizes.

While vertex blocking and grouping helps to alleviate the load imbalance in shared memory and reduces the number of messages in distributed memory we still see quite a significant number of messages transferred over the network. In Power-law graphs we see fewer high-degree vertices and a larger number of low-degree vertices. There will be at least one message send call per each of the lower-degree vertices. Therefore, algorithm execution still generates a large number of messages when processing a power-low graph. To further reduce the latency overhead incurred by small messages, our next step was to introduce *block aggregation*.

9.4.3. Block Aggregation. The overhead of sending each of the small messages (less than the size of a block) to a destination is high compared to sending a large message with several of those smaller messages. To reduce the network overhead of sending small messages we use block aggregation.

The size of a block is not fixed and depends on the block size as well as the degree distribution of vertices. To aggregate blocks, we maintain a buffer of bytes per each destination rank. In the send call of an algorithm (e.g., Line 23 in Algorithm 18), block is not directly sent to the destination rank, rather it is stored in a buffer relevant for the intended destination rank. The maximum block size is configurable and specified in the number of bytes.

When a buffer receives a message that cannot fit into the buffer without exceeding the maximum buffer size, the algorithm sends the buffer to the destination and then the buffer is flushed. Parallel shared memory threads store blocks in destination rank buffers. Concurrent access to the buffer is achieved using atomic operations. When a thread attempts to add a block to a destination buffer while the buffer is being sent to the destination, the thread is suspended until a send operation is called.

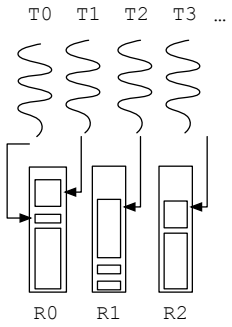


FIGURE 9.8. Block aggregation for four destination ranks. Threads add blocks to different destination buffers.

Further, we use non-blocking send-receive operations in our code (i.e., MPI_Isend & MPI_Irecv). Therefore, the time a thread has to wait until an aggregated buffer is transferred is minimized. Figure 9.8 shows the block aggregation of a triangle counting algorithm running in four ranks. The picture shows the block aggregation in fourth rank and it maintains three outgoing buffers: one per each remote rank. Threads write to buffers as blocks are generated.

Block aggregation reduces the latency overhead of sending large numbers of small messages. We saw a performance benefit with block aggregation even at very small graph scales. For example, Graph500 scale 13 graph on four ranks takes 7.01 seconds to run "blocked PSP" algorithm and takes 29 seconds to run "blocked SSS" algorithm. With block aggregation both algorithms run in a less than a second.

9.5. Degree based Partitioning

The performance of distributed, shared-memory parallel triangle counting algorithms can be further improved by reducing the number of bytes transferred over the network. For example, in PSP algorithm, the number of bytes transferred over the network can be reduced by reducing the size of the predecessor set, because the algorithm sends the same predecessor set to multiple ranks (where successor groups are located). To reduce the size of the *duplicating set* (e.g., predecessor set in PSP) to every rank, we use a degree based neighbor partitioning scheme.

Traditionally, triangle counting algorithms partition neighbors using lexicographical comparison. In the proposed approach we partitioned neighbors based on the degree of the vertices. For the PSP algorithm, if a vertex neighbor's degree is higher than the vertex's degree, then we add the neighbor to predecessor set. If the neighbor's degree is less than the vertex's degree, we add that to the vertex's successor set. If the neighbor's degree is equal to the vertex's degree we perform a lexicographical comparison on vertex ids and add the neighbor to either to the successor set or the predecessor set.

The degree partitioned PSP algorithm is listed in Algorithm 19. This algorithm consists of two super-steps. In the first super-step (Line 1–Line 8), the algorithm iterates over all the vertices and exchanges the degrees of the vertices. The *SendDegree* (Line 5) function will call the *ReceiveDegree* (Algorithm 19) function in the same rank or in a different rank. The SendDegree call sends local vertex v and v's degree to its neighbor u. When the neighbor(u) receives the degree and the vertex, it first checks whether the received degree is less than its degree; if so, the received vertex is added as a successor of u (See Line 2 in *ReceiveDegree*); otherwise v is added as a predecessor of u (*ReceiveDegree*, Line 4). If degrees are equal, then we compare vertex ids to decide whether a neighbor is a successor or predecessor (*ReceiveDegree*, Line 6–Line 10).

With degree partitioning we cannot use the CSR data structure to extract the predecessor set and successor set for a vertex (as in Figure 9.2, Section 9.3). Therefore, to collect the predecessor set and successor set for each vertex a concurrent data structure (*an append*

buffer) is used. Two instances of the concurrent data structure is maintained per every local vertex. The first instance stores the predecessors and the second instance stores the successors. In the first super-step of the algorithm, append buffers for a vertex maybe modified by more than one parallel thread. However, after first super-step we do not need to modify the append buffer concurrently. Even though we partition vertex neighbors based on degrees, our set intersection comparison is based on vertex identifiers (an additional conversion function in set intersection reduces the performance as it tends to reduce cache utilization). Therefore, after calculating predecessors and successors per each vertex, they are locally sorted (See Line 9–Line 12).

Algorithm 19 Degree Partitioned PSP Algorithm

PSP $G^{\text{local}} = (V, E), predblksz, sucblksz, nthreads$:

```
1:  epoch {
2:    for each v ∈ V in a parallel thread do
3:      for each u ∈ neighbors(v) do
4:        d_v = degree(v)
5:        SendDegree(u, v, d_v)
6:      end for
7:    end for
8:  }
9:  for each v ∈ V in a parallel thread do
10:   sort(pred(v))
11:   sort(succ(v))
12: end for
13: ...
14: Rest of code is same as Algorithm 17, PSP (Line 1–Line 29).
```

ReceiveDegree u, v, d_v:

```
1:  if d_v < degree(u) then
2:    succ(u).insert(v)
3:  else if d_v > degree(u) then
4:    pred(u).insert(v)
5:  else
6:    if u > v then
7:      succ(u).insert(v)
8:    else
9:      pred(u).insert(v)
10:   end if
11: end if
```

Receive $preblock, sucblckforrank$:

```
1:  Same as code in Algorithm 17, Receive (Line 1–Line 5).
```

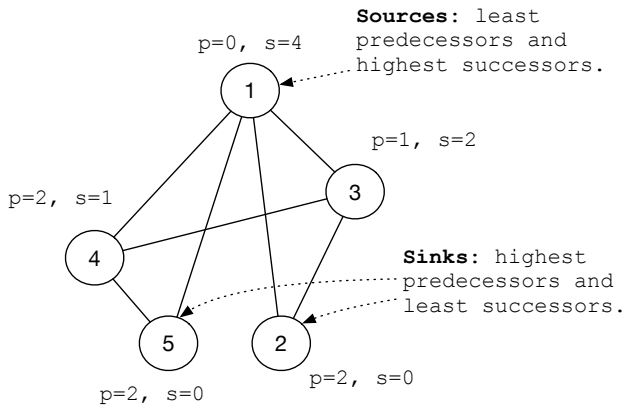


FIGURE 9.9. An example DAG and predecessors and successors counts.

With the predecessor-successor relationship we can depict the whole graph as a DAG (Figure 9.9). Algorithm 19 makes sure that a predecessor of a given vertex has a higher degree than its own degree. This way, Algorithm 19 makes the highest degree vertices sources of the induced DAG. Also, a vertex has fewer number of predecessors than its successor. In other words, sources have higher number successors, but they have zero predecessors, and sinks of the induced DAG have the least number of successors and highest number of predecessors. Since the partitioning scheme minimizes the number of predecessors, it is favourable for the PSP algorithm. For SPS and SSS algorithms, comparisons in the "ReceiveDegree" function must be switched.

9.6. Results

The proposed changes (vertex blocking, block aggregration and neighbor partitioning based on degree) to distributed-memory parallel triangle counting algorithms (PSP, SPS, SSS, PPP) improves the performance of distributed memory parallel triangle counting and minimizes the pre-processing overhead. The remainder of this section will present and explain experimental results to demonstrate this claim.

We ran our experiments on a Cray XC system that has 2 Broadwell 22-core Intel Xeon processors. Our experiments only used up to 16 cores to uniformly to double the problem size and to double the number of processors in weak scaling. Each node consists of 128 GB

Graph	Vertices	Edges
Friendster	6.83E+07	2.59E+09
Twitter	4.17E+07	1.47E+09
Orkut	3.07E+06	1.17E+08
RMAT-1(25)(rmat1)	3.36E+07	5.37E+08
RMAT-2(25)(rmat2)	3.36E+07	5.37E+08

TABLE 9.5. Graph inputs and their attributes used in strong scaling experiments

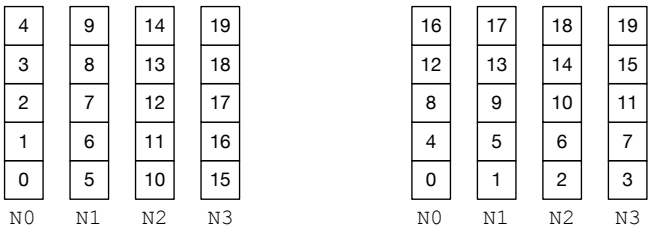
DDR4-2400 memory. We use a MPI+PThread, distributed shared-memory runtime. The MPI implementation is Cray MPICH (version 7.4.4).

We evaluate the triangle counting algorithms in terms of *strong scaling* and *weak scaling*. For weak scaling experiments, we use *R-MAT* **[23]** synthetic graphs. Two types of RMAT synthetic graphs are used. They are: 1.RMAT-1: Graphs based on the current Graph500 **[104]** Breadth First Search benchmark specification with R-MAT parameters $A = 0.57$, $B = C = 0.19$ and $D = 0.05$, and, 2.RMAT-2: Graphs generated based on the proposed Graph500 **[56]** SSSP benchmark specification with R-MAT parameters $A = 0.50$, $B = C = 0.1$ and $D = 0.3$.

In the following sections, the algorithm names starting with "Opt-" are the algorithms that partition neighbors based on vertex degrees (similar to Algorithm 19). Algorithms that do not have the "Opt-" prefix partition neighbors by vertex ids (similar to Algorithm 17).

9.6.1. Graph Data Distribution Selection. As stated in Section 9.3.1 we used 1D distribution to distribute vertices among processes. In 1D distribution vertex identifiers can be permuted in several ways. We first experimented with two types of permutations: 1. assign contiguous numbers of vertex identifiers to each node (*1D block distribution*); and, 2. assign vertex identifiers to nodes in round-robin fashion (*1D cyclic distribution*). These two approaches are depicted in Figure 9.10.

Our initial results showed that the distributed load imbalance is higher when we use the 1D block distribution (Figure 9.10a) compared to 1D cyclic distribution (Figure 9.10b). Figure 9.11 shows the number of comparisons performed in set intersection on a rank when running PPP and Opt-PPP algorithms (on eight ranks). When PPP algorithm is run



(A) (B)

FIGURE 9.10. 1D block distribution and 1D cyclic distribution. "Ni" is rank id.

with block distribution, the number of comparisons performed by each rank is significantly different. However, when we use cyclic distribution, the number of comparisons are not skewed as with block distribution.

The triangle counting algorithms discussed in this chapter use vertex ids to partition neighbors into successors and predecessors, and one of these sets is sent to the owner nodes of the vertices in the other set. For example, PSP (Algorithm 17) sends the predecessor set of a vertex to its successor's owner's nodes. Since block distribution stores a contiguous number of vertices in a node, more messages are sent to nodes that store greater vertex ids. If we use the block cyclic distribution, messages are not centered on a particular node, but rather, work is divided among participating nodes.

The distributed load imbalance is minimized in optimized algorithms (like Algorithm 19), when they are used with the cyclic distribution. These optimized algorithms push higher-degree vertices to a corner of the DAG, and distribute the set intersections equally, significantly reducing the load imbalance. Figure 9.11 shows the set comparisons performed on each rank for Opt-PPP algorithm and as can be seen in the plot, the number of comparisons are almost evenly distributed. Therefore, we used 1D cyclic distribution in all our experiments.

9.6.2. Weak Scaling Results. Weak scaling results for triangle counting algorithms are shown in Figure 9.12. Triangle counting is *not* a linear time ($O(N)$) algorithm (See **[80]** for details). Therefore, we do not see constant time scaling, independent of the number of processors for algorithms that partition neighbors by vertex ids (i.e., PSP, PPP, SPS, SSS). However, degree partitioned algorithms show much better scaling behavior.

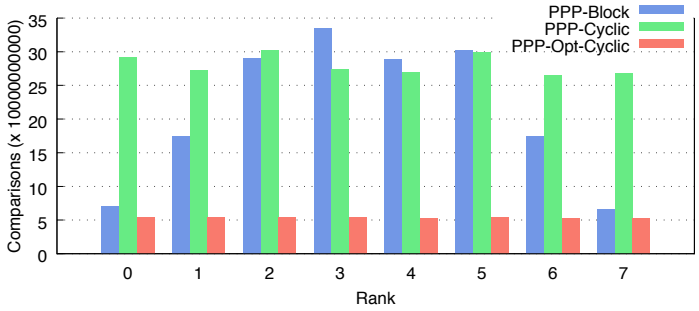


FIGURE 9.11. Total set comparisons performed on ranks for PPP algorithm with block and cyclic distributions, Opt-PPP algorithm with cyclic distribution. Input graph : RMAT-1, Scale 24. Threads per rank is 16.

Execution	Algorithm	Time (sec.)	Set Comparisons	Bytes
In-node	PPP	72.68	6.43E+11	–
	Opt-PPP	31.66	1.70E+11	–
Distributed	PPP	21.03	8.11E+11	9.01E+09
	Opt-PPP	8.20	1.70E+11	3.86E+09

TABLE 9.6. Number of set-comparisons and network bytes transferred in PPP and Opt-PPP for Scale 23 Graph500 input.

Table 9.6 shows the total number of set comparisons in PPP and Opt-PPP algorithms for scale 23, Graph500 input. Shared-memory data is for execution with 16 parallel threads, and distributed execution is for four rank each executing 16 parallel threads. As can be seen in the table, the number of set comparisons for normal PPP algorithms is nearly six times higher than the number of set comparisons for Opt-PPP algorithm. For distributed execution, the number of bytes transferred by the normal PPP algorithm is nearly 2.5 times higher than degree based neighbor partitioned PPP. Also, note that there is not much difference in the number of set comparisons between shared memory execution and distributed memory execution for algorithm Opt-PPP. On the other hand, in PPP algorithms the distributed, shared-memory total set comparisons is higher than the total set comparisons for shared memory.

The degree-based neighbor partitioning algorithm pushes high degree vertices into a corner of the DAG (See Figure 9.9), minimizes the number of set intersections, and minimizes the amount of data to be transferred. Also, we do not see much difference in the number of set comparisons for shared memory and distributed memory because the sets

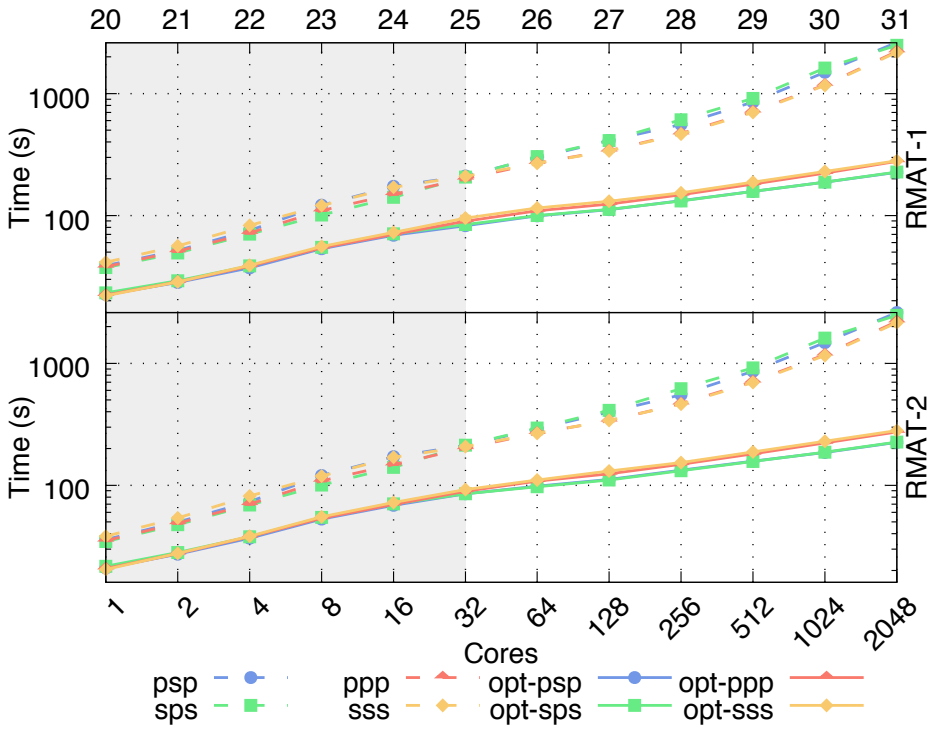


FIGURE 9.12. Optimized and non-optimized triangle counting algorithms weak scaling results for RMAT-1 and RMAT-2 graphs. Shaded region shows the shared memory execution.

(set_1 and set_2, e.g., *pred* for PPP algorithm) are small. Sets are small for Opt-PPP because of the degree partitioning. When small set sizes are closer to the block size(s) a minimum amount of data are duplicated (See the discussion in Section 9.4.2). The PPP algorithm predecessor set sizes are greater than block size, and because of that the same block processed multiple times. Therefore, in distributed settings the total number of set comparisons increases for normal algorithms (i.e., algorithms without "Opt-").

9.6.3. Strong Scaling Results. For strong scaling experiments, we ran degree partitioned triangle counting algorithms on graphs listed in table 9.5 over 1–1024 cores. To gain a better understanding about how algorithms scale relative to each other, we measured Relative Speedup, $= \frac{T_{ref\_1}}{T_n}$ i.e., the ratio of the execution time of the fastest sequential algorithm, T_{ref_1} and the parallel execution time on n processing elements, T_n.

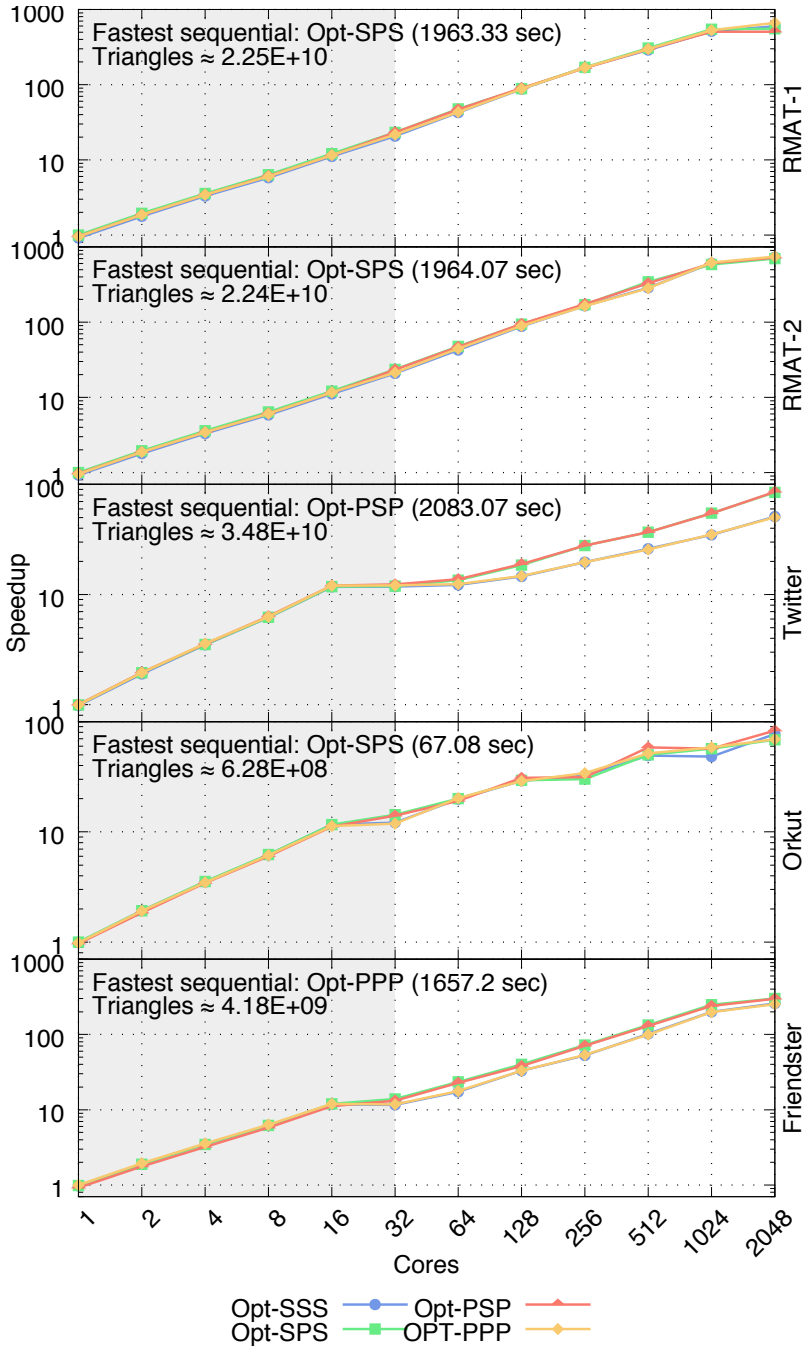


FIGURE 9.13. Strong scaling results.

Figure 9.13 shows strong scaling results for graphs listed in Table 9.5. As shown in Figure 9.13 all the graphs show better strong scaling behaviours for degree partitioned algorithms. We see that for the Twitter graph the Opt-PSP and Opt-SPS performs better than

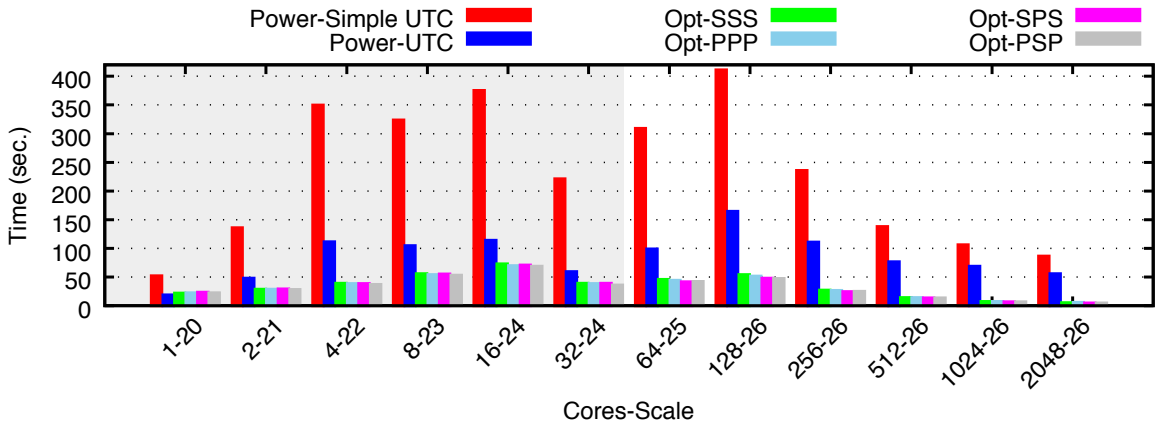


FIGURE 9.14. Comparison with PowerGraph-GraphLab.

Opt-PPP and Opt-SSS. For the Twitter graph, both Opt-PPP and Opt-SSS algorithms send more bytes over the network than Opt-PSP and Opt-SPS (e.g., on 64 nodes Opt-PPP sends 3.07E+11 bytes and Opt-PSP sends 2.45E+11 bytes). Therefore, we see that for Twitter graph Opt-SPS and Opt-PSP, algorithms are slightly better than Opt-PPP and Opt-SSS.

9.6.4. A Comparison with PowerGraph. PowerGraph [**55**] implements two undirected triangle counting algorithms: *simple_undirected_triangle_counting* (Power-Simple UTC), and, and *undirected_triangle_counting* (Power UTC). Our initial intention was to perform weak scaling for these two triangle counting programs with Graph500 input. However, we observed that both of the triangle counting applications fail with an out of memory error at Graph500 scale 27 and higher in distributed execution. Therefore, we compare the performance of PowerGraph triangle counting algorithms with degree partitioned triangle counting algorithms with the maximum scale PowerGraph can run in distributed execution for a given number of nodes.

Figure 9.14 shows the comparison results. Overall, the PowerGraph undirected triangle counting algorithm performs better than the PowerGraph simple undirected triangle counting algorithm. We see that the undirected triangle counting algorithm shows better performance than the degree partitioned triangle counting algorithm in sequential execution, but in all other executions the degree partitioned algorithms outperform PowerGraph triangle counting algorithms. The performance difference between PowerGraph

triangle counting algorithms and degree partitioned triangle counting algorithms is higher at larger scales.

PowerGraph uses Gather-Apply-Scatter (GAS) primitive to perform computations. The undirected triangle counting algorithm (Power UTC) performs better than "Power-Simple UTC" algorithm. According to the logic implemented, the Power-UTC is a hash set version of the triangle counting algorithm in [**121**]. The implementation maintains a list of all of its neighbors in a hashed set. For each edge (u, v) in the graph algorithm counts the number of set intersections of the neighbor set on u and neighbor set on v and stores the number of intersections on each edge. This algorithm counts each triangle three times.

Algorithms proposed in this chapter count triangles only once. Therefore, the number of set intersections are low compared to Power-UTC implementation. Further, Power-UTC shows poor weak-scaling behavior. Also, Power-UTC implementation does not perform operations to reduce the message latency (e.g., aggregation or blocking) and load imbalance.

9.7. Summary

Triangle counting algorithm performance on hybrid runtimes tends to suffer due to in-node load imbalance, high message communication and poor cache utilization.

In this chapter, we showed that different triangle counting algorithms can reduce in-node load imbalance and improve cache utilization by blocking vertices. For distributed execution the block aggregation alleviates the overhead of sending many small messages to a destination. To further reduce the number of set comparisons in set intersection and to reduce the amount of remote communication we presented degree-based vertex neighbor partitioning approach for triangle counting algorithms that consists of two super-steps. In Section 9.2, we showed that different triangle counting algorithms can be modeled using a generic framework, and showed how these techniques are applicable to the generic model and hence to all triangle counting algorithms presented in Section 9.2.

The performance results show that the presented algorithms scale well with the problem size as well as with the number of parallel processors. Further, the comparison with

PowerGraph triangle counting algorithms demonstrate that presented algorithms outperform PowerGraph triangle counting algorithms both in terms of time and space.

10

Runtime API for AGM

AGM abstractly models orderings in asynchronous parallel graph algorithms. An implementation of the AGM model has to interface with a parallel (distributed) runtime. In this chapter, we discuss how an AGM/EAGM framework can interface with a distributed-memory parallel runtime. We also outline the functionality the framework requires from the underlying runtime as an Application Programming Interface (API).

In addition to the API we also discuss some of the important runtime parameters and design choices that may affect the performance of AGM/EAGM algorithms. Distributed message communication with or without *message aggregation*, *thread allocation for communication and computation*, are examples of important design choices and max messages to aggregate. Flow control value is also an important parameter that affect the performance of AGM/EAGM algorithms.

10.1. The Runtime

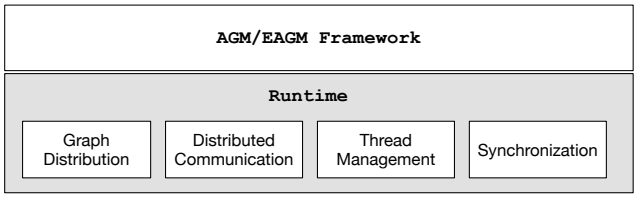


FIGURE 10.1. Layered design of the AGM framework on top of a runtime.

The AGM framework assumes an existence of a distributed-memory parallel runtime. The framework is built on top of this runtime. The interaction between the runtime and the

AGM framework is depicted in Figure 10.1. The runtime provides the following services to the AGM framework:

(1) Graph data distribution
(2) Distributed communication
(3) Thread management: including the management of Non-Uniform Memory Access (NUMA) nodes
(4) Synchronization
(5) Termination

10.1.1. Graph data distribution. In a practical execution environment, the *workitems* generated by a processing function in one node need to be traversed to a processing function in another node. The runtime is responsible for transporting *workitems* in one node to another node.

The runtime decides which node the *workitem* should be transported to, based on a *distribution*. There are two main strategies to distribute graph data. They are:

(1) 1D distribution – distribute vertices across participating nodes
(2) 2D distribution – distribute edges among participating nodes

We find further variations of each distribution, e.g., 1D distribution can be further classified into the following distributions:

(1) 1D-block distribution
(2) 1D-cyclic distribution
(3) 1D-random distribution

(A) 1D-Block distribution.

(B) 1D-Cycle distribution.

(C) 1D-Random distribution.

FIGURE 10.2. Comparison of different 1-D graph data distributions.

The *1D-block* distribution divides total vertices equally among processes and allocates a contiguous range of vertex numbers to each process (Figure 10.2a). The *1D-cycle* distribution also distributes vertices equally among participating nodes but allocates vertex IDs to nodes in a cycle. For example, in a two node system, 1D cycle distribution allocates vertex 0 to node0 allocate vertex 1 to node1, then again assigns vertex 2 to node0 and assigns vertex 3 to node1 etc. This is shown in Figure 10.2b. The 1D-Random distribution allocates vertex IDs to nodes based on a random distribution (Figure 10.2c).

These different graph distributions impact performance differently. These impacts are discussed in future chapters. As far as the AGM framework is concerned, the runtime provides an interface similar to the following:

LISTING 10.1. The *workitem* routing code.

```
1 struct routing_function {
2   template<typename work_item>
3   int operator(const work_item& wi)() {
4     //calculates the node id for wi
5   }
```

```
6 };
```

.

In Listing 10.1 the *routing_function* calculates the node id the *workitem* should be routed to. The logic for *routing_function* depends on the specific distribution used in the runtime.

10.1.2. Distributed Communication. Based on the graph data distribution, the runtime is responsible for sending *workitems* to the appropriate destinations. Put simply, each *workitem* is wrapped into a message and sent to its destination.

10.1.2.1. *Message Aggregation.* The runtime has the option of performing optimizations such as collecting number of *workitems* into a single large message that is sent to a destination. This process reduces the latency overhead of sending every single message over the network. This process of collecting multiple *workitems* into a single large message is called *message aggregation*.

For aggregation, a node maintains a buffer of *workitems* for every destination. A buffer can be updated by several parallel and concurrent threads (Figure 10.3). In Figure 10.3, the node *N0* has buffers for nodes *N1, N2, N3* and these buffers are concurrently updated by thread *T0, T1, T2, and T3*. The concurrent updates can be implemented using either atomic operations or locks. The runtime that is used to implement the AGM framework described in this thesis uses atomic operations to handle concurrency.

How multiple *workitems* aggregate is defined using a configuration. When buffers reach the configured value, the whole buffer is wrapped into a message and sent over the network. A buffer may also be partially sent (before reaching the configured number of *workitems*) if the algorithm runs out of sufficient *workitems* to process.

However, we cannot use this exact strategy when the size of a *workitem* is not fixed. For example, in triangle counting (Chapter 9) we will use *workitems* of variable sizes. When the *workitem* size is variable, we must use a specific number of bytes to aggregate. Likewise, when a buffer reaches that amount of bytes, the buffer is sent to the destination. More details about this aggregation method are discussed in Section 9.4.3.

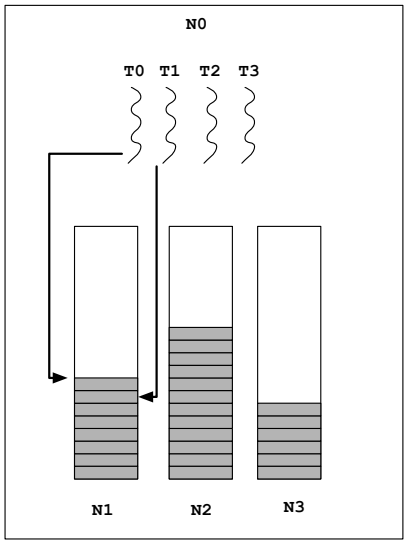


FIGURE 10.3. Node *N0* doing message aggregation for nodes *N1, N2, N3*. The aggregation buffers are concurrently modified.

10.1.2.2. *Self-Sending.* When a particular *workitem* is destined for the same node (i.e., the owner node of the *workitem* is same as where the *workitem* was generated), the runtime has the option of calling the processing function directly without routing through the network. We call this process *self-sending*. The performance of certain algorithms changes based on whether Self-Sending is enabled or not. These performance behaviors will be discussed in future chapters.

10.1.3. Thread Management. The runtime is responsible for creating parallel threads as well as terminating the threads. The number of threads depends on the number of cores. Every thread has a numerical identifier and threads are grouped based on the spatial levels.

The majority of the irregular applications are communication-bound applications. Therefore, applications with non-blocking communication performs better compared to applications with blocking communication (when communication is non-blocking, the application can attempt to perform more computations while the communication is taking place and hence can improve the compute/communication ratio). There are three common models that handle non-blocking communication in threads. They are:

(1) Every thread sends and receives *workitems*

(2) There are dedicated threads to execute processing function and to do ordering as well as other dedicated threads to send and receive *workitems*

(3) There are dedicated threads to execute processing function and to do ordering as well as other dedicated threads to send *workitems*. There are separate threads that receive *workitems*. (i.e., same thread will not both send and receive *workitems*).

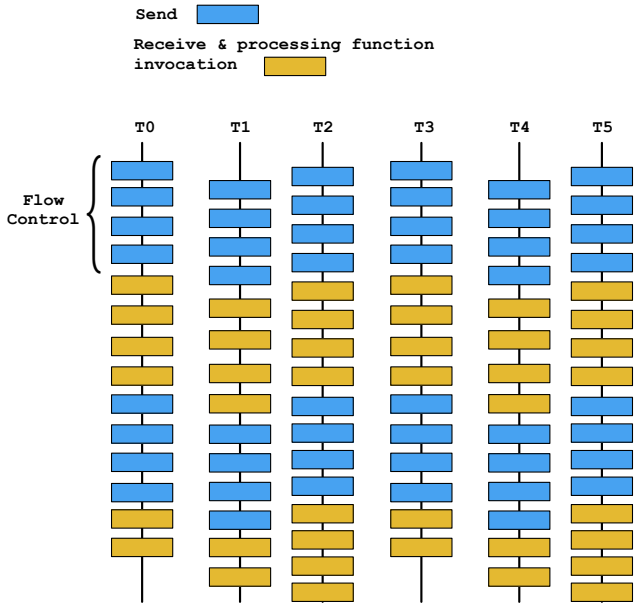


FIGURE 10.4. Every thread performs sending and receiving *workitems* and also executes processing functions.

Figure 10.4 shows the first model in which every thread performs sending, receiving and processing function invocation. In this model, a thread sends up to N number of *workitems* and then starts receiving *workitems* and invoking processing functions. We call this N the *the flow control* value. Figure 10.4 assumes that there is only one destination, but if we have to maintain multiple destinations we will have multiple flow controls maintained, one per every destination.

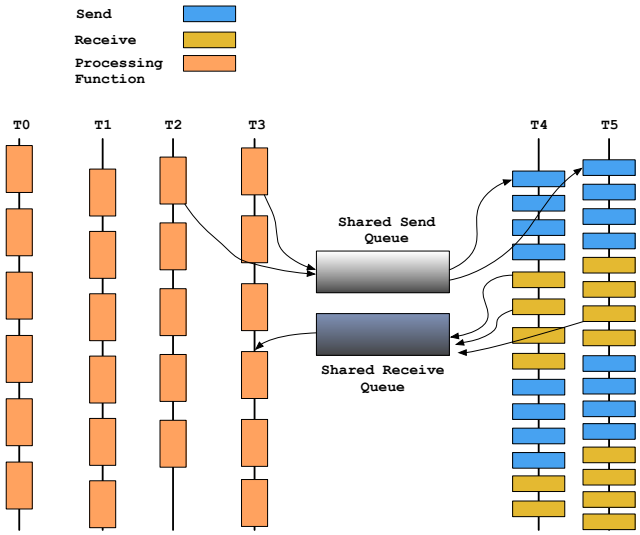


FIGURE 10.5. Only two threads perform sending and receiving *workitems* and other threads execute the processing function.

Figure 10.5 shows how dedicated send/receive threads operate. All the other threads, other than dedicated send/receive threads, perform processing function invocation. The dedicated send/receive threads and compute threads (threads that execute the processing function and ordering) communicate via a shared buffer. There are two buffers; one to exchange send *workitems* and another to exchange received *workitems*.

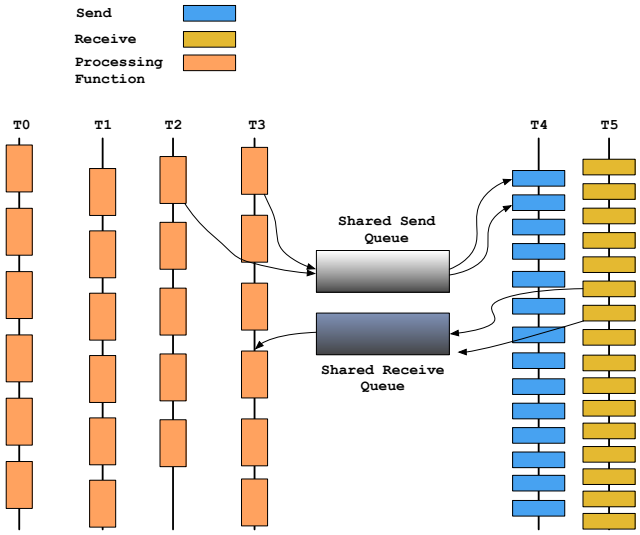


FIGURE 10.6. There are dedicated threads that perform sending and other dedicated threads to perform receiving.

The third model of computation and communication is shown in Figure 10.6. This model is similar to the model presented in Figure 10.5 with the exception that it has dedicated threads for sending *workitems* and also a dedicated set of threads to receive *workitems*.

10.1.3.1. *Thread Allocation to Spatial Domains.* A spatial domain specifies a region of memory. For some parallel threads, accessing a particular memory region is more efficient compared to other parallel threads. For example, Figure 10.7 shows spatial memory division in a typical super-computing cluster. For this example, we only considered three compute nodes. The memory belonging to all the nodes is the global memory. The memory local to a node is the node level memory. For this particular example, each node has two NUMA domains. Therefore, memory local to a NUMA domain is the NUMA spatial level gives us a thread memory where an application keeps thread local data.

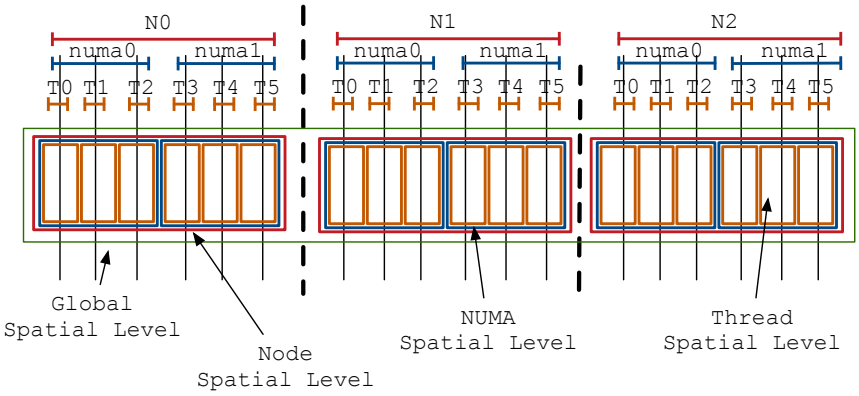


FIGURE 10.7. The spatial memory division.

For the example given in Figure 10.7, the thread *T0* is assigned to node *N0* and NUMA domain *numa0*. The runtime interface has functions to query information about spatial levels for a given thread ID.

Assigning threads to specific spatial regions is performed at the application bootstrap. For some spatial levels, this is trivial (e.g., node level threads). For other spatial levels, such as NUMA, the runtime needed to do the required pre-processing to decide which thread belongs to which NUMA domain.

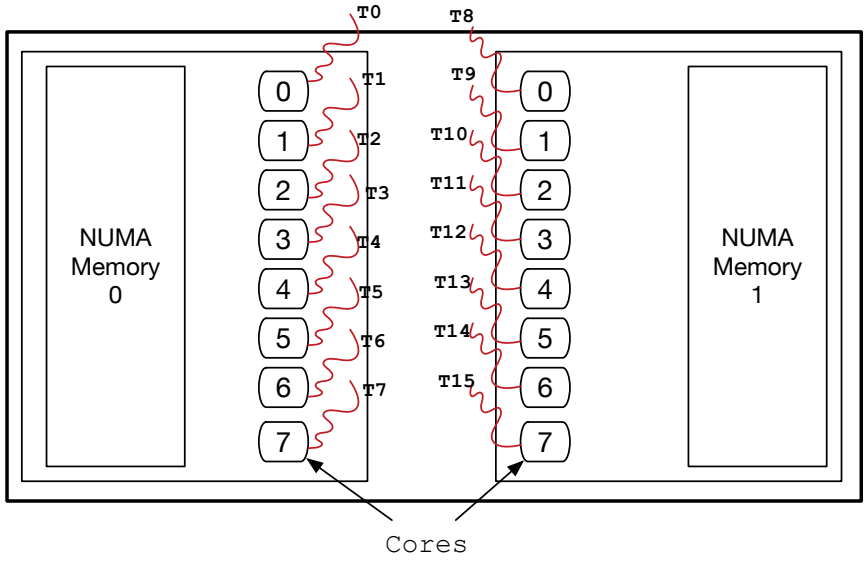


FIGURE 10.8. Pinning threads to cores to achieve NUMA spatial locality.

Figure 10.8 shows a processor with NUMA memory. As shown in the figure, there are two NUMA nodes and there are eight cores that are closer to each NUMA memory. The runtime that we used to implement the AGM framework, pins the threads to the cores to avoid any overhead from thread migration. Threads T0–T7 are pinned to cores in NUMA node 0 and threads T8–T15 are pinned to cores in NUMA node 1. The runtime is responsible for pinning threads to cores and for building the mapping between the cores and threads. More specifically, the AGM framework requires the following functionality from the runtime, related to NUMA spatial level:

LISTING 10.2. Runtime functions for querying NUMA domain details.

```
1 // returns the NUMA node id closer to given thread id
2 int find_numa_node(int tid){...}

4 // returns the number of threads in the NUMA domain where tid is
  in
5 int get_nthreads_for_numa_domain(int tid){...}

7 // the thread index of tid within the NUMA domain it is in
```

```
8  // e.g., T12 has index 4
9  int get_thread_index_in_numa_domain(int tid) {

11 // Every NUMA domain has a leader thread.  Generally,
12 // the smallest index thread is the leader.  This function call returns
13 // true if tid is the leader thread for the NUMA domain tid is in.
14 // e.g., T0 is the leader for NUMA node 0 and T8 is the leader
15 // for NUMA node 1.
16 bool is_main_thread_in_numa_domain(int tid) {..}
```

.

More generally, for a given spatial domain the AGM framework query following information from the runtime interface is as follows:

LISTING 10.3. Runtime functions for querying a general spatial domain details.

```
1  // returns the spatial domain id for the given thread id
2  int find_spatial_domain_node(int tid){...}

4  // returns the number of threads in the spatial domain where tid
   is in
5  int get_nthreads_for_spatial_domain(int tid){...}

7  // Every spatial domain has a leader thread.  Generally,
8  // the smallest index thread is the leader.  This function call returns
9  // true if tid is the leader thread for the spatial domain tid is
   in.
10 bool is_main_thread_in_spatial_domain(int tid) {..}
```

.

10.1.4. Synchronization. To make sure all participating parallel threads have finished processing an equivalence class before processing the next equivalence class, the AGM

framework must synchronize all parallel threads at the end of processing an equivalence class. In a similar fashion, the EAGM framework also needs to synchronize after processing an equivalence class for a given spatial level. Therefore, synchronization is required at different spatial levels and thus, the underlying runtime is responsible for implementing synchronization primitives. For the AGM framework implementation discussed in this thesis, the required runtime API for synchronization is given in Listing 10.4.

LISTING 10.4. Runtime functions for synchronization

```
1  // Performs global synchronization
2  void synchronize(int tid){...}

4  // Synchronize only the threads in the local node
5  void wait_for_threads_to_reach_here(int tid){...}

7  // Synchronize only the thread that are in the NUMA domain that tid
8  // is in
9  void wait_for_numa_domain_threads_to_reach_here(int tid) {...}
```

In Listing 10.4, Line 2 performs global synchronization and Line 5 performs node level synchronization. Line 9 synchronizes only the threads that are in the NUMA domain pinned to the thread id *tid*. The threads that do not belong to the *tid*'s NUMA domain are allowed to continue execution.

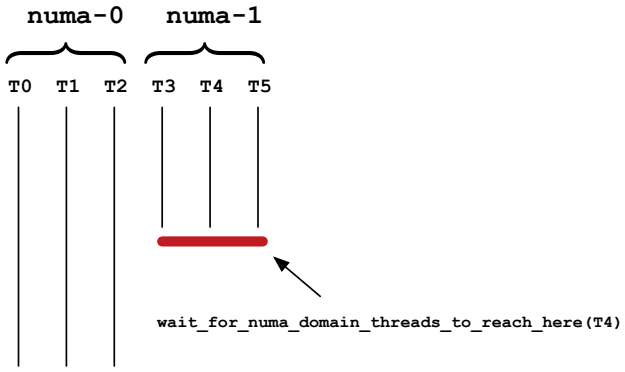


FIGURE 10.9. Synchronization at NUMA spatial locality.

As shown in Figure 10.9, when we invoke *wait_for_numa_domain_threads_to_reach_here(T4)*, only the threads that are pinned to cores in NUMA node 1 are suspended (until all threads in that NUMA domain reach the call). The threads that do not belong to NUMA node 1 are not disturbed by this call.

10.1.5. Distributed Termination Detection. Distributed termination detection is used to identify whether an algorithm has processed all the *workitems* in an asynchronous graph algorithm. More specifically, it identifies whether the algorithm has finished processing all the *workitems* pertaining to the current equivalence class. The reference **[95]**, discusses few distributed termination detection algorithms. Out of these algorithms, we use the *four counter termination detection* (**[95]**, Section 4) algorithm.

Algorithm 20 Distributed Termination Detection Algorithm (On a single node)

```
1:  while not terminated do
2:    while not idle do
3:      // do processing – ordering and processing function execution
4:      //...
5:    end while
6:    // thread does not have any work ...
7:    go to phase 1;
8:
9:    phase 1:
10:   globalactive ← allreduce(active);
11:   globalcomplete ← allreduce(complete);
12:   if globalactive== globalcomplete then
13:     go to phase 2;
14:   else
15:     continue;
16:   end if
17:
18:   phase 2:
19:   globalactive ← allreduce(active);
20:   globalcomplete ← allreduce(complete);
21:   if globalactive== globalcomplete then
22:     terminated ← True;
23:   else
24:     continue;
25:   end if
26: end while
```

The four-counter termination detection algorithm is given in Algorithm 20. The implementation maintains two counters in each locality: *active* to maintain the number of *workitems* that are generated but not yet processed and *complete* to maintain the number of *workitems* that are processed (i.e., completed executing the processing function). The *active* is incremented before pushing a *workitem* to an equivalence class and *complete* is increased when a *workitem* executes the processing function. When a thread reaches an idle state it performs a global reduction over the *active* and *complete*(Line 10). If they are equal, a reduction is carried out for the second time (See phase 2 in Algorithm 20). In both cases, if counts are equal, the thread decides if the algorithm is terminated (Line 22).

The AGM framework is required to modify the counts of *workitems* that are generated and when *workitems* finish executing processing functions. Therefore, the AGM framework requires the following API(Listing 10.5) calls from the runtime.

LISTING 10.5. Runtime functions for modifying termination counts.

```
1 // increments ``active''
2 int increment_active_count(int tid){...}

4 // increments ``complete''
5 int increment_complete_count(int tid){...}
```

.

10.2. Summary

This chapter discussed the API functions that a runtime should implement to interface the runtime with an AGM. We also discussed several design choices for certain runtime features. Some of the important runtime parameters are as follows:

(1) The allocation of threads for communication and computation,

(2) Flow control value,

(3) Whether message aggregation is enabled or not, and if enabled, the size of the maximum messages to aggregate,

(4) Is self-sending is enabled or not.

11

AGM Graph Processing Framework

In this chapter, we discuss how we mapped abstract AGM model to a concrete implementation. Some AGM concepts can be directly mapped to an implementation (e.g., a *workitem* can be directly mapped a message) and some of the other concepts required research to find the suitable implementation (e.g., processing function).

Finding a suitable data structure to store equivalence classes was challenging. We investigated several data structures to hold equivalence classes under concurrent execution. This chapter describes details of the data structures we investigated and their experimental results.

11.1. Implementation of AGM Concepts

The abstract representation of an AGM has the following concepts:

(1) A definition of a Graph

(2) A definition of a set *workitem*,

(3) A set of states,

(4) A processing function definition,

(5) A strict weak ordering relation,

(6) An initial *WorkItems*

In the following, I discuss how each of the above concepts is implemented in the AGM framework.

11.1.1. The Graph. The processing function makes use of the abstract data type Graph. The graph is maybe distributed, using a 1D block distribution or using an edge list distribution. The underlying runtime decides how the graph is distributed. In the current implementation, the local graph can be represented using a compressed sparse row format or adjacency list format.

Graph attributes (e.g., edge weights) are stored as distributed property maps. For example, for edge weights, every local edge has a mapping from edge id to the weight. The AGM framework does not need to access remote edges, whenever there is a requirement to access an attribute in a remote node, it is formulated as a *workitem* and sent over the network.

11.1.2. The set *WorkItems*. In the framework, a *workitem* is a tuple. We use C++ *std::tuple* interface. The *workitems* need to be sent from one node to another node. The underlying runtime needs to serialize the tuple into a message when sending over the network. For MPI, this is done by registering a message type. New data types need to be registered at the initialization of the runtime.

11.1.3. The set of States. A state is maintained against a vertex or an edge. In the implementation, a state is a mapping from a vertex (or edge) to a value type.

11.1.4. The Processing Function. The processing function is implemented as a *C++ functor*. The functor takes a *workitem* and a templated argument called outset. In the following, we discuss processing function implementation detail.

11.2. Processing Function Placement

The AGM framework models an algorithm as a processing function and an ordering. The processing function contains logic to update states and also to generate new *workitems*. A *workitem* generated in one rank may need to be transported to another rank to execute the processing function. The runtime is responsible for transporting a *workitem* from one distributed node to another. The runtime assesses the destination rank by checking the

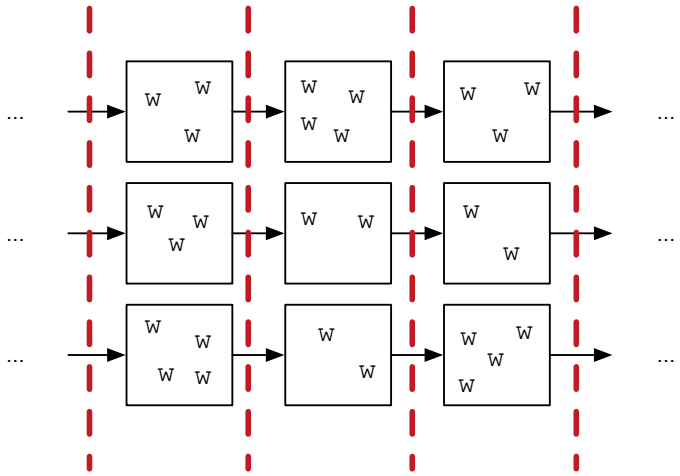


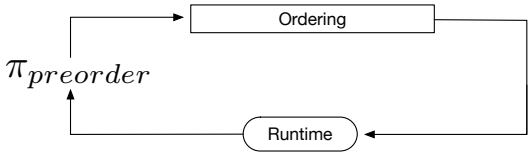
FIGURE 11.1. Local data structure in different ranks.

ownership of a *workitem*. For example, the ownership of a *workitem* is decided by a vertex distribution of a graph.

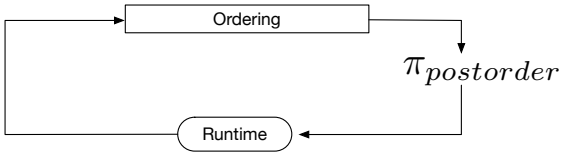
The ordering is performed within each node using a local data-structure. The local data structure has an element per every equivalence class the ordering generates. After locally processing all the *workitems*, all ranks synchronize to assure that they have finished processing the current equivalence class (Figure 11.1).

The data structure can be populated in two ways: 1. When a rank receives a *workitem*, it can first execute the π with the *workitem* and generate new *workitems*. The generated new *workitems* are populated into the data structure. Then, a *workitem* from the smallest equivalence class is popped out and sent it to the owning rank. This execution configuration is called *Pre-Order*(Figure 11.2a) 2. When a *workitem* receives a rank, it first inserts that *workitem* to the data structure for ordering. Then a *workitem* is popped out from the smallest equivalence class and executes the π. The generated *workitems* are sent to their owner ranks. We call this execution configuration *Post-Order* and it is depicted in Figure 11.2b.

Both pre-order and post-order configurations are discussed in detail in Section 11.2.1. For both pre-order and post-order configurations, the processing function logic is almost the same. However, there are differences in the way we calculate the initial *WorkItems*. The initial *WorkItems* bootstrap the algorithm. The π processes initial *workitems* to generate



(A) The processing function placed before ordering. Just after a *workitem* is received, it will execute the processing function.



(B) The processing function is placed after ordering but before a *workitem* is sent to the destination.

new *workitems*. When state converges to the desired output of the algorithm, processing functions to stop generating new *workitems*.

11.2.1. Pre-Order & Post-Order. In pre-order, *workitems* populated into the data structure are destined to execute a processing function in a remote rank. Therefore, the initial *WorkItems* should also be destined for remote ranks and, thus we need to ensure that the states are updated appropriately. However, in post-order, the *workitems* pushed to the ordering data structure are processed by the processing functions in the same rank. The *workitems* generated by processing functions are then transported to their owner ranks. Therefore, the initial *WorkItems* inserted into the data structure are different from pre-order.

In summary, in pre-order, *workitems* inserted into the data structure may execute a processing function in a different rank. However in post-order, *workitems* inserted into the data structure always execute a processing function in the same rank. For example, consider an SSSP algorithm and suppose the algorithm is finding the shortest paths in graph shown in Figure 11.3. Assume s is the source vertex and vertices are distributed as follows: vertices 3, 4 are in rank 0 (R0), vertex s is in rank 1 (R1) and vertices 1, 2 are in rank 2 (R2). In pre-order, *workitems* inserted into the data structure propagate state changes to neighbors of the source vertex. The initial *WorkItems* for the above example include $\{w_{i1}, w_{i2}, w_{i3}, w_{i4}\}$, where each w_{ij} represents a vertex-distance value, i.e., $w_{ij} = (j, d_j)$. Further, *distance* state for s is initialized to 0 (See Figure 11.4).

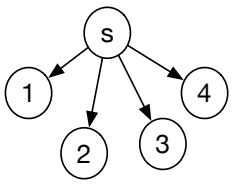


FIGURE 11.3. An example graph

For post-order execution (Figure 11.5), the initial *WorkItems* includes only the *workitem* generated from the source vertex (i.e., $(s, 0)$). The *workitem* inserted to the data structure invokes the post-order processing function on the same rank. For post-order processing

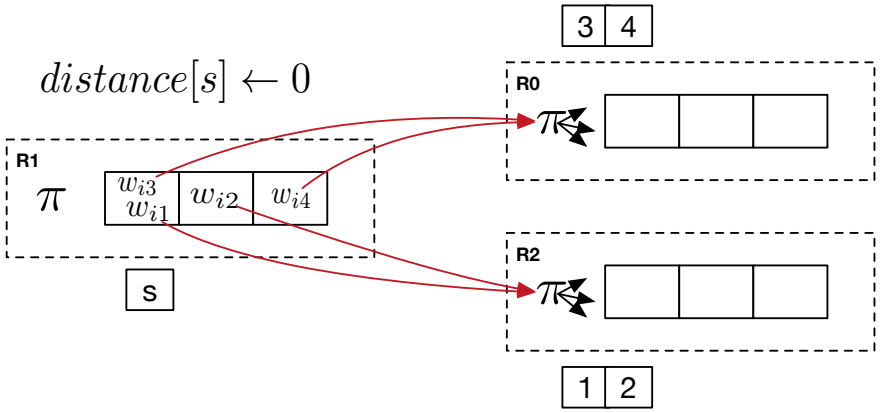


FIGURE 11.4. Initial *WorkItems* in pre-order execution.

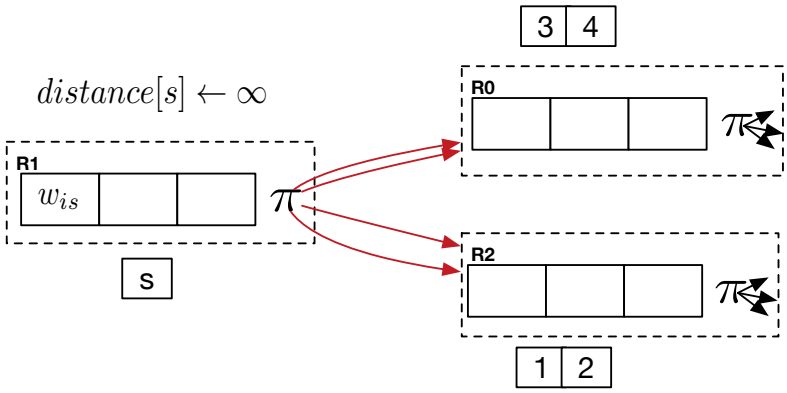


FIGURE 11.5. Initial *WorkItems* in post-order execution.

function to update the state for s and generate new *workitems*, the distance state for s is initialized to a very large value.

In the following, we use SSSP as an example to show pre-order and post-order execution configurations.

11.2.2. Example: Single Source Shortest Path. SSSP application finds the minimum distance to every vertex from a given source vertex. The pre-order processing function for SSSP is given in Listing 11.1.

An SSSP *workitem* is defined as a vertex and a distance. This definition of the *workitem* can be used to order work according to distinct distance values (e.g., Dijkstra's algorithm) and Δ range buckets. The SSSP algorithm uses *vdistance* state to maintain the minimum distance to a vertex from the source vertex. The distance in the incoming *workitem* is compared against the stored distance in the state. The distance in the state variable is updated if

the *workitem* contains a smaller distance (Line 8). The update is performed using Compare And Swap (CAS) operation since there can be multiple shared-memory parallel threads trying to update the same value. The CAS operation returns *true* upon successful update of the distance state, or else it returns as *false*. New work is generated for neighbors if the distance for a vertex is updated (Line 9). The variable *weight* is a property that contains the weight of each edge.

LISTING 11.1. The pre-order Processing function for SSSP

```
1 typedef std::tuple<Vertex, Distance> WorkItem;
2 struct sssp_pf {
3 void operator()(const WorkItem& wi, int tid, buckets& outset) {
4  Vertex v = std::get<0>(wi);
5  Distance d = std::get<1>(wi);
6  Distance old_dist = vdistance[v], last_old_dist;

8  if(CAS(d, vdistance[v])) {
9   for_each(Edge e : out_edges(v)) {
10   Vertex u = target(e);
11   WorkItem w(u, (d+weight(e)));
12   outset.push(w, tid);
13  }
14 };
```

The *vdistance* state is initialized as follows; i.e., vdistance[v] $\leftarrow \infty \forall v \in V - \{s\}$ and vdistance[s]$\leftarrow$ 0.

The post-order processing functions are similar to pre-order processing functions. However, instead of inserting *workitems* to the data structure, they are sent over the network to a remote rank. For example, the *outset.push* call in Listing 11.1 (Line 12) is replaced with *Send(w, tid)*. The *Send* function is responsible for sending the *workitem* to the appropriate destination rank based on the data distribution.

However, in pre-order, the initial state values and initial *workitems* are generated differently. For SSSP, *vdistance* is initialized to ∞ for all vertices (including the source vertex) and a *workitem* with source vertex and 0 distance is pushed into the data structure. Since the processing function is executed on the same rank for *workitems* in the data structure,

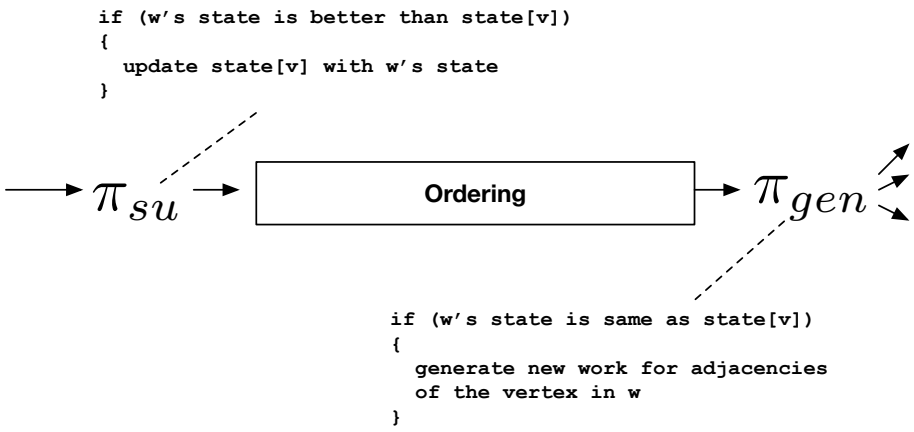


FIGURE 11.6. Separating processing into π_{su} and π_{gen}.

during the first execution, the source vertex state is updated and new work is generated for neighbors.

11.3. Split Order Processing

In pre-order, generated work is immediately pushed into the data structure. Since a lot of *workitems* could get inserted into the data structure at the same time, the contention on the data structure is high compared to post-order. In post-order, the generated *workitems* are distributed among multiple ranks, therefore the contention is not significant compared to pre-order.

To get the benefit of ordering further, we propose a way to split the processing into two logic functions. In Section 11.2, we mentioned that a processing function consists of logic to update states in the algorithm and also to generate new work for further processing. We separate the processing into two functions: 1. a function to update states (π_{su}), and 2. a function to generate new work (π_{gen}).

When a rank receives a *workitem*, π_{su} is invoked before performing the ordering. Consequently, the *workitem* is inserted into a data structure to do the ordering (Figure 11.6). Then, if a *workitem* with a better state value arrives the rank, the state written by a previous *workitem* can be updated and we avoid relaxing the previous *workitem* by having a condition in the π_{gen} function.

Separating processing function into π_{su} and π_{gen} allows us to prune work that may become redundant during the ordering. Hence, it reduces the number of messages sent over the network for label correcting algorithms (e.g., Δ-Stepping) and reduces the number of computations for label setting algorithms (e.g., FIX MIS). The π_{su} changes a state related to a *workitem* and π_{gen} generates new work for the *workitem*. For example, in SSSP, when a *workitem* arrives at a rank, it first updates the distance state with the distance in the *workitem*. Next, the *workitem* is inserted into the ordering data structure. Then a thread picks the ordered *workitem* and generates new work. However, the framework does not need to generate new work for the *workitem*, if the value in the distance state is smaller than the value in the *workitem*. Through the separation of state update logic and new work generation logic, framework can prune work.

The framework takes state update and new work generation functions as C++ *functors*. Listing 11.2 shows the π_{su} functor for SSSP. It takes a *workitem* and updates the distance state. Since multiple threads may access the distance state, the state update is carried out using an atomic CAS operation so that the thread that updates the smallest distance will succeed. CAS returns *true* if distance state is updated. Once the distance state is changed, the *workitem* is pushed into the data structure that performs ordering.

LISTING 11.2. State update function for SSSP

```
1 struct state_update_sssp_pf {
2 void operator()(const WorkItem& wi, int tid, buckets& outset) {
3      Vertex v = std::get<0>(wi);
4      Distance d = std::get<1>(wi);
5      if(CAS(d, distance_state[v])) {
6         outset.push(wi, tid);
7      }
8   }
9 };
```

Listing 11.3 shows the new work generation functor for SSSP. The functor compares the distance associated with *workitem* and the distance stored in the distance state. If they are equal work is generated for neighbors of *v*.

LISTING 11.3. New work generation for SSSP

```
1 struct new_work_gen_sssp_pf {
2 void operator()(const WorkItem& wi, int tid, buckets& outset) {
3      Vertex v = std::get<0>(wi);
4      Distance d = std::get<1>(wi);
5      if (d == distance_state[v]) {
6        for_each(Edge e : out_edges(v)) {
7          Vertex u = target(e);
8          WorkItem w(u, (d+weight(e)));
9          outset.push(w, tid);
10       }
11     }
12   }
13 };
```

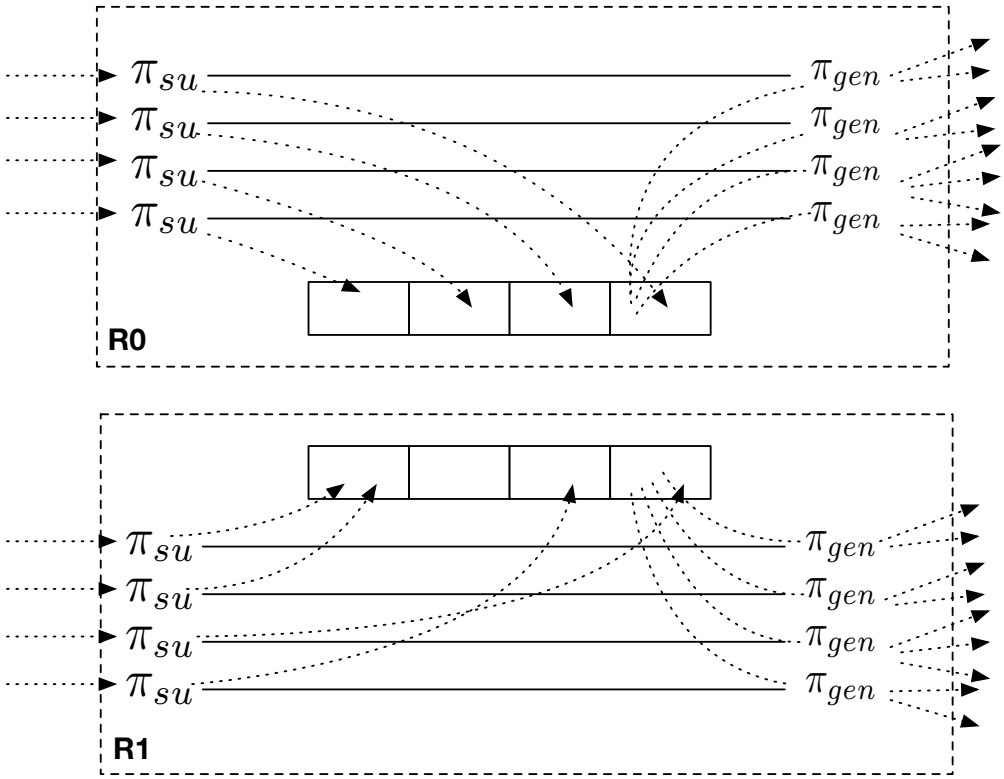


FIGURE 11.7. State update and new work generation processing functions in two ranks. Four threads per rank.

Figure 11.7 shows how π_{su} and π_{gen} processing functions are invoked by the framework in a system that consists of two ranks and four parallel threads per rank. Every computational parallel thread invokes the π_{su} and π_{gen} functions. A *workitem* is consumed by a π_{su} and the state is updated. Then, based on the strict weak ordering relation, that particular

workitem is pushed into the appropriate equivalence class. The π_{gen} processing function extracts *workitems* from the top most equivalence class and generates new *workitems*. The generated *workitems* are sent over the network to the appropriate rank.

11.4. Work Statistics

To further analyze the performance of different placements in processing logic we gather four different types of work statistics.

(1) *Useful Work*: If a *workitem* changes the state, and if no other work item overrides the state change made by this *workitem*, then it is useful.

(2) *Rejected Work*: If a *workitem* does not change the state, then it is a rejected work.

(3) *Invalidated Work*: If a *workitem* changes the state but another *workitem* overrides the state change made by the first *workitem*, then the first *workitem* is invalidated.

(4) *Invalidated Cancel Work*: If an invalidated *workitem* does not generate new work, then the current *workitem* is an invalidated cancel work. The pre-order and post-order execution configurations always generate zero *Invalidated Cancel Work*. However, split-order may generate *Invalidated Cancel Work*.

Algorithm 21 Work statistic generation.

state_update *workitem* w, *State s* :

```
1: v ← get_vertex(w)
2: if w changing s[v] then
3:   if s[v] is changed for the first time then
4:     increment Useful Work
5:   else
6:     increment Invalidated Work
7:   end if
8: else
9:   increment Rejected Work
10: end if
```

new_work_gen *workitem* w, *State s* :

```
1: v ← get_vertex(w)
2: if s[v] is not changed since it was update by w then
3:   generate new work
4: else
5:   increment Invalidated Cancel Work
6: end if
```

Execution	Useful	Rejected	Invalidated	Inv. Cancels	Add. Work
split-order	7367998	264092439	9771286	9749952	21334
pre-order	7367998	399551016	2469025	0	2469025
post-order	7367998	388691665	2284918	0	2284918

TABLE 11.1. Work statistics for scale 24 Graph500 graph on two ranks.

The work statistic calculation algorithm is given in Algorithm 21 for a general algorithm that executes with the split-order execution configuration. However, the work statistics calculation logic inside the *state_update* (Algorithm 21) function applies to pre-order and post-order execution configurations as well. As described earlier if a *workitem* is changing an associated state for the first time we increment the *Useful Work* (Line 4). If a *workitem* is over-writing a state value (Line 6) the algorithm increments the *Invalidated Work* count. If a *workitem* does not change the state at all, then that *workitem* is a *Rejected Work workitem*.

A portion of the generated *Invalidated Work* gets filtered out at the *new_work_gen* function (Algorithm 21). When processing a *workitem* in the *new_work_gen* function, if the state changed by the *workitem* is updated with a better state value by another *workitem*, the algorithm increments the *Invalidated Cancel Work* count (Line 5). The *Invalidated Cancel Work* statistic is only applicable to the split-order execution configuration. Also, it is important to note that new work is only generated for the amount of ((*Invalidated Work* +*Useful Work*)-*Invalidated Cancel Work*).

The split-order configuration eliminates the most amount of redundant work in algorithm execution. Table 11.1 shows the work statistics for execution of Dijkstra's SSSP algorithm on a Graph500 scale 24 graph on two ranks. As mentioned above, the *Invalidated Cancel Work* for pre-order and post-order are zero. All execution configurations perform the same amount of *Useful Work* work. The split-order execution performs the most amount of *Invalidated Work* work. However, most of the *Invalidated Work* performed by split-order get canceled at the π_{gen} function. Therefore, split-order generates the least amount of additional work (See "Add. Work" column). Here "Add. Work" is equal to (*Invalidated Work- Invalidated Cancel Work*).

Table 11.2 shows the execution time for each configuration and the number of inserts into the data structure. As demonstrated, the split-order is the fastest and performs with

Execution	Time(sec.)	Inserts
split-order	14.54	17139284
pre-order	62.55	409388039
post-order	44.09	398344581

TABLE 11.2. Run times for different execution orders.

the fewest inserts to the data structure. Since split-order eliminates most of the additional work it is also able to reduce the contention on the data structure.

As per Table 11.2, post-order performs better than pre-order. In pre-order, *workitems* are inserted into the data structure immediately after being generated. However, in post-order, generated *workitems* are distributed among ranks and *workitems* are inserted into the data structure at the destination. Therefore, contention on the data structure is less in post-order compared to pre-order. Therefore, pre-order execution shows better performance than post-order configuration.

11.5. Temporal Ordering

The strict weak ordering relation partitions work into "ordered equivalence classes". The strict weak ordering relation is specified to the framework as a C++ functor. Listing 11.4 shows an example of how Dijkstra's algorithm ordering is specified. The index parameter specifies the tuple location for the distance in a *workitem*. If two *workitems* have the same distance, then the *workitems* are not comparable. Hence, they belong to the same equivalence class. However, if their distances are different, they belong to different equivalence classes.

LISTING 11.4. Dijkstra's ordering for SSSP

```
1 template<int index>
2 struct dijkstra {
3 public:
4   template <typename T>
5   bool operator()(T i, T j) {
6     return (std::get<index>(i) < std::get<index>(j));
7   }
8 };
```

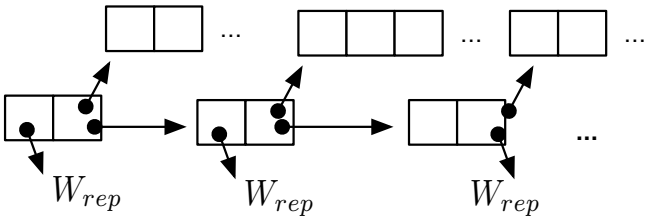


FIGURE 11.8. Data structure that holds *workitems*. Each node has a representative *workitem* and an append buffer.

Within a *Rank*, these partitions are maintained inside a *concurrent data structure*. The data structure supports functionalities similar to an abstract *dictionary data structure*. Every node in the dictionary data structure has a representative *workitem* and an append buffer (Figure 11.8). A *workitem* is inserted into an append buffer of a node only if the representative *workitem* is not comparable to the inserting *workitems*. The node with the smallest representative *workitem* stays in the front of the data structure.

11.5.0.1. *Inserts to Data Structure.* When the framework pushes a *workitem* to the data structure, it first finds the smallest node that has a representative *workitem* which is not less than the *workitem* being inserted (See *push* function in Algorithm 22, Line 2). If there is such node, its representative *workitem* is compared to the *workitem* being inserted. If the found node's representative *workitem* is not comparable to the *workitem* being pushed, *workitem* is inserted into the append buffer attached to the node (Line 5). If the found node's representative *workitem* is greater than the *workitem* being inserted, a new node is inserted before the found node. A pushed *workitem* is also added as the representative *workitem* and inserted into the append buffer associated with the newly created node (Line 10). Since non-comparable operation is transitive in a strict weak ordering relation, we can store the first *workitem* being inserted into the append buffer as the representative *workitem* and compare other incoming *workitems* to that *workitem*.

Multiple shared-memory parallel threads may insert *workitems* into the data structure. The framework needs to guarantee that the data is consistent when operating with multiple threads. While there are a number of ways to handle this, Algorithm 22 shows how we can use read-write locks to make sure that there are no race conditions. Multiple threads

can read and insert *workitems* into existing append buffers (Line 1), but modifying the data structure while inserting a new node is exclusive (Line 9).

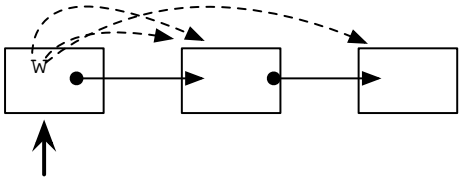
The append buffer associated with each node is also a concurrent data structure. Multiple shared-memory parallel threads may insert *workitems* to the end of the append buffer. In the implementation, we use an atomic-based append buffer to store *workitems* per equivalence class.

Algorithm 22 The *push* function

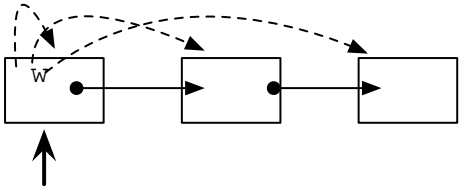
Push *workitem* wi, *strict-weak-ordering* $<_o$:

```
1:  read-lock()
2:    lb ← lower-bound(buckets, wi)
3:    if lb ! = buckets.end() then
4:      rep ← lb → representative
5:      if ((rep ≮o wi) && (wi ≮o rep)) then
6:        buffer ← lb → appendbuffer
7:        buffer.insert(wi)
8:      else
9:        write-lock()
10:       n ← new Node
11:       n → representative ← wi
12:       n → buffer.insert(wi)
13:       buckets.insert(lb, n)
14:     end if
15:   end if
```

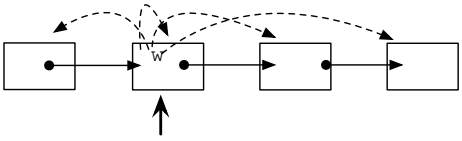
11.5.0.2. *Processing.* The data structure, discussed above is maintained in every *Rank*. Every rank starts processing the *workitems* in the first node's append buffer in the data structure. However, the first node in every *Rank* may not represent the same equivalence class. Therefore, before start processing the first node's append buffer, every node exchanges representative *workitems*. If representative *workitems* are comparable in two nodes, the node that has a higher equivalence class pushes an empty node to represent the smallest equivalence class. This operation requires a global reduction of representative *workitems*. When every rank's first node's representative *workitem* belongs to the same equivalence class every rank starts processing the *workitems* in append buffers associated to the first node of every rank.



(A) Generating *workitems* to higher equivalence classes.



(B) Generating *workitems* to same equivalence classes and to higher equivalence classes.



(C) Generating *workitems* to same, higher and smaller equivalence classes than currently processing equivalence class.

FIGURE 11.9. The *workitem* generation combinations.

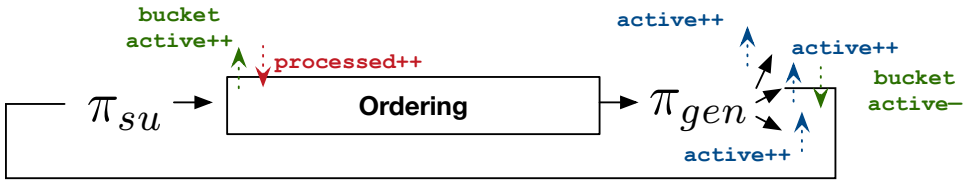


FIGURE 11.10. The life cycle of a *workitem* and how termination counts are modified.

Processing an equivalence class may generate work to an equivalence classes greater than the current equivalence class (Figure 11.9a), or it can generate work to greater equivalence classes and to itself (Figure 11.9b), or to any equivalence class (Figure 11.9c). The framework implements the most general scenario out of the combinations shown in Figure 11.9, i.e., generating *workitems* to any equivalence class, including the currently processing equivalence class.

To decide whether the framework has finished processing the current equivalence class, we use the *four counter termination detection* algorithm described in [**95**]. Every *Rank* maintains two counters locally: 1. *active count*, 2. *processed count*. The life cycle of a *workitem* starts at the π_{gen} function and it terminates when a *workitem* finishes its work after executing the π_{gen} function (Figure 11.10). The active count is incremented when a *workitem* is generated at the π_{gen} function whereas a passive count is incremented when it is pushed into the data structure. Before pushing a *workitem* to the data structure, the framework increases another atomic counter, *bucket-active*. The bucket-active count is decremented after the framework invokes π_{gen} function. The *bucket-active* count represents the number of *workitems* pushed to the current equivalence class. When the global active count summation is equal to the global processed count summation for two consecutive iterations, the four counter algorithm decides that all the messages are exchanged relevant to the current iteration of the equivalence class. Then, if the bucket-active count is also zero, the framework decides that it has finished processing the current equivalence class.

If a *workitem* is *not* destined to the current equivalence class, then it will not increment the bucket-active count right away. To handle this case, each node in the data structure maintains a counter: *pending active counter*. When a *Rank* receives a *workitem* that is comparable to the representative *workitem* of the currently processing equivalence class, it first increments the "pending active counter", and then, increments the processed count and inserts the *workitem* to the appropriate node in the data structure. Then, at the start of processing an equivalence class, the active count is increased by the "pending active count" for the node.

A code region that modifies active, passive counts and performs message send/receive is called an *epoch*. If a particular ordering is always generating work to an equivalence class, the number of epochs required is the same as the total number of equivalence classes generated by the algorithm. However, if processing current equivalence class generates work for the same equivalence class, then we need an additional epoch to decide that the current equivalence class has finished processing.

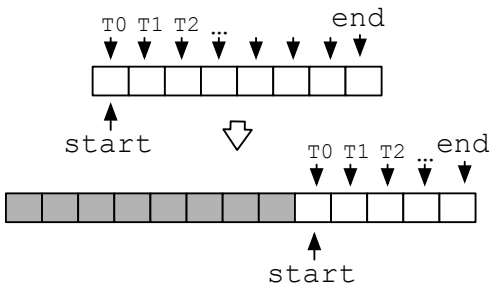


FIGURE 11.11. Processing of a single class.

A more precise logic for processing equivalence classes is listed in Algorithm 22. The framework selects the top node from the data structure (Line 5). If the data structure is empty, an un-initialize *workitem* is created (See Line 8, An un-initialized *workitem* is a *workitem* that has its all tuple values initialized to maximum possible value). The framework then finds the smallest *workitem* in the system using the strict weak ordering relation (Line 10). If the reduced *workitem* is also un-initialized, then data structures in all ranks are empty. Hence we can terminate the processing (See Line 11 and Line 20). If the reduced *workitem* is less than the representative *workitem* in the top node, a node with an empty buffer is inserted into the data structure (Line 14). Then, the framework selects the top node and initializes the *bucket-active* count to the *pendingactive* count for the node. After updating the active count, every rank starts processing the top equivalence class (Line 23).

When processing an equivalence class, *workitems* may get generated into the same equivalence class. Therefore, the framework performs an all reduce to calculate the total number of *workitems* in the current equivalence class (Line 2). If the calculated amount is not zero, the framework invokes *ProcessClass* with the equivalence class.

After all participating nodes decide that current equivalence class has finished processing, it will be removed from the data structure (Line 26). Then, the first node in the data structure is chosen for next execution (Line 5). This way the execution proceeds until there are no more equivalence classes in the data structure. When there are no more classes to process algorithm terminates.

11.5.0.3. *Processing a Single Equivalence Class.* Processing of a single equivalence class is shown in Figure 11.11. When processing an equivalence class, *workitems* can get generated

Algorithm 23 The *Process* function, in a single *Rank*

Process *threadid tid* :

```
1:  while true do
2:    workitem wi
3:    if tid is the main thread then
4:      if !buckets.empty() then
5:        top ← buckets.first
6:        wi ← top.representative
7:      else
8:        wi ← Un-initialized
9:      end if
10:     workitem w_red ← AllReduce(wi, <_o)
11:     if w_red==Un-initialized then
12:       shouldbreak = true
13:     else if w_red <_o wi then
14:       PushEmptyNode(w_red)
15:     end if
16:     top ← buckets.first
17:     active ← top.pendingactive
18:   end if
19:   threadbarrier.wait()
20:   if shouldbreak then
21:     break
22:   end if
23:   process(top, tid)
24:   if tid is the main thread then
25:     top.clear()
26:     buckets.pop()
27:   end if
28: end while
```

PushEmptyNode w_{red}:

```
1: n ← new Node
2: n → representative ← w_red
3: insert n to buckets
```

Process *top*, *threadid tid* :

```
1: while true do
2:   allactive ← AllReduce(active)
3:   if allactive ! = 0 then
4:     epoch
5:       ProcessClass(top, tid);
6:   else
7:     break
8:   end if
9: end while
```

to the same equivalence class. The newly generated *workitems* are appended to the back of the buffer for the current node. Therefore, while processing an equivalence class, the length of the buffer can be changed and new *workitems* get added to the buffer.

Within a *Rank*, every parallel thread picks a *workitem* from the buffer and executes the π_{gen} function (See the loop in Line 7 of Algorithm 24). The *start* index of the buffer is kept as a member variable and initialized to the beginning of the buffer (Line 1). The processing of the append buffer may stretch into several iterations. Therefore, after finishing an iteration, the start index for the next iteration is stored as the end of the previous iteration (Line 11). New end index is calculated at the start of each iteration. The *ProcessClass* function is invoked until all the *workitems* are processed for the equivalence class.

Algorithm 24 The processing of a single equivalence class in a single *Rank*

Initialize *Node top* :

```
1: start ← top → buffer.begin()
```

Process *Node top*, *threadid tid* :

```
1: buffer ← top → buffer
2: if tid is main thread then
3:   end ← buffer.end()
4: end if
5: threadbarrier.wait()
6: while start ! = end do
7:   for i = start; i < end; i+ = threads do
8:     invoke π_gen with buffer[i]
9:   end for
10:  if tid is main thread then
11:    start ← end
12:  end if
13:  threadbarrier.wait()
14: end while
```

11.6. Data Structure for Equivalence Class

As discussed above, the data structure that holds equivalence classes locally maintains equivalence classes in a "Dictionary" data structure. It also globally synchronizes after processing an equivalence class. We evaluated the performance of several data structures that hold *workitems* locally. In sequential execution we can use a tree data structure to hold equivalence classes (e.g., Standard Template Library(C++) (STL) map or set). However,

the framework data structure needs to handle concurrent inserts to equivalence classes and thus concurrent modification of equivalence classes. We experimentally evaluated several possible data structures:

(1) Linked List

(2) Binary Search Trees

(3) Concurrent SkipList

(4) Partitioning Scheme

11.6.1. Linked List. Every compute *Rank* maintains a linked list. A node in the linked list has a representative *workitem* and a pointer to an append buffer. The append buffer holds the *workitems* for the equivalence class. Inserting a new *workitem* has the complexity of $O(n)$, where n is the maximum possible number of equivalence classes. An advantage of the Linked List implementation is that we do not need to lock the whole data structure when inserting *workitems* or when inserting new equivalence classes. When inserting a new equivalence class, we only need to lock the node immediately before the one that is being inserted. However, we need to be careful to handle race conditions that may occur when there are *workitems* trying to create the same equivalence class at the same time.

11.6.2. Binary Search Trees. Binary Search Tree (BST) can search an equivalence class with $O(lg(n))$, a time complexity. However, BST data structures re-balance the tree during insertion and deletion to guarantee the search complexity. Therefore, we need to lock the whole data structure to assure that data structure is not in an inconsistent state after data structure operations.

11.6.3. Concurrent SkipList. SkipList [**114**] is an alternative probabilistic data structure that gives the same complexity guarantees as BSTs but avoids the need for re-balancing. Since SkipList avoids the need for re-balancing, it is a good candidate for concurrent execution. A concurrent, lock-free SkipList algorithm is listed in [**65**]. An implementation of this algorithm is also available in LibCDS (`http://libcds.sourceforge.net/`). We use this implementation in our framework and evaluate its performance.

11.6.4. Partitioning Scheme. In this method, every rank maintains two sets of vectors to store *workitems*. They are *storing vectors* and *processing vectors*. Each set contains vectors equal to the number of parallel threads. The first set holds the incoming *workitems* that do not belong to the current equivalence class and the second set of vectors store the *workitems* that belong to the currently processing equivalence class. The framework also maintains a *workitem* to represent the currently processing equivalence class (W^g_{min}). When a thread receives a *workitem*, it is compared against the W^g_{min}. If the *workitem* is not comparable to W^g_{min}, then it is directly processed, otherwise the *workitem* is pushed into the storing vector that belongs to the processing thread. Assuming the vector insert does not cause a resize of the vector, the insert operation can be done in $O(1)$.

Further, every thread maintains a minimum (as per the strict weak ordering relation) *workitem* inserted into the storing vector. After processing an equivalence class (i.e., when processing vectors are empty), the framework calculates the global minimum *workitem* by performing a global reduction. This minimum value is assigned to W^g_{min}. Then, using W^g_{min}, storing vectors are partitioned and *workitems* that are not comparable to W^g_{min} are moved to at the end of each vector. The partitioned *workitems* are moved to the processing vector of the relevant thread and the storing vectors are resized to the release memory (See Figure 11.12 for details).

To avoid in-node load imbalance, *workitems* collected to a vector of one thread are processed by all the threads (Figure 11.13). Once one vector is processed, all threads start processing the next vector. In other words, all processing vectors are arranged in a horizontal line and parallel threads process elements in the vectors (As depicted in Figure 11.13). However, when consuming work in processing vectors, new work is not inserted into them. Therefore, we do not need to use locking while executing *workitems* in processing vectors.

In all other cases, equivalence classes are created at the time of inserting a *workitem*, but in this scheme, *workitems* are partitioned after processing a previous equivalence class. This approach reduces the contention when processing in multiple threads. However, this approach consumes time when partitioning the next equivalence class. The partitioning

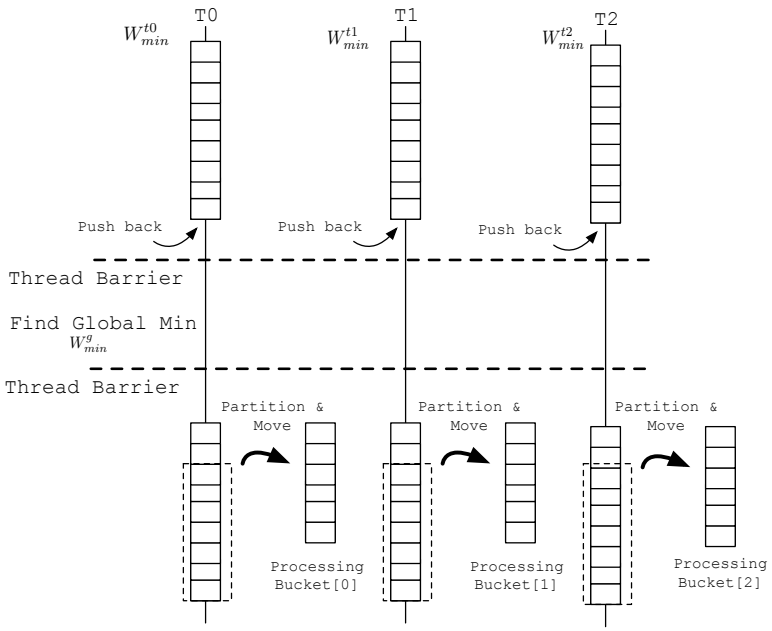


FIGURE 11.12. Partition scheme functionality.

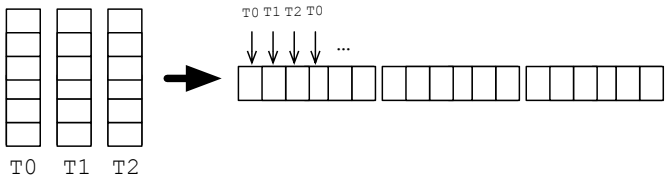


FIGURE 11.13. Partition scheme functionality: avoiding in-node load imbalance.

Data Structure	Execution Time (sec.)
Linked List	≈ 60
BST	≈ 82
Concurrent SkipList	≈ 89
Partitioning Scheme	≈ 50

TABLE 11.3. SSSP Δ-Stepping algorithm execution time with different data structures. Experiment is ran on a Graph500 scale 25 on four ranks and 16 parallel threads per rank

process could take $O(n)$ amount of comparisons with the strict weak ordering relation and could take $O(n)$ amount of swaps to move *workitems* to the end of the vector. Here n is the length of a storing vector.

11.6.5. Performance. Table 11.3 shows execution time (pre-order execution configuration) of Δ-Stepping AGM algorithm on a scale 25 Graph500 graph input on four ranks. As can be seen in the table, the best performance is achieved with the partitioning scheme.

We believe the poor performance in SkipList is due to the contention caused by concurrent threads, BST shows poor performance since we have to lock the whole data structure to avoid any inconsistencies that could occur from tree balancing. The Linked List data structure shows slightly better performance since we do not lock the whole data structure but only the node that is being inserted. The partitioning scheme uses the minimum amount of locks, hence the contention is minimized.

11.7. AGM Framework Usage

Listing 11.5 shows how the AGM framework can be invoked. The framework takes definitions of a graph, *workitem*, processing function, and ordering as template arguments. The RuntimeModelGen creates an instance of a runtime with the interface functions defined in Chapter 10. When instantiating the AGM, we need to pass an instance of a RuntimeModelGen, an instance of an ordering functor, a processing function instance, and an initial *workitem* set.

LISTING 11.5. Invoking AGM for SSSP

```
1 // SSSP (AGM) algorithm
2 typedef agm<Graph,
3           WorkItem,
4           ProcessingFunction,
5           StrictWeakOrdering,
6           RuntimeModelGen> sssp_agm_t;

8 sssp_agm_t ssspalgo(rtmodelgen,
9                 ordering,
10                pf,
11                initial);
```

Once executed, the AGM framework will populate initial *workitems* to the data structure and start processing the smallest equivalence class. As equivalence class process works locally, it may get more work. When a *Rank* does not have more work for the current equivalence class, it will attempt to perform termination for the current equivalence class. If all the ranks have finished processing work for the current equivalence class, then all ranks move to the next equivalence class. Figure 11.14 shows the processing of

equivalence classes by the framework (Δ-Stepping algorithm). The dashed red lines show locally finished work but the rank gets more work for the same equivalence class. The thick red line shows an end of processing an equivalence class by all ranks.

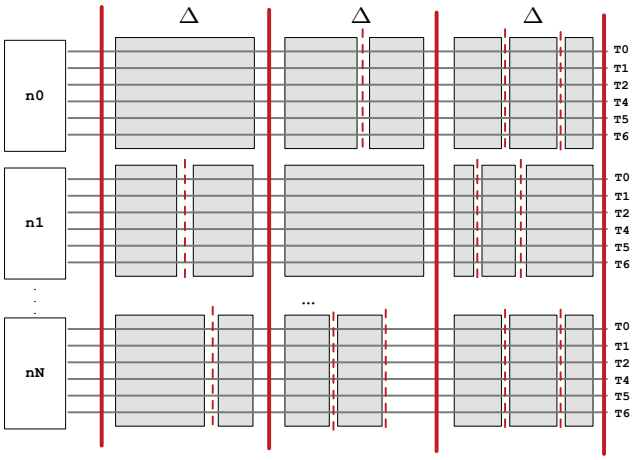


FIGURE 11.14. AGM execution of Δ-Stepping algorithm.

11.8. Summary

This chapter describes the AGM framework implementation. Two main inputs to the framework are the processing function and ordering. With these two inputs, the framework executes the algorithm.

AGM is an abstract model and does not involve a description of a runtime. However, when mapping AGM to the actual distributed hardware, we need to incorporate a runtime that wraps functionalities related to threading, messaging etc.,. We identified that there are three approaches to place the processing function in the framework: 1.pre-order, 2. post-order, 3. split-order. This chapter also showed that out of these three choices, the split-order approach demonstrates better performance as it is able to eliminate the most amount of redundant work and reduce the contention on the data structure.

Further, we investigated several data structures to hold equivalence classes within a rank and showed that the partitioning scheme shows best performance since it is able to avoid much of the contention.

12

EAGM Graph Processing Framework

Expressing an algorithm using a processing function and an ordering is abstract and independent of the implementation. However, distributed-memory graph algorithms are strongly impacted by the properties of the distributed architecture that they run on. To capture this impact, we map ordering of *workitems* to different *spatial distributions* on a distributed-memory platform.

Currently, we recognize 4 hierarchical levels of distribution that roughly match modern distributed systems (arrows indicate inclusion):

Global $\longrightarrow$ **Process** $\longrightarrow$ **Numa** $\longrightarrow$ **Thread**

These spatial memory levels are depicted in Figure 12.1. As per the figure, every *Rank* has its own memory (red boxes) and collectively all the memory in all ranks forms the global memory (green box). Further, every *Rank* has two NUMA domains and in each one, there are three parallel threads relevant to the three cores in the NUMA domain.

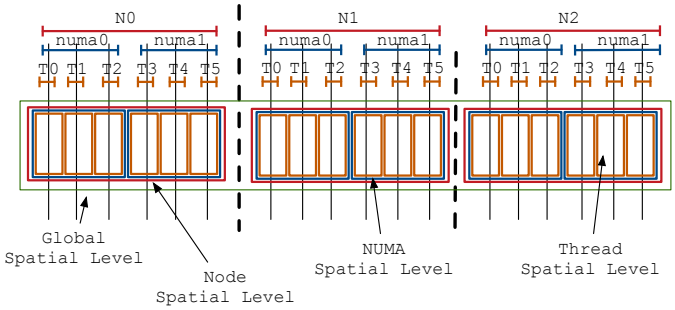


FIGURE 12.1. Spatial memory hierarchy.

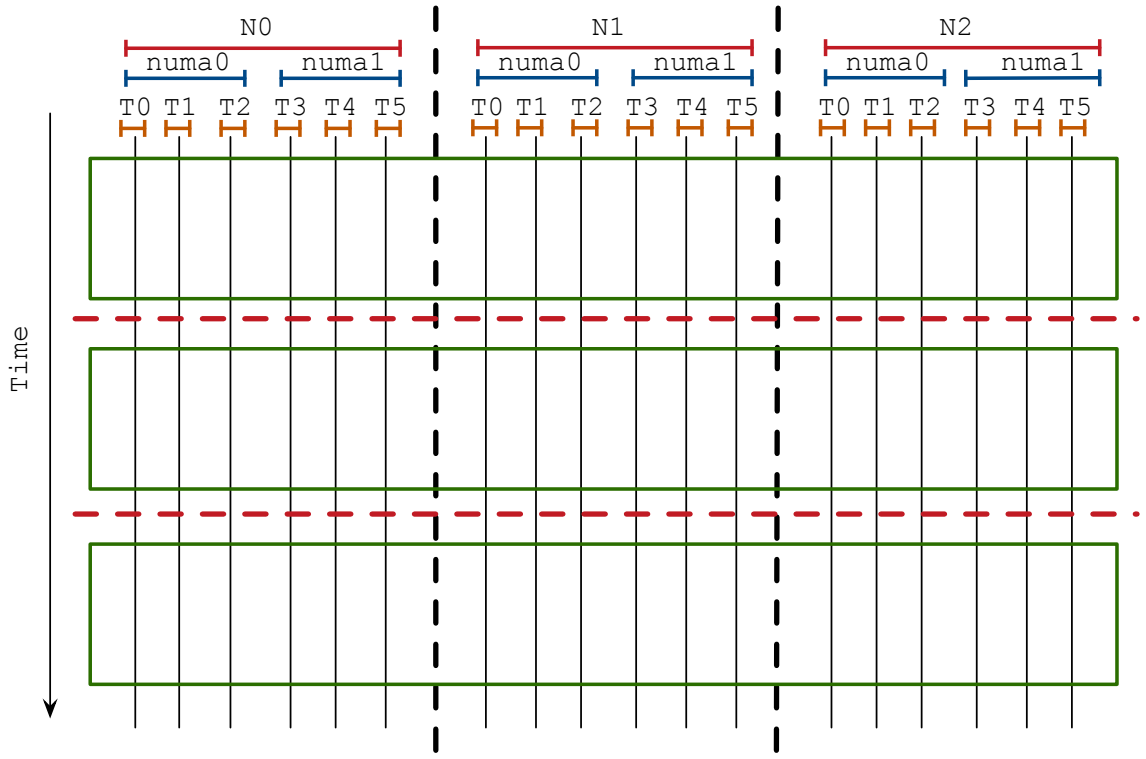


FIGURE 12.2. Global ordering Δ-Stepping SSSP algorithm execution.

Spatial orderings apply non-semantic ordering on *workitems* throughout the spatial hierarchy of a distributed machine. The ordering at the **Global** level creates global equivalence classes. The global equivalence classes can be further ordered at the lower levels of the hierarchy. For example, two different spatial orderings for Δ-Stepping (with $\Delta = 5$) are $<_{\Delta(5)} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{ch}$ and $<_{\Delta(5)} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{dj}$ where each ordering corresponds to the appropriate spatial level (the orderings are as defined in the previous section). The first spatial ordering enforces $<_{\Delta(5)}$at the global level, but leaves execution in buckets unordered ($<_{ch}$). Execution of this algorithm is depicted in Figure 12.2. The second spatial ordering applies Dijkstra's ordering at the **Thread** level ($<_{dj}$). This means that *workitems* at every thread are ordered in a priority queue as they reach the thread in the spatial distribution.

Ordering applied to the *Process* level creates equivalence classes locally at the process. After processing an equivalence class, all threads underneath the node are synchronized. For example, the spatial ordering $<_{\Delta(5)} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{ch}$ only creates equivalence classes

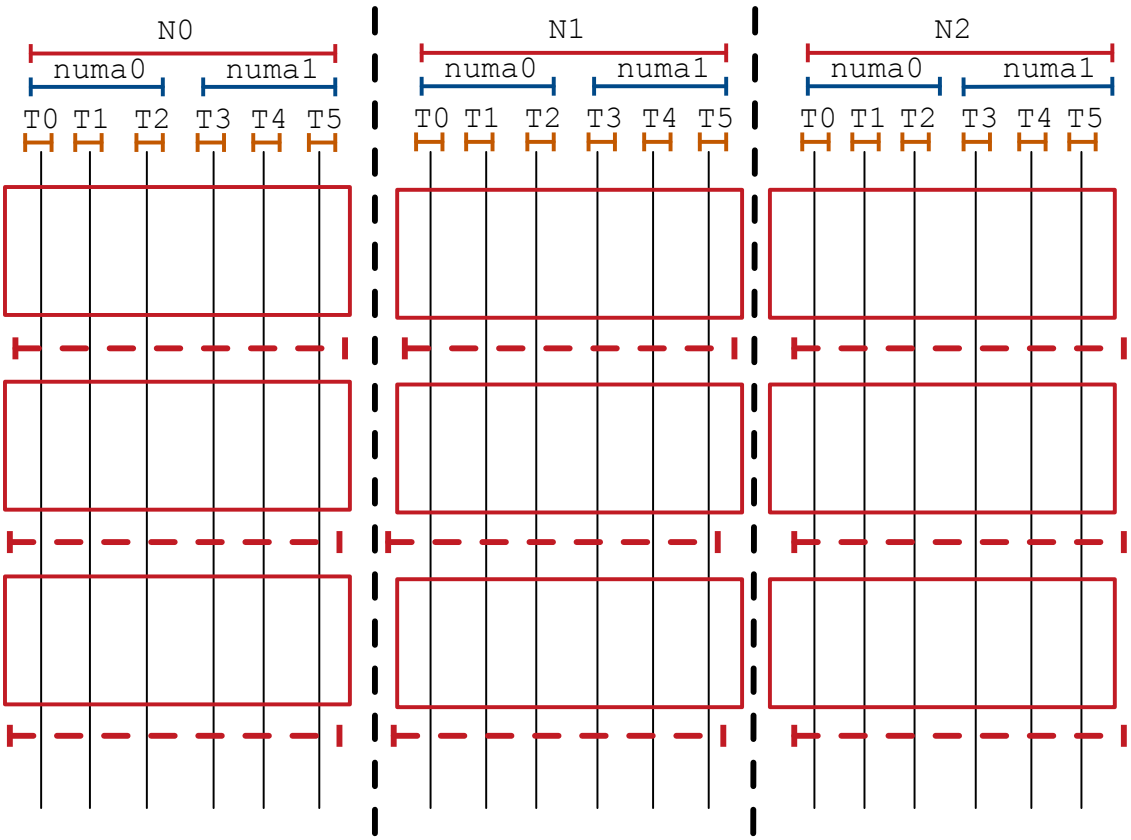


FIGURE 12.3. Globally asynchronous, but process level synchronous.

at the process level. Each *Rank* has its own equivalence classes and synchronization takes place within the *Rank* only. There are no global equivalence classes like in ordering, $<_{\Delta(5)} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{ch}$. See Figure 12.3.

Similarly, when ordering is applied to NUMA level, equivalence classes are created and threads underneath a particular NUMA domain are synchronized. We assume that within a node parallelism is constrained to the number of cores available in the node and each parallel thread is pinged to a core. A parallel thread belongs to a NUMA region if it is pinged to a core in that particular NUMA region.

Figure 12.4 summarises the spatial and temporal execution of a graph algorithm. The primitive workforce of a parallel algorithm is a parallel thread. Every parallel thread executes the processing function for the algorithm. The boxes enclosed in green shows equivalence classes generated for global spatial level. After processing a global equivalence class,

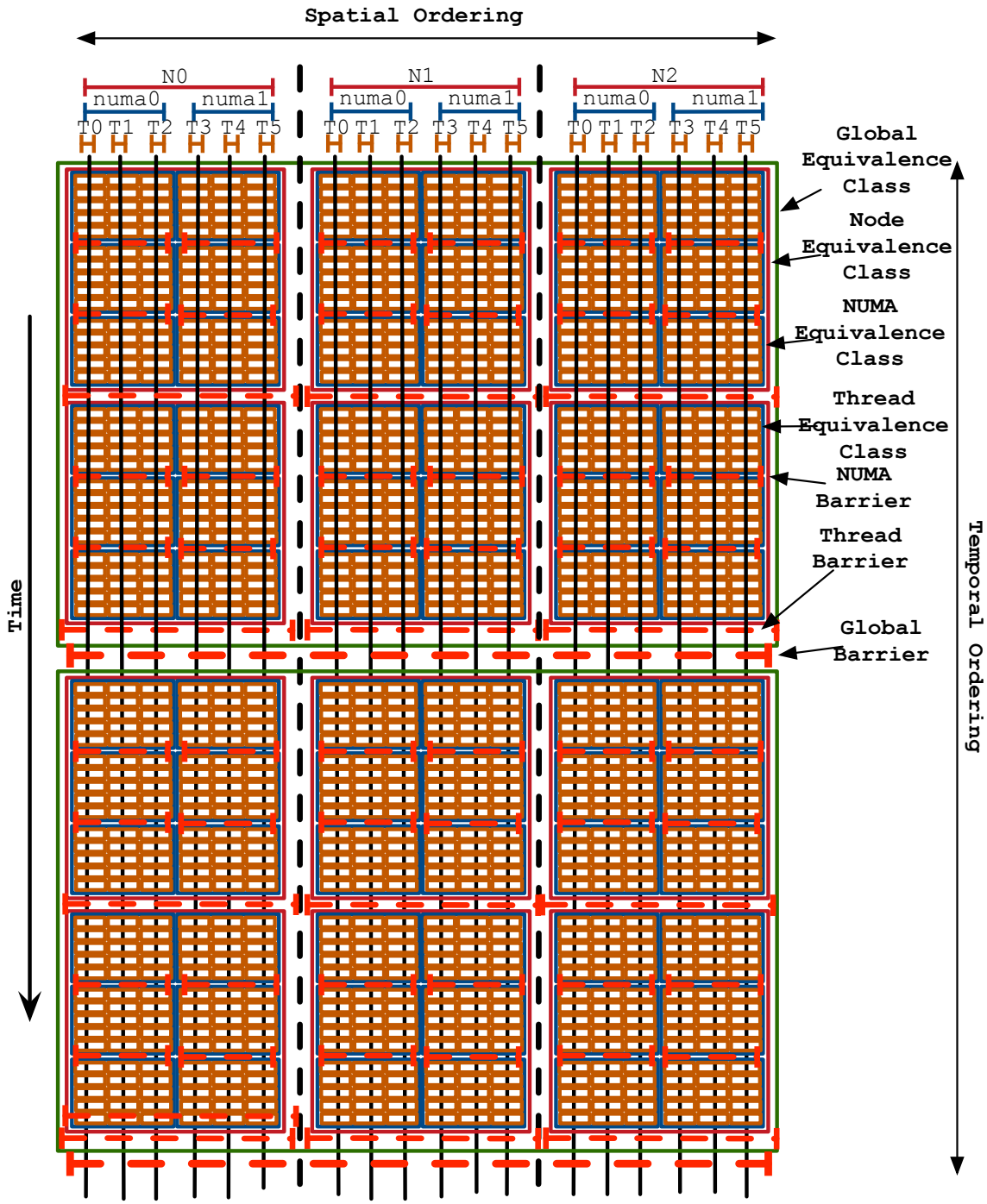


FIGURE 12.4. Spatial & Temporal ordering execution.

a global barrier is executed. The same logic is applied to lower spatial levels in the memory hierarchy. Within a global equivalence class, there can be several equivalence classes generated from node level ordering. After processing a node level equivalence class, all the parallel threads belonging to that node are synchronized. Similar logic is applied to NUMA level equivalence classes.

The ordering controls the amount of work generated. However, too much ordering comes with synchronization overhead. In this model, we control the amount of work generated as well as the overhead of synchronization by defining strict weak ordering relations at each spatial level. For example, if we specify chaotic ordering at the global level and specify $<_{\Delta(5)}$ at the node level, we get a single large equivalence class at the global level and Δ ordering at node level. Hence, the algorithm is globally asynchronous by orders working in Δ buckets at the node level.

12.1. Spatial Ordering Implementation

The framework is further extended to explore orderings of *spatial memory distribution* on a distributed memory platform. Each level of a spatial memory hierarchy is annotated with an ordering. For example, consider the following memory hierarchy:

Global $\longrightarrow$ **Process** $\longrightarrow$ **Numa** $\longrightarrow$ **Thread**

. For this memory hierarchy, two example ordering configurations are $<_{\Delta(5)} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{ch}$ and $<_{ch} \rightarrow <_{\Delta(5)} \rightarrow <_{ch} \rightarrow <_{ch}$. The first configuration creates global equivalence classes and separates *workitems* into global equivalence classes based on their distances. The second configuration creates similar equivalence classes but at the node level. The global execution is asynchronous for the second configuration. Synchronization is only performed at the node level.

Figure 12.5 depicts how framework executes second ordering. The framework creates a single large equivalence class for the chaotic ordering. At the node level, after processing each equivalence class, there is a thread barrier executed in each node. It is important to note that this barrier is local to a node. The algorithm performs global synchronization

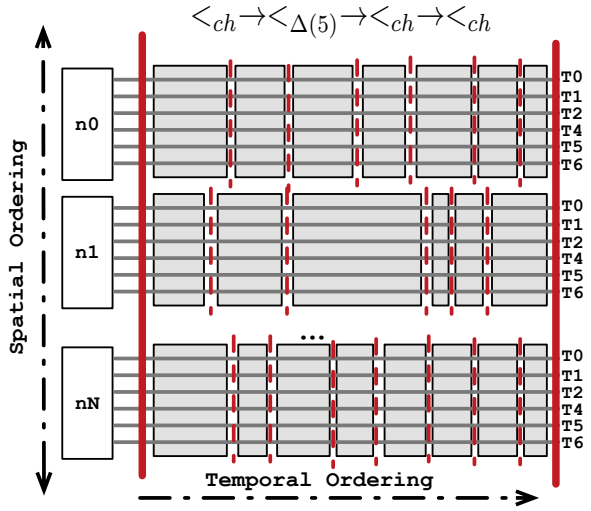


FIGURE 12.5. Execution of ordering $<_{ch} \rightarrow <_{\Delta(5)} \rightarrow <_{ch} \rightarrow <_{ch}$.

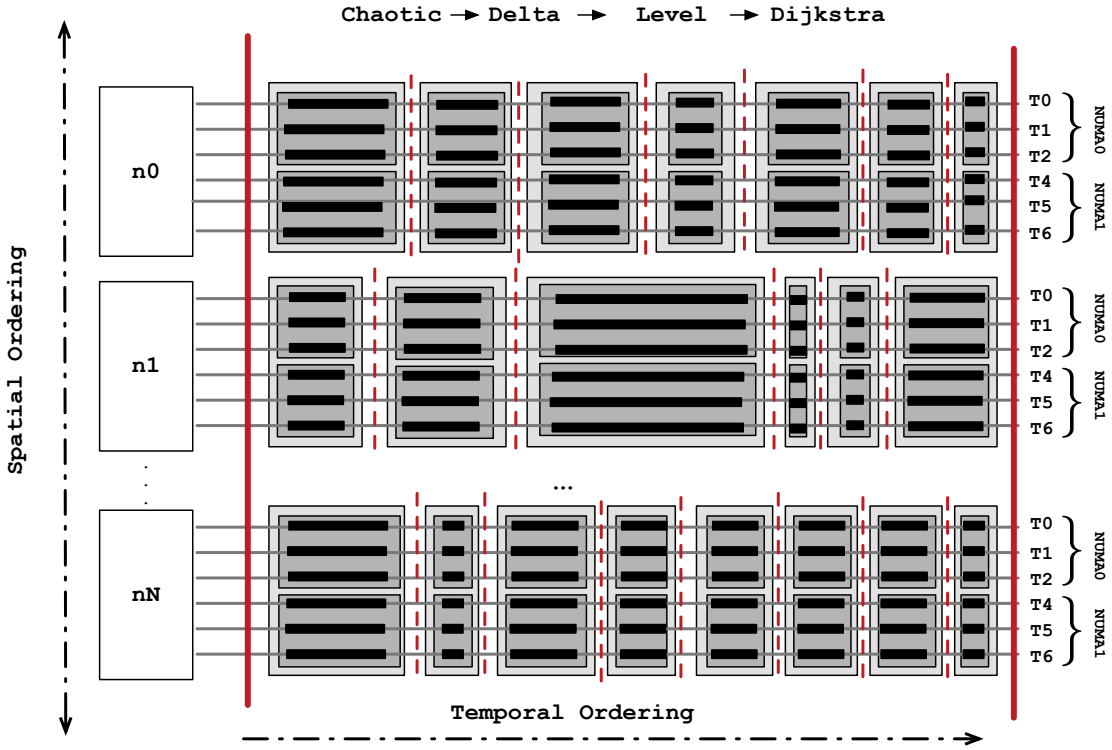


FIGURE 12.6. Execution of ordering $<_{ch} \rightarrow <_{\Delta(5)} \rightarrow <_{level} \rightarrow <_{dj}$.

only once. We say this algorithm is globally asynchronous, node level synchronous, numa level asynchronous, and thread level asynchronous. In other words, whenever there is a chaotic ordering defined for a memory level, the execution is asynchronous.

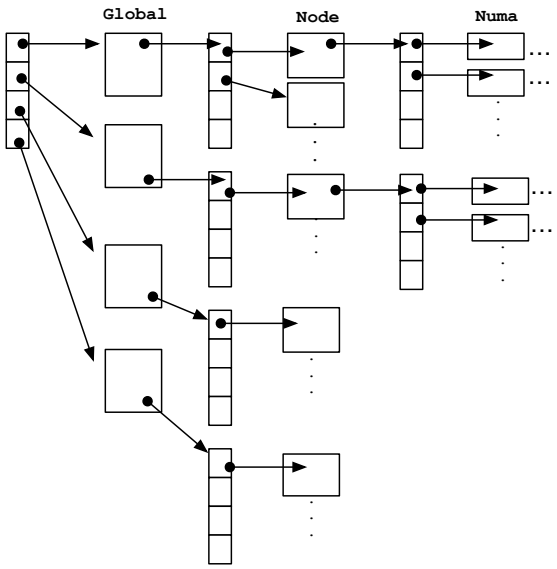


FIGURE 12.7. The data structure that stores spatial and temporal *workitems*.

The execution of another ordering configuration is depicted in Figure 12.6. This ordering configuration executes *workitems* globally asynchronously. However, at the node level and numa level algorithm is synchronous. In the implementation, every thread is pinged to a core and threads are classified based on the numa region they are operating on. As can be seen in Figure 12.6, horizontally we order *workitems* temporally and vertically we order *workitems* spatially. Hence, the framework is able to perform spatial and temporal orderings on *workitems*.

The primitive execution unit in the framework is a thread. When a thread receives a *workitem*, it is pushed into a data structure. However, the data structure is more complicated than the data structure we used in Section 12.1. The data structure has nested levels for each spatial domain. There is a set of equivalence classes for each memory level in the hierarchy. An equivalence class for a memory level includes a set of equivalence classes that correspond to the child memory level in the hierarchy. The structure of the nested data structure is shown in Figure 12.7. All the *workitems* are stored at the thread level. Processing an equivalence corresponding to an upper memory level requires processing all equivalence classes corresponding to lower levels in the memory hierarchy. For example, when processing an equivalence class at numa level, the framework processes all the

equivalence classes in thread level under that numa domain. As per Figure 12.6, processing an equivalence class in n0, NUMA0 requires processing all the equivalence classes in threads T0, T1, T2. After processing an equivalence class in an intermediate memory level, the framework synchronizes the processing threads that belong to the intermediate memory level. For example, after processing an equivalence class in numa domain, all threads that are pinged to the cores in that numa domain are synchronized. As per Figure 12.6, threads T0, T1 and T3 are synchronized after processing an equivalence class in numa0. Then, the framework removes the currently processing equivalence class from the data structure and moves to the next equivalence class. When global ordering is defined, the synchronization takes place at the global level. After processing a global equivalence class, all processing threads (in-node and distributed) are synchronized.

The data structure depicted in Figure 12.7 is built at compile time using C++ template meta-programming. The framework uses *ordering_traits* structure (Listing 12.1) to create a spatial data structure at compile time. The `ordering_traits` structure takes ordering defined for each memory level as a type parameter. In addition, it also takes an *ordering_config* structure and an implementation of the runtime. The framework does not rely on a particular runtime, rather the runtime is abstract out in the implementation. Any runtime implementation can adopt the framework by implementing functions in the abstract runtime. The structure is statically constructed to store equivalence classes in each level.

LISTING 12.1. Code for static construction of the data structure to hold spatial equivalence classes

```
1 template<typename GlobalOrdering,
2          typename NodeOrdering,
3          typename NumaOrdering,
4          typename ThreadOrdering,
5          typename EAGMConfig,
6          typename Runtime>
7 struct ordering_traits {
8   <global equivalence classes structure>
9   <node equivalence classes structure>
10  <numa equivalence classes structure>
11  <thread equivalence classes structure>
12 };
```

12.1.1. EAGM Schedulers. The AGM(Chapter 11) framework, always selects the next smallest (as per the strict weak ordering relation) equivalence class after processing an equivalence class. However, in EAGM, an equivalence class at a lower spatial level (e.g., Process, Numa or Thread) can get *workitems* to previously processed equivalence classes. This is because there is no guarantee that all ranks have finished processing the equivalence class at the same time. While there are multiple ways to handle this, the framework provides two approaches to execute EAGM equivalence classes:

(1) While the algorithm is not terminated, process current equivalence class. Then, when current equivalence class is empty, move to the next equivalence class. Repeat this process until the algorithm reaches the end of the equivalence class data structure. If the algorithm is not yet terminated, move to the beginning of the data structure that holds equivalence classes.

(2) While the algorithm is not terminated, process current equivalence class. When the current equivalence class is empty and if the algorithm is not terminated, select the smallest non-empty equivalence class from the data structure that holds equivalence classes and process it.

As per our initial results, the second approach in processing equivalence classes showed better performance than the first approach since the second approach was able to reduce *Invalidated Work* work compared to the first approach.

LISTING 12.2. AGM

```
1 // SSSP (AGM) algorithm
2 typedef agm<Graph,
3           WorkItem,
4           ProcessingFunction,
5           StrictWeakOrdering,
6           RuntimeModelGen>
               sssp_agm_t;

8 sssp_agm_t
      ssspalgo(rtmodelgen,
9                ordering,
10               pf,
11               initial);
```

LISTING 12.3. EAGM

```
1 // SSSP (EAGM) algorithm
2 typedef eagm<Graph,
3           WorkItem,
4           ProcessingFunction,
5           EAGMConfig,
6           RuntimeModelGen>
               sssp_agm_t;

8 sssp_agm_t
      ssspalgo(rtmodelgen,
9                config,
10               pf,
11               initial);
```

The AGM framework takes a processing function and a strict weak ordering as input and executes the algorithm. The EAGM framework takes a list of orderings that are annotated to each spatial level and a processing function and executes the algorithm (See Listing 12.2 and Listing 12.3). The orderings relevant to each spatial levels are specified using an *EAGMConfig*.

An example of how an EAGMConfig is created is given in Listing 12.4. The `CHAOTIC_ORDERING_T`, `DELTA_ORDERING_T`, and `DIJKSTRA_ORDERING_T` are ordering functors defined similarly to the functor in Listing 11.4. The EAGM invoked with the configuration in this example Listing 12.4, creates the execution ordering $<_{ch} \rightarrow <_{\Delta(5)} \rightarrow <_{ch} \rightarrow <_{dj}$.

LISTING 12.4. EAGMConfig instantiation.

```
1 CHAOTIC_ORDERING_T ch;
2 DELTA_ORDERING_T delta(agm_params.delta);
3 DIJKSTRA_ORDERING_T dj;
4 auto config = boost::graph::agm::create_eagm_config(ch,
5                                                     delta,
6                                                     ch,
7                                                     dj);
```

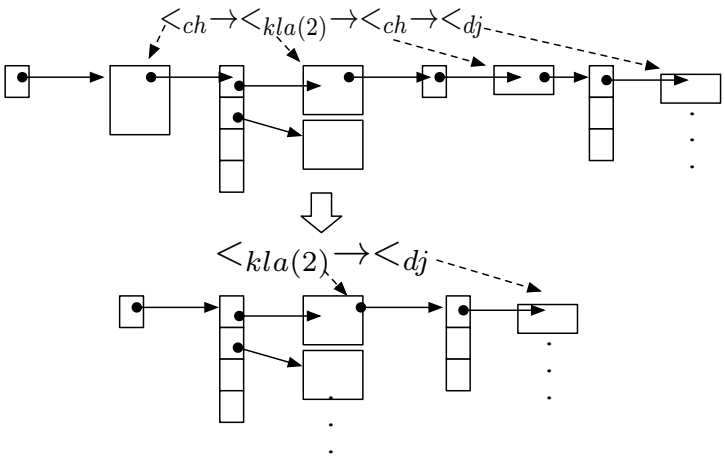


FIGURE 12.8. An example of optimizing spatial orderings.

12.3. Optimizations

The framework treats `CHAOTIC_ORDERING` as a special ordering. In `CHAOTIC_ORDERING` any given two *workitems* are not comparable to each other. Therefore, `CHAOTIC_ORDERING` ordering creates a single large equivalence class. When there is a `CHAOTIC_ORDERING`, ordering defined for a spatial level, we do not need to order *workitems* at that spatial level. Therefore, when a spatial level is defined with a `CHAOTIC_ORDERING`, ordering we can skip those ordering levels in the data structure shown in Figure 12.7.

The ordering configuration $<_{ch} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{ch}$ represents a complete asynchronous algorithm and does not perform any ordering at any spatial level. Therefore, *workitems* reaching a *Rank* in this configuration are not pushed into the EAGM data structure. Instead they are immediately processed with the processing function. The new work generated is also not pushed into the EAGM data structure. Instead it is directly sent to the destination rank.

Figure 12.8 shows how the framework optimizes the data structure relevant to ordering configuration, $<_{ch} \rightarrow <_{kla(2)} \rightarrow <_{ch} \rightarrow <_{dj}$. In the optimized version, the framework does not create equivalence class data structures for memory levels that have `CHAOTIC_ORDERING`. When a *workitem* is pushed into the data structure, it finds the appropriate equivalence class based on $<_{kla(2)}$. Within that equivalence class it is then inserted to the equivalence class in thread level (defined by $<_{dj}$).

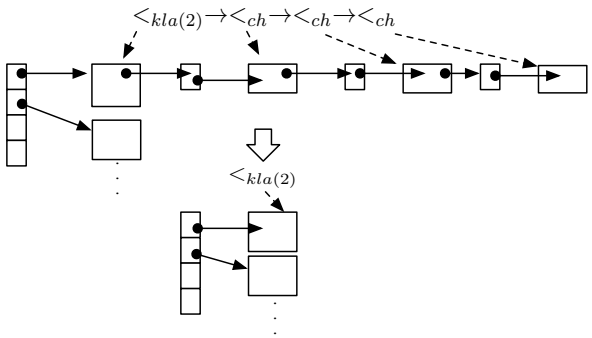


FIGURE 12.9. An example of optimizing spatial orderings.

Another example of ordering optimization is shown in Figure 12.9. In this case, there is a global ordering defined. However, all other orderings are `CHAOTIC_ORDERING`. Therefore, the framework statically generates a data structure that handles equivalence classes at the global level and skips all other levels. The generated data structure is local to a node and thus *workitems* may be pushed into the data structure by multiple threads. Therefore, to store *workitems* pushed by multiple threads we use a concurrent append data structure. This ordering configuration is equivalent to an ordering definition without spatial levels (Section 11.5).

If an ordering configuration specifies `CHAOTIC_ORDERING` for all memory levels, the framework avoids creating the data structure. The *workitems* are not pushed into a data structure, instead, they are sent over the network to the recipient. If all orderings are `CHAOTIC_ORDERING`, that represents a pure asynchronous algorithm.

In addition to `CHAOTIC_ORDERING` related optimizations, we also convert buffers to priority queues when there is an ordering defined at the thread-level (Figure 12.10). At the thread level, we do not need to maintain a concurrent data structure as the *workitems* in the data structure are not shared by multiple threads. Therefore, sequential priority queues are more suitable for thread level orderings.

All these optimizations are applied statically at compile time using template specialization. For example, when an ordering configuration similar to Figure 12.8 is specified, the template specialization in Listing 12.5 is invoked at compile time. The structure in

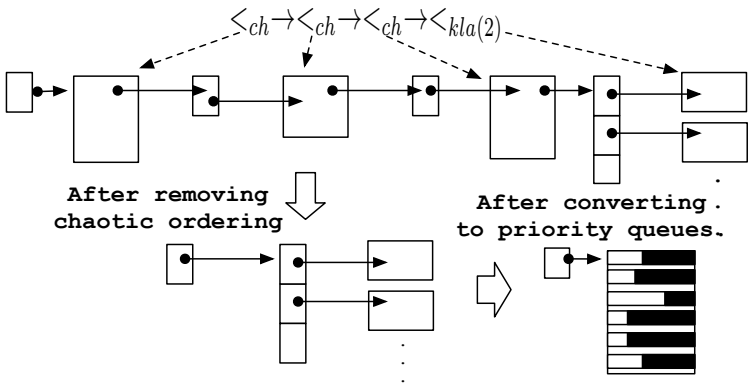


FIGURE 12.10. An example of optimizing spatial orderings.

Listing 12.5 only constructs the structure to hold node level equivalence classes, unlike the general definition of `ordering_traits` in Listing 12.1.

LISTING 12.5. Template specialization for ordering configurations that define node level ordering.

```
template<typename GlobalOrdering,
         typename NodeOrdering,
         typename NumaOrdering,
         typename ThreadOrdering,
         typename EAGMConfig,
         typename Runtime>
struct ordering_traits<CHAOTIC_ORDERING,
                       NodeOrdering,
                       CHAOTIC_ORDERING,
                       CHAOTIC_ORDERING> {
  // only constructing data structure
  // to hold node equivalence classes
  <node equivalence classes structure>
};
```

12.4. More Usecases

In the following, we present a few more examples of ordering configurations and how their execution is taking place.

Figure 12.11 shows an execution of an asynchronous algorithm. There is a single large equivalence class created and barriers are executed before and after processing.

If an algorithm only specifies the global ordering (e.g., $<_{\Delta(5)} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{ch}$), then its execution is equivalent to AGM algorithm execution (Figure 11.14).

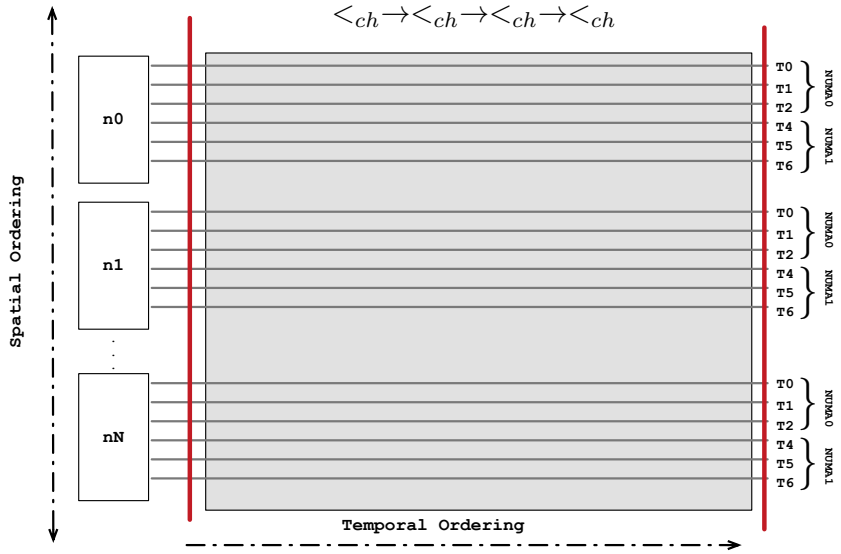


FIGURE 12.11. Complete asynchronous algorithm.

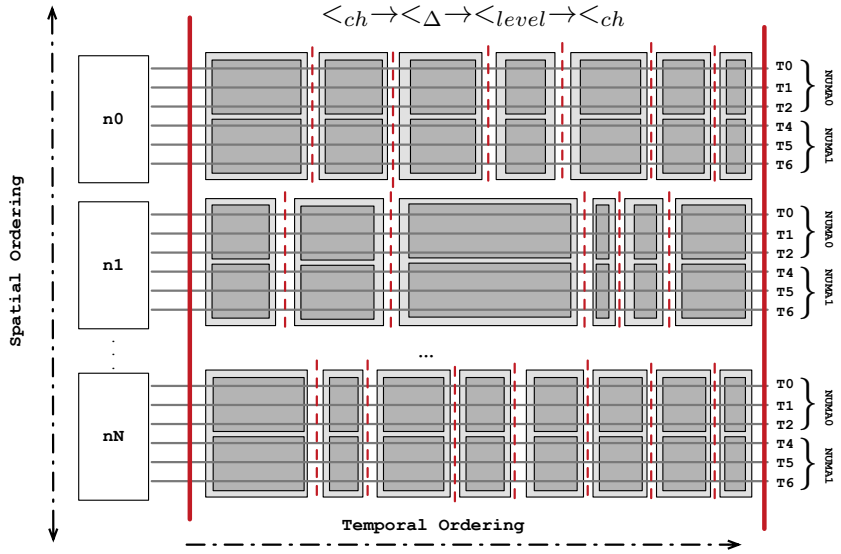


FIGURE 12.12. Globally asynchronous, but process and numa ordered execution.

Figure 12.12 shows an algorithm that is globally asynchronous, process level Δ ordered and NUMA level ordered.

As shown above, different algorithms can be generated by assigning different orderings to different spatial levels within a memory hierarchy.

12.5. Summary

In this chapter, we discussed how AGM framework is extended to explore spatial orderings. The resulting framework (EAGM framework) enables us to specify orderings at different spatial levels. The heart of the EAGM framework is the nested data structure that manages *workitems* in each spatial levels.

The chaotic ordering is treated with a special significance and is used to optimize ordering specified in the EAGM config. With EAGM, we achieve two-dimensional ordering. Horizontally, we achieve ordering by separating *workitems* into equivalence classes and vertically, we achieve asynchronous execution at different spatial levels.

13

Breadth First Search

Breadth First Search (BFS) is a straightforward graph application that was discussed in this thesis several times (e.g., Section 2.2.1). In this chapter, we present pre-order, post-order and split-order processing functions for BFS and performance results for several Extended Abstract Graph Machine (EAGM) algorithms.

13.1. Pre-order BFS

BFS checks the reach-ability from a given source vertex to other vertices. The processing function for BFS is given in Listing 13.1 as a C++ functor. For BFS a *workitem* is defined as a tuple of an edge and the level value (Line 1). An edge is represented as a source vertex and a destination vertex. The state *levelst* maintains the lowest level a vertex can be reached from the source vertex and parentst maintains the parent vertex of a vertex in the BFS tree. The π compares the level in the *workitem* with the level stored in *levelst* and if the incoming *workitem*'s level smaller the state is updated. The update is performed using Compare And Swap (CAS) (Line 8) operation since there can be multiple shared-memory threads trying to update the same value. The CAS operation returns *true* upon successful update of the level state. If current processing thread is able to update the distance, it will generate new work to neighboring vertices (Line 13). These newly generated *workitems* are pushed into the data structure for ordering.

LISTING 13.1. Processing function for BFS

```
1 typedef std::tuple<Vertex, Vertex, Level> WorkItem;
2 struct bfs_pf {
```

```
3 void operator()(const WorkItem& wi, int tid, buckets& outset) {
4   Vertex v = std::get<0>(wi);
5   Vertex pv = std::get<1>(wi);
6   int level = std::get<2>(wi);

8   if(CAS(level, levelst[v])) {
9    parentst[v] = pv;
10   for_each(Edge e: out_edges(v)) {
11    Vertex u = target(e);
12    WorkItem generated(u, v, (level+1));
13    outset.push(generated, tid);
14   }
15  }
16 };
```

The BFS algorithm uses two states (*parentst* and *levelst*). These two states are initialized as follows: parentst[v] $\leftarrow v \forall v \in V$ and levelst[v] $\leftarrow \infty \forall v \in V - \{s\}$ and levelst[s] $\leftarrow 0$. The vertex s is the source. Initial *workitems* are generated for neighbors of source vertex and populated into the data structure for ordering (Algorithm 25).

Algorithm 25 The initial *workitem* generation for pre-order BFS.

Initialize *Vertex s* :

```
1: for Edge e: out_edges(s) do
2:    Vertex u = target(e);
3:    WorkItem generated(u, s, 1);
4:    outset.push(generated);
5: end for
```

13.2. Post-order BFS

The post-order processing functions are similar to pre-order processing functions, but instead of inserting *workitems* to the data structure they are sent over the network to a remote rank. For example, the *outset.push* call in Listing 13.1 (Line 13) is replaced with *Send(generated)*. The *Send* function is responsible for sending *workitem* to the appropriate destination rank based on the data distribution.

13.3. Split-order BFS

The state update functor for BFS is given in Listing 13.2. The logic remains same as Listing 13.1 except that Listing 13.2 only do the state update. New work is generated in

Listing 13.3. Note that π_{su} is inserting *workitem* to the data structure (Line 10) whereas π_{gen} is sending newly generated *workitems* to remote ranks (Line 10).

LISTING 13.2. State update function for BFS

```
1 typedef std::tuple<Vertex, Vertex, Level> WorkItem;
2 struct state_update_bfs_pf {
3 void operator()(const WorkItem& wi, int tid, buckets& outset) {
4      Vertex v = std::get<0>(wi);
5      Vertex pv = std::get<1>(wi);
6      Level l = std::get<2>(wi);
7      if(CAS(l, levelst[v])) {
8        if (l == levelst[v]) {
9          parentst[v] = pv;
10         outset.push(wi, tid);
11       }
12     }
13   }
14 };
```

LISTING 13.3. Work generating function for BFS

```
1 struct new_work_gen_bfs_pf {
2 void operator()(const WorkItem& wi, int tid, buckets& outset) {
3      Vertex v = std::get<0>(wi);
4      Vertex pv = std::get<1>(wi);
5      Level l = std::get<2>(wi);
6      if(l == levelst[v]) {
7        for_each(Edge e : out_edges(v)) {
8          Vertex u = target(e);
9          WorkItem w(u, v, (l+1));
10         Send(w, tid);
11       }
12     }
13   }
14 };
```

13.4. Orderings

We evaluate the performance of BFS for the following orderings:

(1) $<_{level} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{ch}$

(2) $<_{kla(2)} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{ch}$

(3) $<_{ch} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{level}$

The first two ordering configurations applied at the global level where the third configuration is globally asynchronous and ordering is performed at the thread level. The functor definitions for $<_{level}$ and $<_{kla(2)}$ are given in Listing 13.4 and in Listing 13.5. The *index* template parameter specifies how to query the level attribute from a *workitem* (recall that a *workitem* is defined as a tuple and "index" specifies the location of the level attribute inside the tuple).

The $<_{level}$ puts two *workitems* to the same equivalence class if they have the same level and $<_{kla(2)}$ inserts two *workitems* to the same equivalence class if their levels within $[nk, (n+1)k)$ range, where $n \in \{0, 1, \dots\}$.

LISTING 13.4. The definition of $<_{level}$

```
template<int index>
struct level {
public:
  template <typename T>
  bool operator()(T i, T j) {
    return (std::get<index>(i) < std::get<index>(j));
  }
};
```

LISTING 13.5. The definition of $<_{kla(2)}$

```
template<int index>
struct klevel {
private:
  int k;

public:
  klevel(int _k):k(_k){}

  template <typename T>
  bool operator()(T i, T j) {
    return ((std::get<index>(i)/k) < (std::get<index>(j)/k));
  }
};
```

13.5. Experimental Evaluations

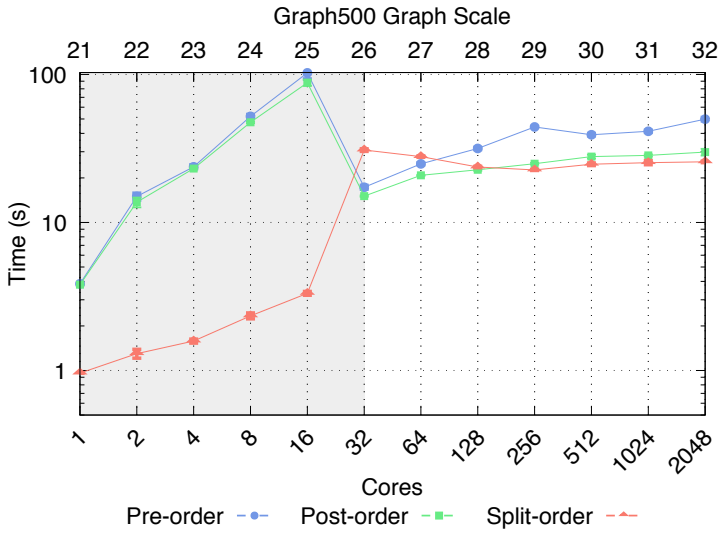


FIGURE 13.1. A comparison of pre-order, post-order and split-order execution configurations for BFS.

13.5.1. Processing Function Execution. In this section, we experimentally evaluate the performance of different processing function execution configurations for $<_{level} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{ch}$. These experiments were carried out on a Cray XC system that has 2 Broadwell 22-core Intel Xeon processors. Our experiments only used up to 16 cores to uniformly to double the problem size and to double the number of processors in weak scaling. Each node consists of 128 GB DDR4-2400 memory. We use an MPI+PThread, distributed shared-memory runtime. The MPI implementation is Cray MPICH (version 7.4.4). The input is Graph500 **[104]** graphs from scale 21–32.

The results are shown in Figure 13.1. As per results the pre-order and post-order configurations show poor performance in shared memory execution (when cores < 32, execution is shared memory and shaded region shows the shared-memory execution). This is mainly due to high contention created by pre-order and post-order execution configurations. When execution is distributed, the post-order shows slightly better performance than pre-order, but when the execution scales to many nodes we see that pre-order and post-order performance degrading compared to split-order.

13.5.2. Weak Scaling. For weak scaling experiments, we use *R-MAT* [**23**] synthetic graphs. Weak scaling of above discussed algorithms are evaluated on three types of synthetic graphs. They are:

(1) RMAT-1: Graphs based on the current Graph500 [**104**] Breadth First Search benchmark specification with R-MAT parameters $A = 0.57$, $B = C = 0.19$ and $D = 0.05$,

(2) RMAT-2: Graphs generated based on the proposed Graph500 [**56**] SSSP benchmark specification with R-MAT parameters $A = 0.50$, $B = C = 0.1$ and $D = 0.3$.

(3) ER: This is the Erdos-Renyi [**42**] random graphs.

Our initial experiments showed that the ordering $<_{ch} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{level}$ is remarkably slow compared to other orderings. This is mainly because of the *self-sending* (Section 10.1.2.2) of *workitems*. Recall that when self-sending is enabled, *workitems* destined for current *Rank* are not routed through the network instead a function call is made to the processing function. In Single-Source Shortest Paths (SSSP), with split-order configuration, a *workitem* is pushed into the data structure and that *workitem* is picked up by a thread and executes the π_{gen} function. When self-sending is enabled, the new work generated by π_{gen} function is inserted into the same thread's priority queue. Therefore, work is always pushed into a single thread priority queue irrespective of the number of parallel threads being executed.

There are two ways to overcome above-discussed performance issue: 1. disable self-sending and route all the messages through the network, 2. run one process per core so that, for some portion of the work, processing function is executed through the stack and for the rest, messages are sent over the network.

By disabling self-sending we can utilize all the threads for processing *workitems* but then we need to pay the cost of sending all messages through Message Passing Interface (MPI) buffers and lower runtime layers. However, in distributed execution, the self-sending avoid the overhead of sending a message through those MPI buffers and load imbalance between in-node threads is also reduced due to distributed execution. Therefore, in distributed execution, the self-sending improves the performance.

Cores	1 process and multiple threads per process	multiple processes and 1 thread per process
1	1.21	1.21
2	2.9	1.24
4	7.18	2.32
8	18.37	3.98
16	48.45	4.96

TABLE 13.1. Shared-memory results for BFS with two different process execution configurations.

Our initial experiments showed that running a process per core (for shared-memory) gives better performance compared to the first approach. For example, Table 13.1 shows the shared-memory performance of $<_{ch} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{level}$ ordering with 1 process per core and 1 process for 16 threads. As explained above when we execute a process and 16 threads with self-sending, execution takes place in a single thread. When we use multiple processes the in-node load imbalance is alleviated.

The weak scaling results for algorithms are given in Figure 13.2. In addition to EAGM orderings discussed above, we also included results from Parallel Boost Graph Library, version 2 (Parallel BGLv2) [**38**], BFS algorithm on Graph500 input (The first plot in Figure 13.2). The Parallel BGLv2 BFS algorithm uses a distributed queue and colors to further avoid any redundant work. Therefore, Parallel BGLv2 BFS performs better than other algorithms in shared-memory and EAGM algorithms show competitive performance in distributed memory.

For all three graph inputs the level synchronous equivalent BFS ($<_{level} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{ch}$) is faster than other orderings. For ER graphs the $<_{ch} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{level}$ is much slower than other two algorithms. ER graphs have a uniform degree distribution and a low diameter (e.g., scale 30 graphs have about 8 levels), therefore the first and second orderings are able to eliminate much of the redundant work and the synchronization overhead is also minimized. However, the third ordering performs more invalidated work than first two orderings, hence its performance is poor compared to other two orderings.

Certain orderings coincide with each other depending upon the spatial memory boundary they are executing on. For example, $<_{level} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{ch}$ and $<_{ch} \rightarrow <_{level} \rightarrow <_{ch} \rightarrow <_{ch}$ has the same performance in shared-memory execution (Table 13.2). $<_{level} \rightarrow <_{ch} \rightarrow <_{ch}$

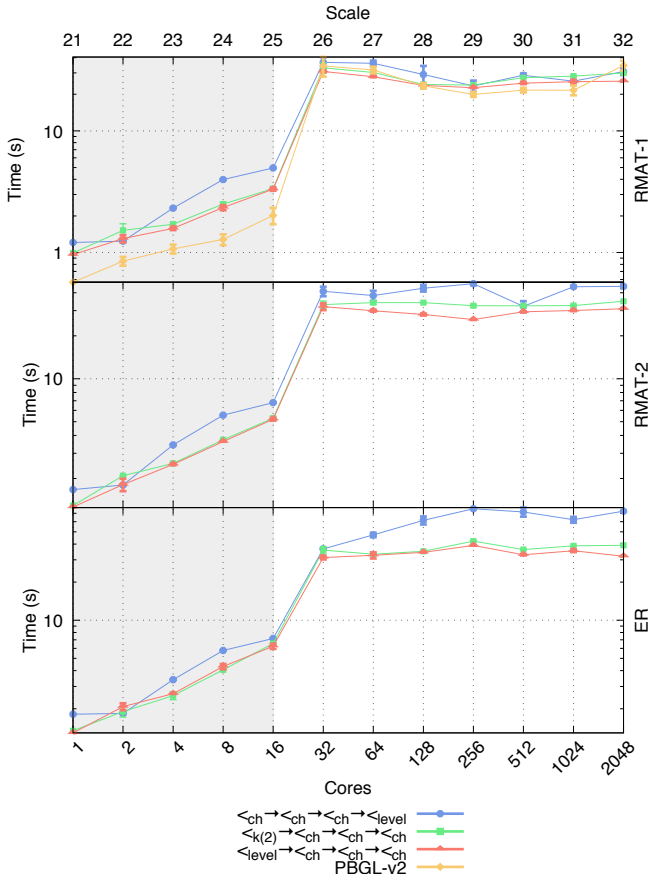


FIGURE 13.2. Weak scaling results for BFS orderings.

Cores	$<_{level} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{ch}$	$<_{ch} \rightarrow <_{level} \rightarrow <_{ch} \rightarrow <_{ch}$
1	0.96	0.85
2	1.3	1.06
4	1.58	1.59
8	2.34	2.11
16	3.32	3.19

TABLE 13.2. Two different level orderings showing similar performance in-node.

$\rightarrow <_{ch}$ has a global barrier after processing an equivalence class and $<_{ch} \rightarrow <_{level} \rightarrow <_{ch} \rightarrow <_{ch}$ synchronizes all the thread under the process, after processing an equivalence class. When algorithms are executed in a single node, the global barrier is equivalent to the thread barrier. Because of that, both above configurations show similar performance.

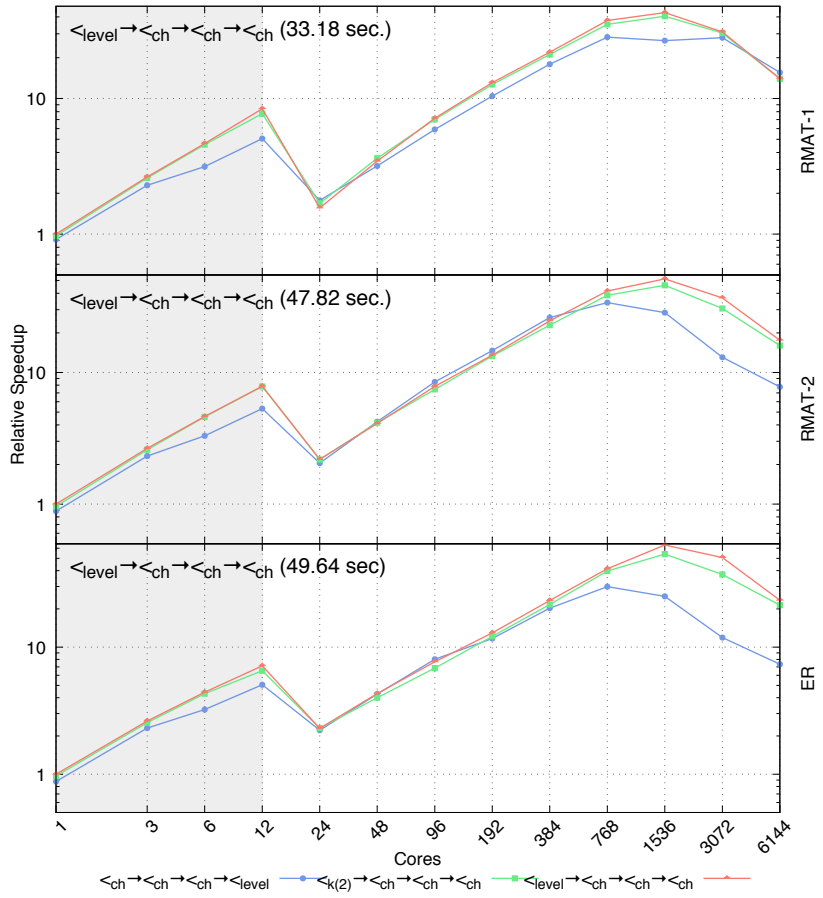


FIGURE 13.3. Strong scaling results for BFS orderings. Plots show the relative speed-up. The fastest sequential algorithm is shown on the plot with the timing.

13.5.3. Strong Scaling. Strong scaling experiments were carried out on a Cray XC30 supercomputer. Each node in the system has two Intel Xeon processors and each processor has 12 cpus (cores). Each node has 64 GB of DDR3 RAM. The system uses Aries interconnect.

Speed-up results for scale 25, RMAT-1, RMAT-2 and ER graphs are shown in Figure 13.3. To gain a better understanding about how algorithms scale relative to each other, we measured *Relative Speedup*, $= \frac{T_{ref\_1}}{T_n}$ i.e., the ratio of the execution time of the fastest sequential algorithm, T_{ref_1} and the parallel execution time on n processing elements, T_n.

Nodes	$<_{level} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{ch}$	$<_{ch} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{level}$
1	1.7	2.67
2	2.47	1.17
4	3.88	0.83
8	6.06	0.88
16	8.82	0.72
32	12.01	0.52
64	15.45	0.42
128	21.19	0.38

TABLE 13.3. Timing results to process CA road [**84**] network graph on distributed nodes for two algorithms (Time in seconds).

In distributed execution, all synthetic graphs show similar speedups for all the algorithms until about 768 cores. Afterward, the amount of parallelism reduces and speed-up also decreases.

13.5.4. High Diameter Graphs. Table 13.3 shows timing results to process road network graph in distributed nodes. As can be seen, the global asynchronous EAGM algorithm runs faster than global level ordered algorithm. In fact, global level ordering performance decreases with the increasing number of nodes.

Road networks have very high diameters compared to other graphs (e.g., CA road network diameter is 830 and a selected random source in average visits at least 540 levels). Therefore, a globally synchronous algorithm may perform at least 540 barriers. Because of the overhead of barrier synchronization, the globally level ordered algorithm shows poor performance in distributed execution.

14

Single Source Shortest Paths

Single-Source Shortest Paths (SSSP) is a seemingly simple problem where the task is to find the shortest path from a source vertex s to every other vertex in the graph. A number of sequential algorithms exist. The well-known Dijkstra's algorithm [**34**] is the "work optimal", where vertices are ordered in a priority queue based on their distance from the source s, and every edge is traversed only once. Work optimality, however, comes at a cost of limited parallelism and extensive synchronization. Subsequent development focused on relaxing the strict ordering of the Dijkstra algorithm is to make more work available in parallel at the cost of some "wasted work" that has to be invalidated and repeated. For example, the Δ-Stepping [**100**] algorithm groups vertices into Δ-sized *buckets* based on their distances from the source s, giving an approximation of Dijkstra ordering. Vertices in a bucket are processed in parallel. Picking an appropriate Δ ensures the right balance between parallelism and wasted work. The *K-Level Asynchronous* [**64**] algorithm is similar, but it uses topological distances instead of shortest path distances from the source s to order work into buckets[1].

In this chapter, we show how these different approaches are implemented with a processing function and an ordering using the Abstract Graph Machine (AGM) framework.

14.1. Pre-order SSSP

The SSSP application finds the minimum distance to every vertex from a given source vertex. The pre-order processing function for SSSP is given in Listing 14.1.

[1] K-Level Asynchronous with single-hop buckets is equivalent to the Bellman-Ford algorithm [**14**].

An SSSP *workitem* is defined as a vertex and a distance. This definition of the *workitem* can be used to order work according to distinct distance values (e.g., Dijkstra's algorithm) and Δ range buckets. The SSSP algorithm uses *vdistance* state to maintain the minimum distance to a vertex from the source vertex. The distance in the incoming *workitem* is compared against the stored distance in the state. The distance in the state variable is updated if the *workitem* contains a smaller distance (Line 8). New work is generated for neighbors if the distance for a vertex is updated (Line 9). The variable *weight* is a property that contains the weight of each edge.

LISTING 14.1. Processing function for SSSP

```
1 typedef std::tuple<Vertex, Distance> WorkItem;
2 struct sssp_pf {
3 void operator()(const WorkItem& wi, int tid, buckets& outset) {
4  Vertex v = std::get<0>(wi);
5  Distance d = std::get<1>(wi);
6  Distance old_dist = vdistance[v], last_old_dist;

8  if(CAS(d, vdistance[v])) {
9   for_each(Edge e : out_edges(v)) {
10   Vertex u = target(e);
11   WorkItem w(u, (d+weight(e)));
12   outset.push(w, tid);
13  }
14 };
```

The *vdistance* state is initialized similar to *levelst* in Breadth First Search (BFS) (Section 13.1), i.e., vdistance[v] $\leftarrow \infty \forall v \in V - \{s\}$ and vdistance[v]$\leftarrow 0$. The initial *WorkItems* are calculated according to Algorithm 26.

Algorithm 26 The initial *workitem* generation for pre-order SSSP.

Initialize *Vertex s* :

```
1: for Edge e: out_edges(s) do
2:    Vertex u = target(e);
3:    WorkItem generated(u, weight(e));
4:    outset.push(generated);
5: end for
```

14.2. Post-order SSSP

In post-order, the initial state values and initial *workitems* are generated differently than in pre-order. This is because SSSP *vdistance*(Listing 14.1) is initialized to ∞ for all vertices (including the source vertex) and a *workitem* with source vertex and 0 distance is pushed into the data structure. Since the processing function is executed on the same rank for *workitems* in the data structure, source vertex states are updated and new work is generated for neighbors in the first execution of the processing.

14.3. Split-order SSSP

Listing 14.2 shows the π_{su} functor for SSSP. It takes a *workitem* and updates the distance state. Since multiple threads may access the distance state, the state update is carried out using an atomic Compare And Swap (CAS) operation wherein the thread that updates the smallest distance will succeed. CAS returns *true* if distance state is updated and if distance state is changed. The *workitem* is then pushed into the data structure that performs ordering.

LISTING 14.2. State update function for SSSP

```
1 struct state_update_sssp_pf {
2 void operator()(const WorkItem& wi, int tid, buckets& outset) {
3     Vertex v = std::get<0>(wi);
4     Distance d = std::get<1>(wi);
5     if(CAS(d, distance_state[v])) {
6        outset.push(wi, tid);
7     }
8  }
9 };
```

Listing 14.3 shows the new work generation functor for SSSP. The functor compares the distance associated with *workitem* and the distance stored in the distance state. If they are equal *workitems* are generated for neighbors of *v*.

LISTING 14.3. New work generation for SSSP

```
1 struct new_work_gen_sssp_pf {
2 void operator()(const WorkItem& wi, int tid, buckets& outset) {
3     Vertex v = std::get<0>(wi);
```

```
4       Distance d = std::get<1>(wi);
5       if (d == distance_state[v]) {
6         for_each(Edge e : out_edges(v)) {
7           Vertex u = target(e);
8           WorkItem w(u, (d+weight(e)));
9           Send(w, tid);
10        }
11      }
12    }
13 };
```

14.4. Orderings

We evaluate the performance of BFS for the following orderings:

(1) $<_{\Delta(5)} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{ch}$

(2) $<_{dj} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{ch}$

(3) $<_{kla(2)} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{ch}$

(4) $<_{kla(2)} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{dj}$

(5) $<_{ch} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{dj}$

The first ordering separates work into Δ range buckets and buckets that are globally ordered. The second ordering is similar to Dijkstra's algorithm but differs in that equal distance *workitems* are inserted into the same bucket. The third and fourth orderings use a different definition of a *workitem*. This new definition includes the level visited in addition to vertex and distance. The third configuration globally orders work by level and the fourth configuration globally orders work by level and by distance at the thread level. The final configuration is globally asynchronous but orders work by the distance at the thread level.

The ordering functor for $<_{\Delta(5)}$ is given in Listing 14.4. The definition for $<_{dj}$ and $<_{kla(2)}$ was given in Listing 11.4 and Listing 13.5.

LISTING 14.4. Delta ordering functor.

```
1 template<int index>
2 struct delta {
3 private:
4   int delta;
```

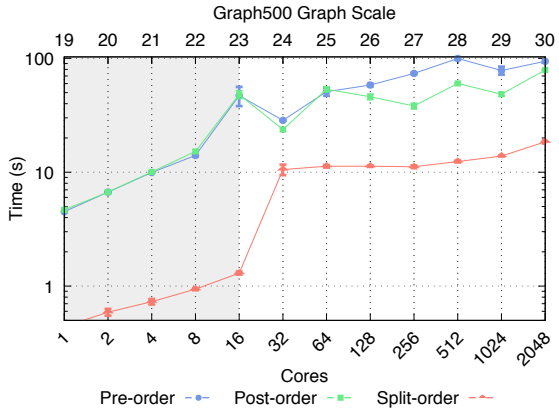


FIGURE 14.1. A comparison of pre-order, post-order and split-order execution configurations for SSSP.

```
5 public:
6   delta_ord(int _d) : delta(_d) {}

8   template <typename T>
9   bool operator()(T i, T j) {
10    return ((std::get<index>(i)/delta) <
          (std::get<index>(j)/delta));
11  }
12 };
```

14.5. Experimental Evaluations

14.5.1. Processing Function Execution. We experimentally evaluated the performance of different processing function execution configurations for $<_{\Delta(5)} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{ch}$. These experiments were carried out in the same environment in which we carried out BFS processing function experiments (Section 13.5.1). The input is Graph500 **[104]** graphs from scale 19–30.

The split-order configuration performed ≈ 5 times faster than post-order configuration (See Figure 14.1). Further, the post-order configuration was faster than pre-order and the performance difference between pre-order and post-order was more visible at higher scales.

The split-order configuration is faster because it eliminates the most amount of redundant work. Since post-order distributes insertions to the data structure, the contention

on the data structure is less compared to pre-order. Hence, pre-order is faster than post-order. Why split-order is faster than pre-order and post-order is discussed in detail in Section 11.4.

14.5.2. Weak Scaling. For weak scaling, we used the same graph inputs we discussed in Section 13.5.2. We compared weak scaling results of different algorithms against three popular graph algorithms: 1. PowerGraph-GraphLab **[90]**, 2. Parallel Boost Graph Library (Parallel BGL) **[39]** 3. Parallel Boost Graph Library, version 2 (Parallel BGLv2) **[38]**.

The first set of weak scaling experiments were carried out on Cray XE6/XK7 nodes, each with 2 AMD Opteron Abu Dhabi CPUs (for a total of 32 cores), and 64 GB of memory per node (4 numa domains, 2 per CPU). The weak scaling results for this set of experiments is shown in Figure 14.2. In distributed execution, the globally asynchronous SSSP with thread ordering performed much faster than other algorithms. As the scale increased Dijkstra's algorithm performance degraded because of the synchronization overhead and less parallelism available for the same distance value. This is because both Parallel BGL and Parallel BGLv2 implement the Δ-Stepping algorithm, but Parallel BGL uses a distributed data structure. Also, the execution is completely distributed. Parallel BGLv2 uses active messages and a hybrid execution, similar to the Extended Abstract Graph Machine (EAGM) framework. EAGM framework Δ ordering showed better scaling and better performance in distributed execution.

Unlike BFS, SSSP is a metric based (i.e., distance) algorithm. Ordering based on distance can avoid more work in SSSP than ordering based on the level. Therefore, the ordering $<_{kla(2)} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{ch}$ shows higher timing values than other orderings. The execution time for $<_{kla(2)} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{ch}$ is improved in $<_{kla(2)} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{dj}$ because of the distance ordering at thread level.

Both PowerGraph and Parallel BGLv2 do not scale very well in distributed execution; especially when processing RMAT-2 synthetic graphs. PowerGraph implements SSSP using Gather-Apply-Scatter (GAS) primitives and there is global synchronization between

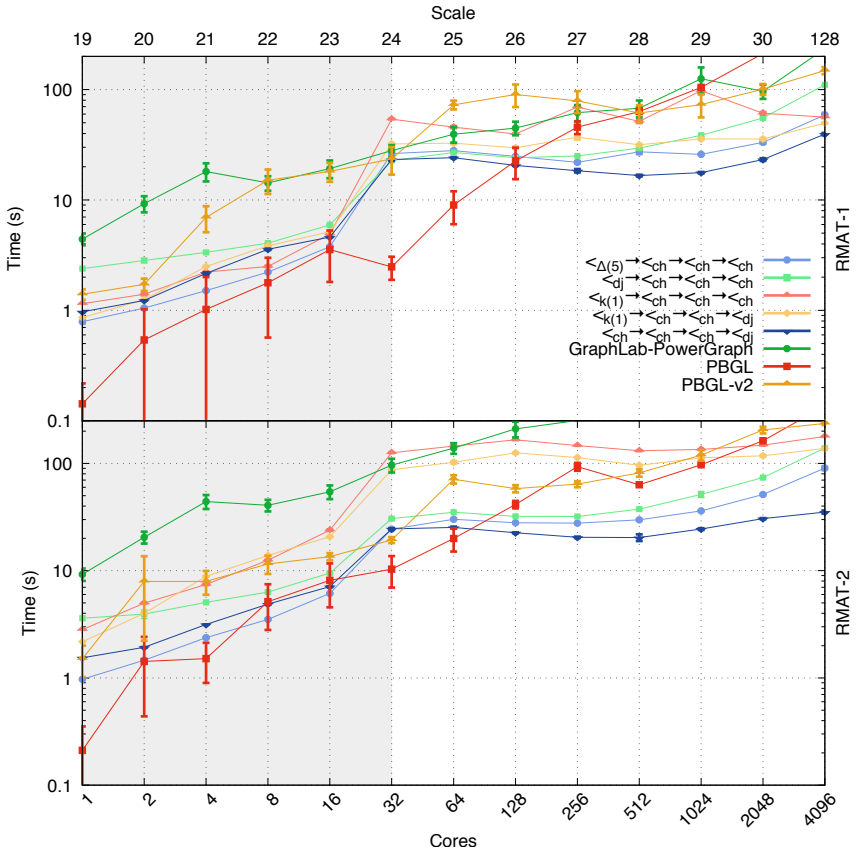


FIGURE 14.2. Weak scaling results for SSSP orderings and a comparison with other graph processing systems.

Gather and Apply phases. There is also global synchronization between Apply and Scatter phases. Further, PowerGraph does not perform ordering based on distance. Therefore, PowerGraph process more redundant work than other algorithms. PowerGraph also provides a version that does not synchronize between Apply and Scatter phases ("asynchronous engine"). However, our initial results showed that its performance is poorer than that of their synchronous engine (two synchronization phases as discussed above).

Weak scaling results for an architecture that has 2048 cores (this is the same system we used for BFS weak scaling tests) is shown in Figure 14.3. The RMAT-1 and RMAT-2 results show the same trend we saw in Figure 14.2. The RMAT-1 plot also includes results

Algorithm	Inv.	Inv. Cancel	Redundant
$<_{\Delta(5)} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{ch}$	1082711221	1080504309	2206912
$<_{ch} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{dj}$	3796319738	3452704617	343615121

TABLE 14.1. Work statistic comparison for two orderings on ER graphs.

for PowerGraph. EAGM algorithms out-performed PowerGraph performance. In shared-memory results, we see that $<_{\Delta(5)} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{ch}$ gives the best performance for all three graph inputs.

For ER graphs, the $<_{\Delta(5)} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{ch}$ ordering performance is better than other orderings and we see that $<_{ch} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{dj}$ ordering runtime increased. For ER graphs, the algorithm $<_{\Delta(5)} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{ch}$, processed fewer equivalence classes than for RMAT-1 graphs. For example, for Scale 29 on 32 nodes $<_{\Delta(5)} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{ch}$ processed ≈ 134 equivalence classes (the number of equivalence classes is also same the same as the number of global barriers executed). For ER graphs, the same algorithm processed ≈ 85 equivalence classes. Therefore, the synchronization overhead when processing RMAT-1 graphs with $<_{\Delta(5)} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{ch}$ ordering is clearly higher compared to the synchronization overhead when processing ER graphs with the same algorithm. In addition, globally synchronized, distance-based ordering eliminates more redundant work than asynchronous thread orderings when processing ER graphs. See Table 14.1. In this table, the redundant work is equal to (*Invalidated Work- Invalidated Cancel Work*).

ER graphs have a uniform degree distribution. Therefore, the possibility a vertex reaching shortest distance after a visit from its previous level is high compared to power-law graphs such as RMAT-1 and RMAT-2. Hence, for ER graphs, both global Δ ordering and global Dijkstra ordering eliminate more *Invalidated Work* work and process a fewer number of equivalence classes (i.e., fewer number of global barriers).

14.5.3. Strong Scaling. Strong scaling experiments were carried out on a Cray XC30 supercomputer. Each node in the system has two Intel Xeon processors and each processor has 12 CPUs (cores). Each node has 64 GB of DDR3 RAM. The system uses Aries interconnect.

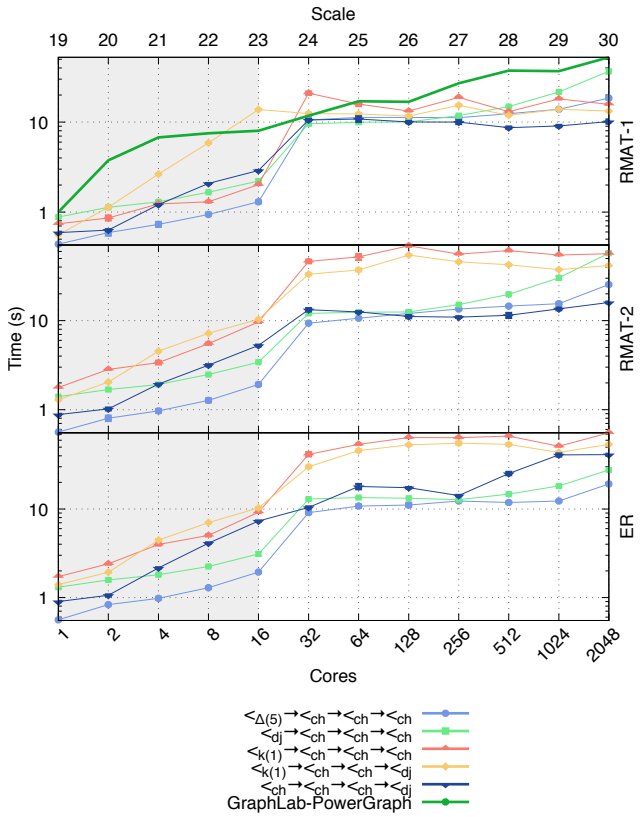


FIGURE 14.3. Weak scaling results for SSSP orderings for experiments ran on a architecture with fewer nodes.

Figure 14.4 shows strong scaling performance of different EAGM orderings for SSSP (relative speedup). At a smaller number of nodes, globally Δ ordered and globally Dijkstra ordered algorithms showed better speedup compared to other orderings. However, as we increased the number of nodes, the speedup decreased due to synchronization overhead of those orderings. Globally chaotic and thread Dijkstra orderings show better speedup in distributed execution. Algorithms achieve maximum parallelism around 1536 cores and afterward, all algorithms show a decrease in their speedups.

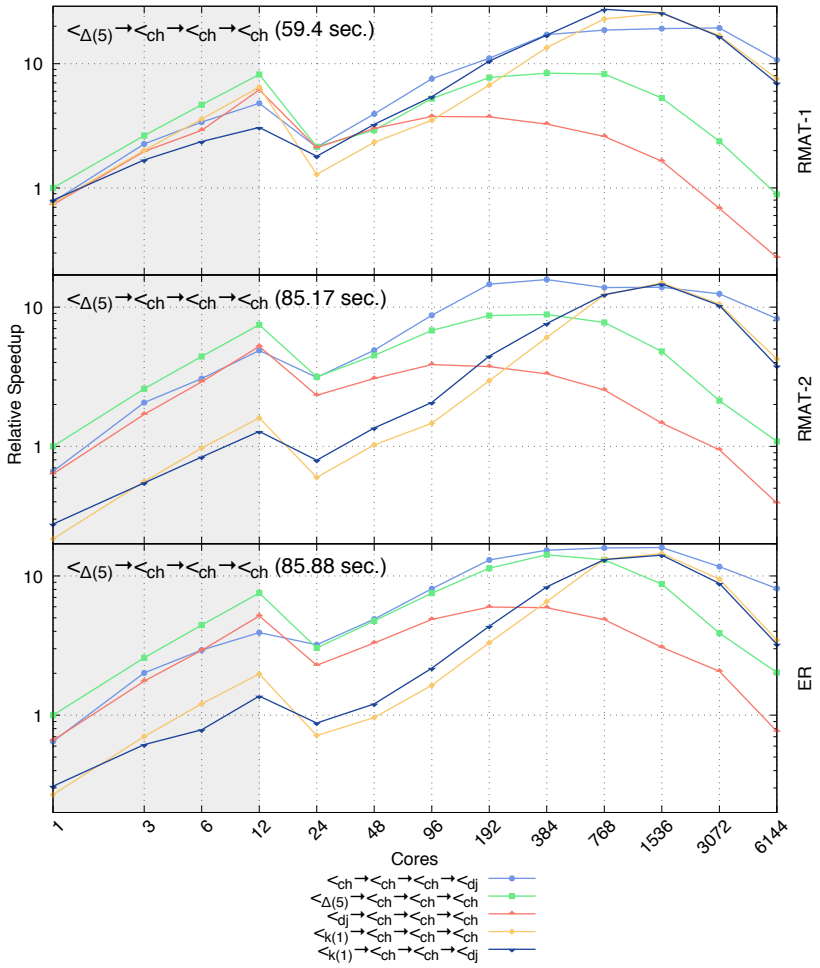


FIGURE 14.4. Strong scaling results for SSSP orderings. Plots show the relative speed-up. The fastest sequential algorithm is shown on the plot with the timing.

15

Connected Components

In this chapter, we show how Connected Components (CC) can be implemented using the Extended Abstract Graph Machine (EAGM) framework we also evaluate its performance for different processing function execution configurations and its strong and weak scaling performance for different graph inputs. The framework implements the CC algorithm discussed in Chapter 6 and evaluates the performance of three different spatial and temporal orderings.

15.1. Pre-order Connected Components

The processing function for CC is presented in Listing 15.1. A unit of work for CC is defined as a pair of vertices and a component id (Line 1). Initially, all vertices are assigned their vertex ids as their component ids. For every vertex, its component id is recorded in the *vcomponent* state which is updated when a *workitem* with a smaller component identifier is received (Line 6). The functor generates new work for successor vertices of the vertex in the incoming *workitem* (Line 10). However, if algorithm detects that the vertex in the *workitem* has a neighbor that is smaller than itself, the algorithm stops propagating state changes to its successors (since the smaller neighbor is going to dominate the component value for all its connecting vertices). The *haslowernbr* flag (Line 16) is set to *true* if the vertex has a neighbor with a vertex id lower than the incoming component id. If there isn't a lower neighbor, work is generated (Line 20).

LISTING 15.1. State update function for CC

```
1 typedef std::tuple<Vertex, Component> WorkItem;
```

```
2 struct cc_pf {
3 void operator()(const WorkItem& wi, int tid, buckets& outset) {
4   Vertex v = std::get<0>(wi);
5   Component component = std::get<1>(wi);
6   if (CAS(&vcomponent[v], component)) {
7     if (component == vcomponent[v])
8       set<Vertex> adjacencies;
9       bool haslowernbr = false;
10      for_each(Edge e : out_edges(v)) {
11       Vertex u = target(e);
12       if (u > component) {
13        adjacencies.insert(u);
14       }else if(u < component) {
15         // v has a lower neighbor
16         haslowernbr = true;
17         break;
18       }
19      }
20      if (!haslowernbr) {
21       for_each(Vertex w : adjacencies) {
22        WorkItem generated(w, component);
23        outset.push(generated, tid);
24       }
25      }
26    }
27   }
28 };
```

Initially, *vcomponent* is initialized to its vertex id (i.e., vcomponent[v] $\leftarrow v$). If a vertex does not have neighbors that are less than itself, we call that vertex a *source*. The algorithm calculates those source vertices and the initial *WorkItems* set is generated for successors of those source vertices. The algorithm to generate initial *workitems* is listed in Algorithm 27.

Algorithm 27 iterates over vertices in parallel (Line 1) and checks whether a vertex is a source. To check whether a vertex is a source, it checks whether there are neighbors less than the vertex. The variable *haslowernbr* is set to true if the vertex has a lower neighbor (Line 7). If *haslowernbr* is false at the end of the loop for iterating neighbors (Line 13), then that vertex is a source and work is generated for its adjacencies (Line 14) and pushed into the data structure.

Algorithm 27 The initial *workitem* generation for pre-order CC.

Initialize :

```
1:  for Vertex v: V in parallel do
2:    haslowernbr ← false
3:    Set:adjacencies
4:    for Edge e: out_edges(v) do
5:      Vertex u = target(e);
6:      if u < v then
7:        haslowernbr ← true
8:        break
9:      else
10:       adjacencies.insert(u)
11:     end if
12:   end for
13:   if haslowernbr == false then
14:     for Vertex w : adjacencies do
15:       WorkItem generated(w, v);
16:       outset.push(generated);
17:     end for
18:   end if
19: end for
```

15.2. Post-order Connected Components

The post-order processing functions are similar to pre-order processing functions. However, instead of inserting *workitems* to the data structure they are sent over the network to a remote rank.

In CC (Listing 15.1), *vcomponent* state is initialized to *v* for each vertex with the exception of the source vertices. For source vertices, *vcomponent* is initialized to ∞ and initial *workitems* are generated for sources (not to the neighbors of sources). See Algorithm 28 for details.

Algorithm 28 The initial *workitem* generation for post-order CC.

Initialize :

```
1: for Vertex v: V in parallel do
2:   haslowernbr ← false
3:   for Edge e: out_edges(v) do
4:     Vertex u = target(e);
5:     if u < v then
6:       haslowernbr ← true
7:       break
8:     end if
9:   end for
10:  if haslowernbr == false then
11:    vcomponent[v] ← ∞
12:    WorkItem generated(v, v);
13:    outset.push(generated);
14:  end if
15: end for
```

15.3. Split-order Connected Components

The state update and new work generation functions for CC are given in Listing 15.2 and Listing 15.3. The component state update is performed in Listing 15.2 and state change is notified to successors in the Directed Acyclic Graph (DAG) in Listing 15.3.

LISTING 15.2. State update function for CC

```
1 typedef std::tuple<Vertex, Component> WorkItem;
2 struct state_update_cc_pf {
3 void operator()(const WorkItem& wi, int tid, buckets& outset) {
4     Vertex v = std::get<0>(wi);
5     Component component = std::get<1>(wi);
6     if (CAS(&vcomponent[v], component)) {
7       if (component == vcomponent[v])
8         outset.push(wi, tid);
9     }
10  }
11 };
```

LISTING 15.3. New work generation function for CC

```
1 struct new_work_gen_cc_pf {
2 void operator()(const WorkItem& wi, int tid, buckets& outset) {
3     Vertex v = std::get<0>(wi);
4     Component component = std::get<1>(wi);
5     if (component == vcomponent[v]) {
```

```
6       set<Vertex> adjacencies;
7       bool haslowernbr = false;
8       for_each(Edge e : out_edges(v)) {
9         Vertex u = target(e);
10        if (u > component) {
11          adjacencies.insert(u);
12        }else if(u < component) {
13          // v has a lower neighbor
14          haslowernbr = true;
15          break;
16        }
17      }

19      if (!haslowernbr) {
20        for_each(Vertex w : adjacencies) {
21          WorkItem generated(w, component);
22          Send(generated, tid);
23        }
24      }
25    }
26  }
27 };
```

15.4. Orderings

We evaluate the performance of CC for the following orderings:

(1) $<_{level} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{ch}$

(2) $<_{level} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{djcc}$

(3) $<_{ch} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{level}$

The definition of $<_{level}$ is the same as in Listing 13.4. The ordering $<_{djcc}$ orders *workitems* by the component id. Such ordering helps algorithm to converge quickly since the algorithm uses global vertex identifiers as its priorities.

15.5. Experimental Evaluations

15.5.1. Processing Function Execution. We experimentally evaluated the performance of different processing function execution configurations for $<_{level} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{ch}$. These experiments were carried out on a Cray XC system that has 2 Broadwell 22-core Intel Xeon processors. Our experiments only used up to 16 cores to uniformly double the problem

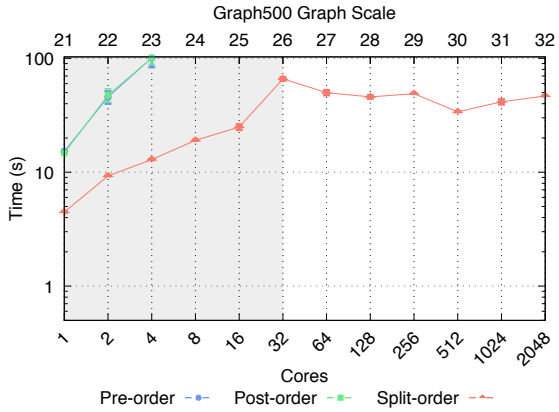


FIGURE 15.1. A comparison of pre-order, post-order and split-order execution configurations for CC.

size and to double the number of processors in weak scaling. Each node consists of 128 GB DDR4-2400 memory. We used an MPI+PThread, distributed shared-memory runtime. The MPI implementation is Cray MPICH (version 7.4.4). The input is Graph500 [**104**] graphs from scale 21–32.

The results are shown in Figure 15.1. As per the results, the pre-order and post-order configurations show poor performance that is not scalable in distributed execution.

15.5.2. Weak Scaling. For weak scaling, we used the same graph input we used for previous graph applications (See Section 13.5.2).

The weak scaling results for algorithms are given in Figure 15.2. In addition to the EAGM orderings discussed above, we also included results from PowerGraph, CC algorithm on Graph500 input (The first plot in Figure 15.2). As explained in Section 14.5.2, PowerGraph uses Gather-Apply-Scatter (GAS) primitives to implement CC algorithm. As can be seen in the plots, the PowerGraph version of the CC algorithm was not scalable in distributed execution and was running out of memory.

Overall, the thread-level ordering ($<_{ch} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{level}$) shows better performance across all graphs. The global level ordering shows competitive performance to thread-level ordering for RMAT-1 and RMAT-2 graphs. However, for ER graphs the performance difference between thread-level ordering and global level ordering is more visible. The ordering $<_{level} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{djcc}$ does not reduce much work over $<_{level} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow$

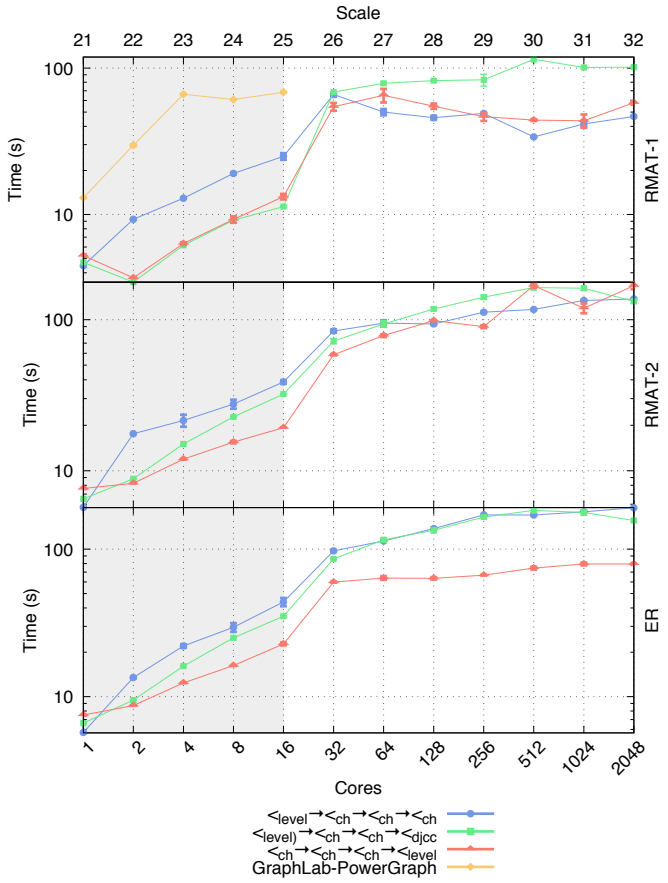


FIGURE 15.2. Weak scaling results for CC orderings.

$<_{ch}$ and also increases the ordering time. Therefore, we do not see much benefit from $<_{level} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{djcc}$.

To further compare the performance of EAGM algorithms, we ran experiments against three other algorithm implementations. The weak scaling results for those experiments are given in Figure 15.3. The *PBGL2-SV* is the active message version of Shiloach-Vishkin [**123**] algorithm and *PBGL2-CC* is the CC implementation of Algorithm 3 on Parallel Boost Graph Library, version 2 (Parallel BGLv2). As per the plot in Figure 15.3, EAGM global level ordering outperformed all other algorithms in distributed execution for both graph inputs.

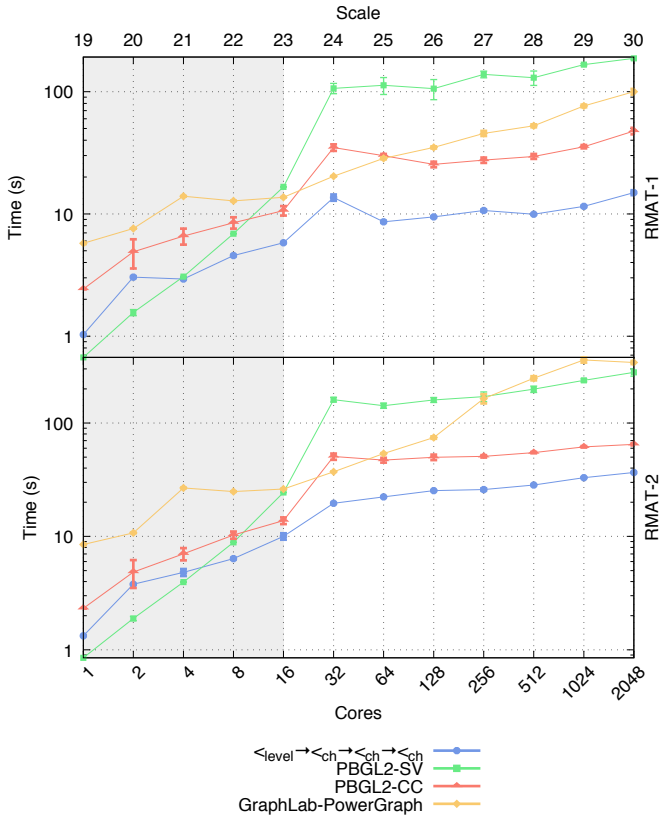


FIGURE 15.3. Weak scaling result comparison for CC.

15.5.3. Strong Scaling. Strong scaling experiments were carried out on a Cray XC30 supercomputer. Each node in the system has two Intel Xeon processors and each processor has 12 CPUs (cores). Each node has 64 GB of DDR3 RAM. The system uses Aries interconnect.

Figure 15.4 shows the strong scaling speed-up results for CC orderings. As can be seen for RMAT1 graphs, the global level ordering shows sound speed-up until about 768 cores, but afterward, the speedup decreases due to synchronization overhead. The globally asynchronous, but thread local, ordering shows more parallelism even at 6144 cores. Both RMAT1 and ER graphs show similar performance trends. Thread local ordering shows better speedup in both cases.

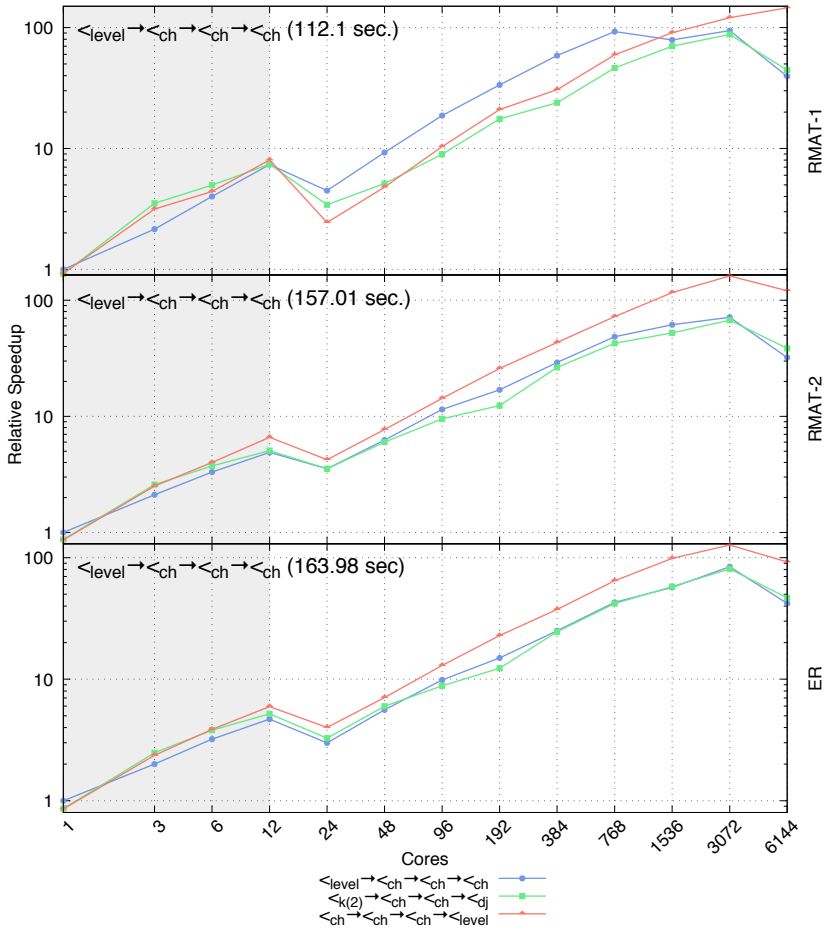


FIGURE 15.4. Strong scaling results for CC orderings. Plots show the relative speed-up. The fastest sequential algorithm is shown on the plot with the timing.

16

Maximal Independent Set

This chapter implements the Extended Abstract Graph Machine (EAGM) version of Maximal Independent Set (MIS) algorithm described in Chapter 8. We evaluate the performance of different processing function execution configurations and their strong and weak scaling performance for different graph inputs.

16.1. Pre-order Maximal Independent Set

The pre-order processing function for MIS is given in Listing 16.1. A *workitem* for MIS is defined as an edge (source vertex and destination vertex) and the source vertex state (Line 1). The *MISState* datatype says whether a vertex is in FIX0, FIX1 or UNFIX state. These states are maintained in *vmis* against every vertex. If the source vertex is in FIX1 state, the destination vertex is moved to FIX0 state and all of its successors are notified (Line 8). In addition to vertex MIS state, every vertex maintains a state to track the number of FIX0 predecessors it processes (`lower_fixed_neighbors`). The `lower_neighbors` is initialized to keep the total number of predecessor for a vertex. When the `lower_fixed_neighbors` count is equal to `lower_neighbors` for a given vertex, its state is moved to FIX1 (Line 20) and its successors are notified of its state change.

LISTING 16.1. State update function for MIS

```
1 typedef std::tuple<Vertex, Vertex, Component> WorkItem;
2 struct mis_pf {
3 void operator()(const WorkItem& wi, int tid, buckets& outset) {
4  Vertex dest_vertex = std::get<0>(wi);
5  Vertex source_vertex = std::get<1>(wi);
6  MISState source_state = std::get<2>(wi);
```

```
 7 if (source_state == MIS_FIX1) {
 8  if (CAS(&vmis[dest_vertex], MIS_UNFIX, MIS_FIX0)) {
 9   for_each(Edge e : out_edges(v)) {
10    Vertex u = target(e);
11    if (u > dest_vertex) {
12     WorkItem wi(u, v, vmis[v]);
13     outset.push(w, tid);
14    }
15   }
16 }else {
17  expected = lower_fixed_neighbors[dest_vertex];
18  newval = oldexpected + 1;
19  if(CAS(&lower_fixed_neighbors[dest_vertex], expected, newval))
       {
20   if (newval == lower_neighbors[dest_vertex]) {
21    vmis[dest_vertex] = MIS_FIX1;
22    for_each(Edge e : out_edges(v)) {
23     Vertex u = target(e);
24     if (u > dest_vertex) {
25      WorkItem wi(u, v, vmis[v]);
26      outset.push(w, tid);
27     }
28    }
29   }
30  }
31 }
32 }
33 }
34 };
```

The *vmis* is initialized to UNFIX for all the vertices, except for source vertices. Source vertex states are initialized to FIX1. The routine to generate initial *workitems* is the same as Algorithm 27, but with added logic to update *vmis* for source vertices (See Algorithm 29, Line 14).

Algorithm 29 The initial *workitem* generation for pre-order MIS.

Initialize *vmis* :

```
1:  for Vertex v: V in parallel do
2:    haslowernbr ← false
3:    Set:adjacencies
4:    for Edge e: out_edges(v) do
5:      Vertex u = target(e);
6:      if u < v then
7:        haslowernbr ← true
8:        break
9:      else
10:       adjacencies.insert(u)
11:     end if
12:   end for
13:   if haslowernbr == false then
14:     vmis[v] ← FIX1
15:     for Vertex w : adjacencies do
16:       WorkItem generated(w, v);
17:       outset.push(generated);
18:     end for
19:   end if
20: end for
```

16.2. Post-order Maximal Independent Set

In post-order execution, every vertex state is set to UNFIX except for the sources. Source vertex states are set to FIX1 and *workitems* are generated for the sources (not to the neighbors of source vertices).

16.3. Split-order Maximal Independent Set

The state update function for MIS (Listing 16.2) updates the vertex state to FIX0 if the source vertex state is FIX1 (Line 7). If the source vertex state is FIX0, the state update function updates `lower_fixed_neighbors` (Line 13). When source vertex is in FIX1, the incoming *workitem* is directly pushed into the data structure. However, the FIX0 *workitem* is only pushed to the data structure if all predecessors are in FIX0 state. The π_{gen} function for MIS (Listing 16.3) generates *workitems* to propagate states.

LISTING 16.2. State update function for MIS

```
1 struct state_update_mis_pf {
```

```
2 void operator()(const WorkItem& wi, int tid, buckets& outset) {
3    Vertex dest_vertex = std::get<0>(wi);
4    Vertex source_vertex = std::get<1>(wi);
5    MISState source_state = std::get<2>(wi);
6    if (source_state == MIS_FIX1) {
7      if (CAS(&vmis[dest_vertex], MIS_UNFIX, MIS_FIX0)) {
8        outset.push(wi, tid);
9      }
10   }else {
11     expected = lower_fixed_neighbors[dest_vertex];
12     newval = oldexpected + 1;
13     if(CAS(&lower_fixed_neighbors[dest_vertex], expected,
          newval)) {
14       if (newval == lower_neighbors[dest_vertex]) {
15         vmis[dest_vertex] = MIS_FIX1;
16         outset.push(wi, tid);
17       }
18     }
19   }
20 }
21 };
```

LISTING 16.3. New work generation function for MIS

```
1 struct new_work_gen_mis_pf {
2 void operator()(const WorkItem& wi, int tid, buckets& outset) {
3    Vertex v = std::get<0>(wi);
4    for_each(Edge e : out_edges(v)) {
5      Vertex u = target(e);
6      if (u > dest_vertex) {
7        WorkItem wi(u, v, vmis[v]);
8        outset.push(w, tid);
9      }
10   }
11 }
12 };
```

16.4. Orderings

We evaluate the performance of MIS for the following orderings:

(1) $<_{ch} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{ch}$

(2) $<_{level} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{ch}$

(3) $<_{stlevel} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{ch}$

(4) $<_{ch} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{level}$

The first ordering represents an asynchronous version of the MIS algorithm. Unlike the other graph applications discussed previously, FIX algorithm is a label setting algorithm even without any ordering (e.g., Single-Source Shortest Paths (SSSP) is label correcting). Therefore, the amount of messages that are generated by the algorithm are constant with any ordering. However, ordering can help to reduce the amount computations (See Section 8.3 for details).

The second ordering globally synchronizes execution after processing a level. The level is calculated as the topological distance from a source vertex in the Directed Acyclic Graph (DAG). The definition of $<_{level}$ is previously discussed (See Listing 13.4). Third ordering is similar to second ordering, but, $<_{stlevel}$ first orders work based on the state of the *workitem* and then on the level.

The definition of $<_{stlevel}$ is given in Listing 16.4. Note that FIX1 has a smaller state value than FIX0. The objective of this ordering is to propagate FIX1 *workitems* faster than FIX0 *workitems*. Then, we can eliminate some of the computations we have to do for FIX0 *workitems*.

LISTING 16.4. The definition of $<_{stlevel}$

```
1 template<int stindex, int levelindex>
2 struct stlevel {
3 public:
4   stlevel(){}

6   template <typename T>
7   bool operator()(T i, T j) {
8     if (std::get<stindex>(i) , std::get<stindex>(j))
9      return true;
10    else
11     return (std::get<levelindex>(i) , std::get<levelindex>(j));
12  }
13 };
```

The last ordering creates equivalence classes based on the level. However, these are created at the thread level. Its definition was discussed in Listing 13.4.

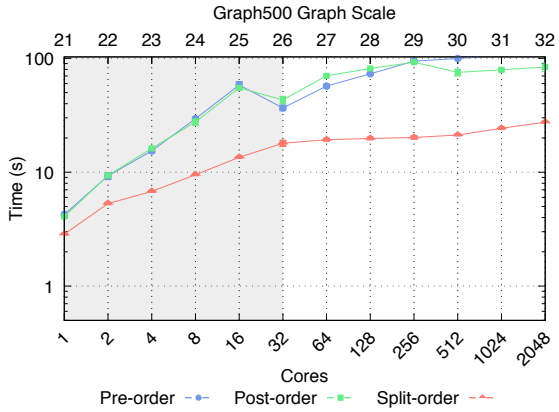


FIGURE 16.1. A comparison of pre-order, post-order and split-order execution configurations for MIS.

16.5. Experimental Evaluations

16.5.1. Processing Function Execution. We experimentally evaluated the performance of different processing function execution configurations for $<_{level} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{ch}$. These experiments were carried out on a Cray XC system that has 2 Broadwell 22-core Intel Xeon processors. Our experiments only used up to 16 cores to uniformly double the problem size and to double the number of processors in weak scaling. Each node consisted of 128 GB DDR4-2400 memory. We used an MPI+PThread, distributed shared-memory runtime. The MPI implementation was Cray MPICH (version 7.4.4). The input was Graph500 [**104**] graphs from scale 21–32.

The results are shown in Figure 16.1. As per the results, the pre-order and post-order configurations showed poor performance that is not scalable in distributed execution. Both pre-order and post-order executions generated new work in a single function. Therefore, the contention on the data structure is quite significant compared to split-order.

16.5.2. Weak Scaling. For weak scaling, we used the same graph input that we used for previous graph applications (i.e., RMAT-1, RMAT-2, and ER graphs). The RMAT-1 graphs also included a comparison between Parallel Boost Graph Library, version 2 (Parallel BGLv2) FIX algorithm and Parallel BGLv2 Luby algorithms.

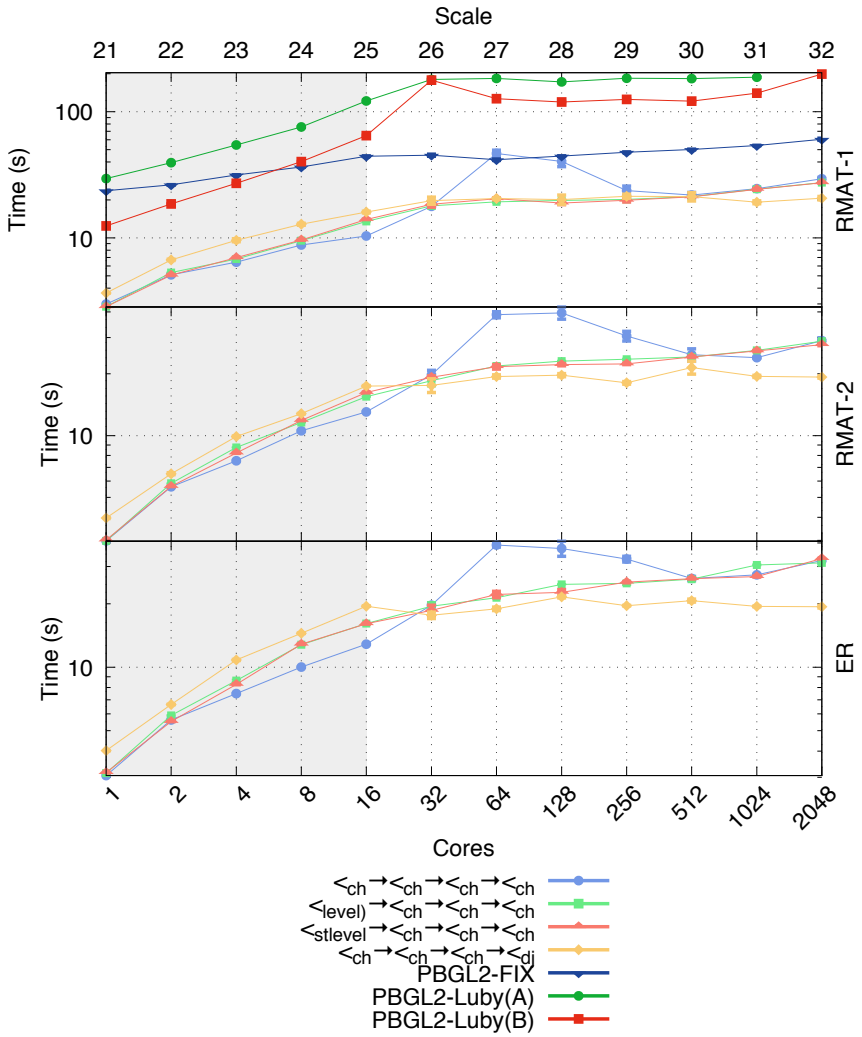


FIGURE 16.2. Weak scaling results for MIS orderings.

Weak scaling results for MIS experiments are given in Figure 16.2. For RMAT-1 graphs, EAGM algorithms outperformed Parallel BGLv2 algorithms. For 64 and 128, the asynchronous ordering performed poorly compared to other orderings for all graph types. At scales 27 and 28, the chaotic ordering performs more atomic operations in updating `lower_fixed_neighbors`, than other orderings. For other EAGM algorithms, ordering helped to keep the number of atomic operations minimum. Hence, we do not see fluctuations as in the asynchronous algorithm at scale 27 and 28. We see that $<_{ch} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{level}$ scales well at high scales because this ordering does not incorporate synchronization overhead.

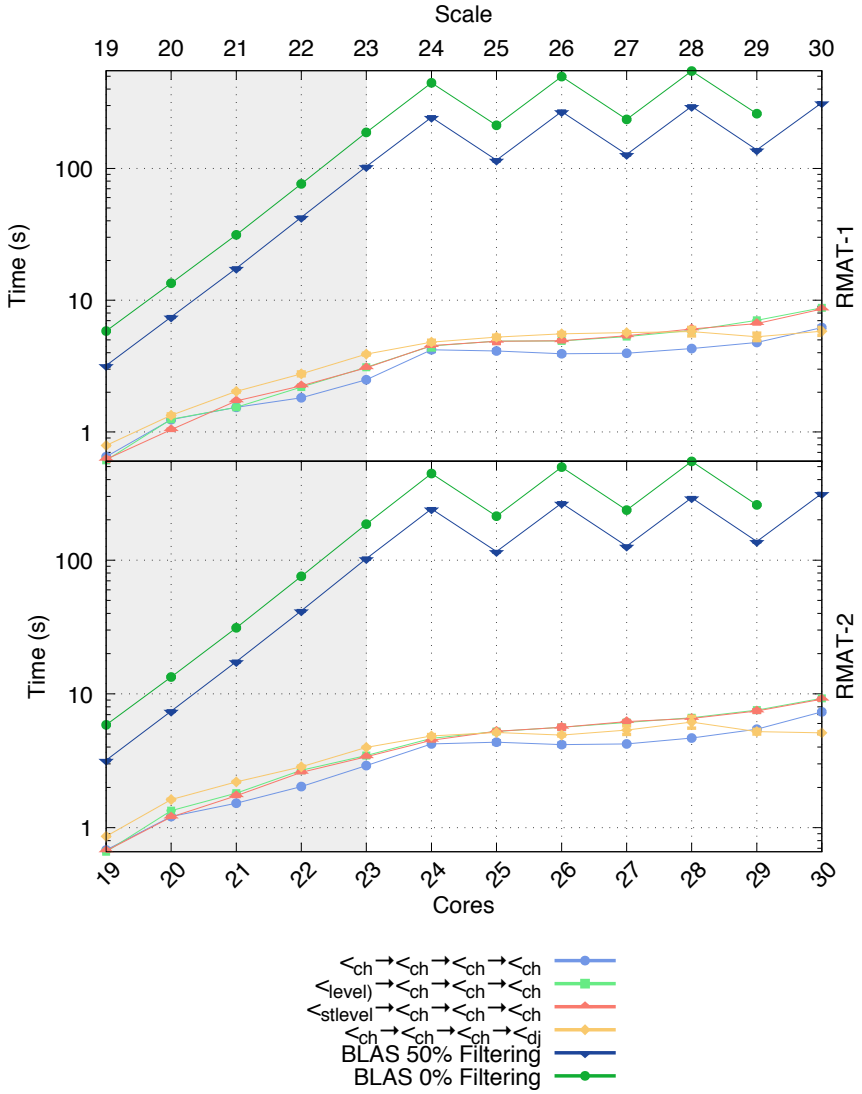


FIGURE 16.3. Weak scaling result comparison for MIS orderings.

We also compared the performance of the above-mentioned orderings with CombBLAS **[20]** FilteredMIS implementation. As mentioned earlier, we were unable to run CombBLAS in the scale for previous weak scaling experiments (i.e., Figure 16.2). Figure 16.3 shows the comparison between CombBLAS and EAGM orderings. The EAGM algorithms are several times faster than CombBLAS 50% edge filtered execution and also 0% edge filtered graphs.

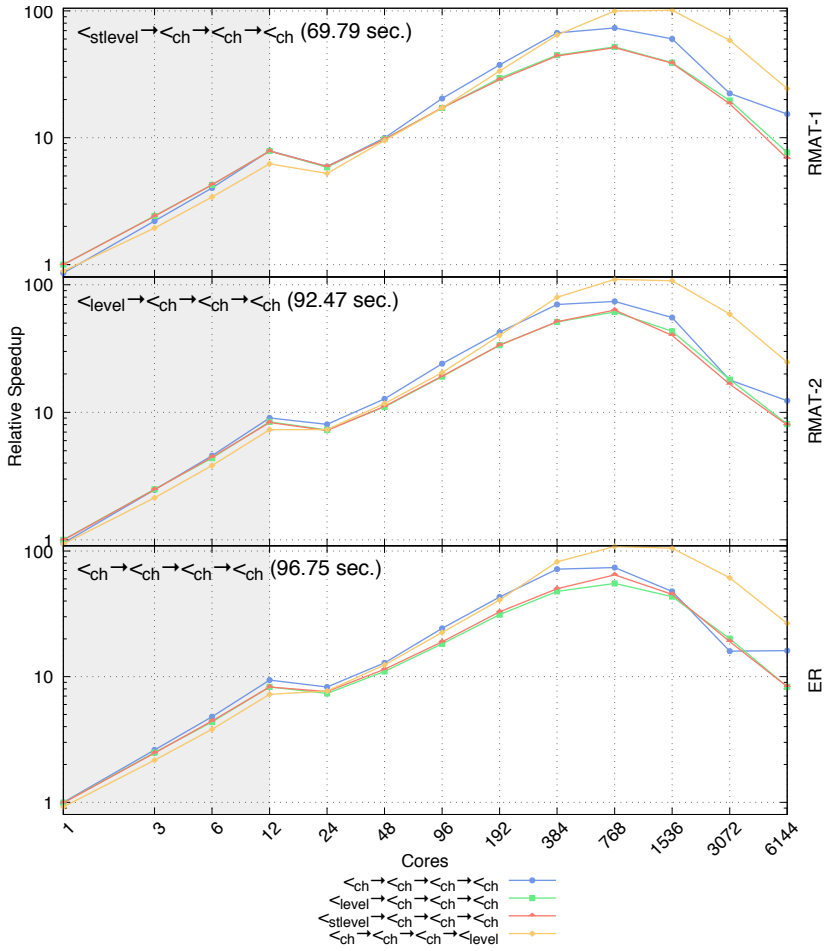


FIGURE 16.4. Strong scaling results for MIS orderings. Plots show the relative speed-up. The fastest sequential algorithm is shown on the plot with the timing.

16.5.3. Strong Scaling. Strong scaling experiments were carried out on a Cray XC30 supercomputer. Each node in the system had two Intel Xeon processors and each processor has 12 CPUs (cores). Each node had 64 GB of DDR3 RAM. The system used Aries interconnect.

Strong scaling speed-up results for MIS variations are shown in Figure 16.4. As we increased the number of parallel threads, the "thread local" ordering speed-up outperformed other orderings. The chaotic ordered algorithm showed almost the same speed-up results as the thread local ordering. However, for several cases, thread local ordering was slightly faster as it was able to reduce the number of computations performed.

17

Conclusion

The Abstract Graph Machine (AGM) model expresses a graph algorithm as a processing function and an ordering. The ordering is specified as a *strict weak ordering* relation. The algorithm starts by executing the processing function with an initial *workitem* set. When executing a processing function more *workitems* can get generated. The work units generated by the processing function are ordered according to the strict weak ordering relation. The strict weak ordering relation creates equivalence classes and work units in an equivalence class can be executed in parallel. However, execution of work units in different equivalence classes must be ordered.

The AGM model generalizes existing parallel graph processing paradigms like Δ-Stepping , K-Level Asynchronous (KLA). These existing parallel graph algorithms (e.g., Single-Source Shortest Paths (SSSP) algorithms), are different only because of the way they order work. We showed that by introducing new orderings we can generate different parallel graph algorithms. New orderings can be introduced either by incorporating new orderings relations on *workitems* or else by introducing new ordering attributes to *workitems*.

The AGM model is further extended to explore spatial orderings of a given architecture. The extended model orders work temporally as well as spatially. The spatial ordering decides how much synchronization cost we need to pay. When an ordering is specified for a global spatial level, after processing every equivalence class we need to execute a global barrier. However, if the ordering is local to a thread or to a process, then we spend less

time on synchronization. We showed that by changing spatial orderings we can generate more efficient parallel graph algorithms.

We showed that new asynchronous algorithms we developed for Connected Components (CC), Maximal Independent Set (MIS) and Triangle Counting (TC) are scalable and execute faster compared to extended shared-memory parallel algorithms. Those new algorithms are modeled using the AGM framework and new orderings were applied. All those algorithm variations were implemented in the same framework and experimented results showed that the ordered algorithms outperform algorithms without ordering. In addition to those AGM algorithms, we also extend two of Luby's seminal algorithms for distributed execution and used that as a baseline to compare the performance of AGM algorithms.

If an AGM is executing an algorithm in a label correcting manner with chaotic ordering, then further orderings will help to reduce the redundant work. Hence, it reduces the number of messages exchanged over the network. Breadth First Search (BFS), SSSP, CC are few examples. However, if an AGM is executing an algorithm in a label setting manner with chaotic ordering, then ordering helps to reduce the number of computations. Correctly defined orderings help label-correcting AGMs to converge faster by eliminating redundant work. Therefore, the performance difference between chaotic ordered execution and ordered execution for label correcting AGMs is higher compared to label setting algorithms. The benefit of ordering is clearly visible for label setting algorithms when they are executed on fewer ranks, but when executing on a larger number of nodes, the benefit of avoiding computation is not significant due to the overhead of distributed synchronization and message communication.

The AGM model is implemented as a graph processing framework on top of a lightweight Message Passing Interface (MPI) wrapper. AGM is an abstract model and mapping AGM model to a parallel hardware can be done in several ways. One of the main choices we had to make was how to place the processing function within the framework. We showed that fundamentally there are two ways to map processing function into an implementation; they are 1. pre-order, 2. post-order. We showed that contention is one

of the main factors that affect the performance of processing function execution configuration. We showed that post-order reduce the contention (especially in distributed execution) since it divides work among multiple ranks.

We came up with a new processing function execution configuration, that gives better performance than pre-order and post-order configurations. The new configuration (split-order) executes state update before ordering and new work generation after ordering. For label correcting algorithms the split-order further prunes work. For label setting algorithms, the split-order configuration reduces the contention on the data structure.

One of the most important parts of an implementation of the AGM model is the data structure that holds *workitems*. We experimented multiple data structures to hold equivalence classes. Two most important aspects we needed to address for a candidate data structure are 1. contention, and, 2. lookup time. Data structures with fast lookups (e.g., Binary Search Tree (BST)) tend to perform unsafe operations (e.g., tree balancing) while doing insertions and deletions (in a multi-threaded environment). Therefore, we need extra concurrency handling to assure those operations leave the data structure in a consistent state. Because of the extra concurrency handling, data structures with fast lookups show poor performance in a multi-threaded environment. Further, concurrent data structures that avoid re-balancing (e.g., SkipList) shows poor performance because of the high contention. Even though linked lists have a linear lookup time, they show better performance with concurrent threads because of the reduced contention. Further, the partitioning scheme we came up, showed the best performance since, it minimizes the contention, insertion time and partitioning time.

The AGM framework is extended to incorporate functionalities of an Extended Abstract Graph Machine (EAGM). The heart of the EAGM implementation is the data structure that holds *workitems*. The EAGM data structure is nested and maintains equivalence classes for different spatial orderings. We statically optimized EAGM orderings by treating `CHAOTIC_ORDER` as a special ordering. Those optimizations help to bypass nested spatial levels with `CHAOTIC_ORDER`.

We evaluated the performance of AGMs with several synthetic graphs and real-word graphs. Results showed that EAGM algorithms performed better than existing parallel algorithms. In most cases, the Erdos-Reyi graphs performed better with globally synchronizing orderings and power-law graphs showed sound performance with EAGM algorithms. For high diameter graphs, global asynchronous algorithms outperformed global synchronous algorithms because of the reduced overhead of synchronization.

Bibliography

[1] Virat Agarwal, Fabrizio Petrini, Davide Pasetto, and David A Bader. Scalable graph exploration on multicore processors. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010.

[2] Ajit Agrawal, Lena Nekludova, and Willie Lim. *A parallel O (log N) algorithm for finding connected components in planar images*. Thinking Machines Corporation, 1987.

[3] Ravindra K Ahuja, Thomas L Magnanti, and James B Orlin. Network flows: theory, algorithms, and applications. 1993.

[4] William Aiello, Fan Chung, and Linyuan Lu. A random graph model for power law graphs. *Experimental Mathematics*, 10(1):53–66, 2001.

[5] Hidayet Aksu, Mustafa Canim, Yuan-Chi Chang, Ibrahim Korpeoglu, and Özgür Ulusoy. Multi-resolution social network community identification and maintenance on big data platform. In *2013 IEEE International Congress on Big Data*, pages 102–109. IEEE, 2013.

[6] Noga Alon, László Babai, and Alon Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *Journal of algorithms*, 7(4):567–583, 1986.

[7] Shaikh Arifuzzaman, Maleq Khan, and Madhav Marathe. Patric: A parallel algorithm for counting triangles in massive networks. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pages 529–538. ACM, 2013.

[8] Baruch Awerbuch and Yossi Shiloach. New connectivity and msf algorithms for shuffle-exchange network and pram. *IEEE Transactions on Computers*, 100(10):1258–1263, 1987.

[9] Ariful Azad, Aydin Buluç, and John Gilbert. Parallel triangle counting and enumeration using matrix algebra. In *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*, pages 804–811. IEEE, 2015.

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

[11] Jiri Barnat, Petr Bauch, Lubos Brim, and Milan Ceška. Computing strongly connected components in parallel on cuda. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 544–555. IEEE, 2011.

[12] Vladimir Batagelj and Matjaz Zaversnik. An o (m) algorithm for cores decomposition of networks. *arXiv preprint cs/0310049*, 2003.

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

[14] Richard Bellman. On a Routing Problem. Technical report, DTIC Document, 1956.

[15] Richard Bellman. On a routing problem. *Quarterly of applied mathematics*, pages 87–90, 1958.

[16] Guy E. Blelloch, Jeremy T. Fineman, and Julian Shun. Greedy Sequential Maximal Independent Set and Matching Are Parallel on Average. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 308–317. ACM, 2012.

[17] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[18] Doruk Bozdağ, Assefaw H Gebremedhin, Fredrik Manne, Erik G Boman, and Umit V Catalyurek. A framework for scalable greedy coloring on distributed-memory parallel computers. *Journal of Parallel and Distributed Computing*, 68(4):515–535, 2008.

[19] Aydin Buluc, Erika Duriakova, Armando Fox, John R Gilbert, Shoaib Kamil, Adam Lugowski, Leonid Oliker, and Samuel Williams. High-productivity and high-performance analysis of filtered semantic graphs. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 237–248. IEEE, 2013.

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

[21] Aydin Buluç and Kamesh Madduri. Parallel breadth-first search on distributed memory systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 65. ACM, 2011.

[22] Ümit V Çatalyürek, John Feo, Assefaw H Gebremedhin, Mahantesh Halappanavar, and Alex Pothen. Graph coloring algorithms for multi-core and massively multithreaded architectures. *Parallel Computing*, 38(10):576–594, 2012.

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

[24] Francis Y Chin, John Lam, and I-Ngo Chen. Efficient parallel algorithms for some graph problems. *Communications of the ACM*, 25(9):659–665, 1982.

[25] Avery Ching. Scaling apache giraph to a trillion edges. *Facebook Engineering blog*, page 25, 2013.

[26] Ka Wong Chong and Tak Wah Lam. Finding connected components in o (log n log log n) time on the erew pram. *Journal of Algorithms*, 18(3):378–402, 1995.

[27] Alok Choudhary and Rajeev Thakur. Evaluation of connected component labeling algorithms on shared and distributed memory multiprocessors. In *Parallel Processing Symposium, 1992. Proceedings., Sixth International*, pages 362–365. IEEE, 1992.

[28] Jonathan Cohen. Graph twiddling in a mapreduce world. *Computing in Science & Engineering*, 11(4):29–41, 2009.

[29] Guojing Cong, George Almasi, and Vijay Saraswat. Fast pgas implementation of distributed graph algorithms. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010.

[30] Stephen A Cook. A taxonomy of problems with fast parallel algorithms. *Information and control*, 64(1):2–22, 1985.

[31] Thomas H Cormen. *Introduction to algorithms*. MIT press, second edition, 2009.

[32] Andreas Crauser, Kurt Mehlhorn, Ulrich Meyer, and Peter Sanders. A Parallelization of Dijkstra's Shortest Path Algorithm. In *Mathematical Foundations of Computer Science 1998*, pages 722–731. Springer, 1998.

[33] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten Von Eicken. Logp: Towards a realistic model of parallel computation. In *ACM Sigplan Notices*, volume 28, pages 1–12. ACM, 1993.

[34] Edsger W Dijkstra. A Note on Two Problems in Connexion With Graphs. *Numerische mathematik*, 1(1):269–271, 1959.

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

[36] Jean-Pierre Eckmann and Elisha Moses. Curvature of co-links uncovers hidden thematic layers in the world wide web. *Proceedings of the national academy of sciences*, 99(9):5825–5829, 2002.

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

[38] Nicholas Edmonds, Jeremiah Willcock, and Andrew Lumsdaine. Expressing graph algorithms using generalized active messages. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 283–292. ACM, 2013.

[39] Nick Edmonds, Alex Breuer, Douglas Gregor, and Andrew Lumsdaine. Single-source shortest paths with the parallel boost graph library. In *The Ninth DIMACS Implementation Challenge: The Shortest Path Problem*, Piscataway, NJ, November 2006.

[40] Nick Edmonds, Alex Breuer, Douglas Gregor, and Andrew Lumsdaine. Single-Source Shortest Paths with The Parallel Boost Graph Library. *The Ninth DIMACS Implementation Challenge: The Shortest Path Problem, Piscataway, NJ*, pages 219–248, 2006.

[41] Benedikt Elser and Alberto Montresor. An evaluation study of bigdata frameworks for graph processing. In *Big Data, 2013 IEEE International Conference on*, pages 60–67. IEEE, 2013.

[42] Paul Erdös and Alfréd Rényi. On random graphs, i. *Publicationes Mathematicae (Debrecen)*, 6:290–297, 1959.

[43] Paul Erdös and Alfréd Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci*, 5(17-61):43, 1960.

[44] Lisa K Fleischer, Bruce Hendrickson, and Ali Pınar. On identifying strongly connected components in parallel. In *Parallel and Distributed Processing*, pages 505–511. Springer, 2000.

[45] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 114–118. ACM, 1978.

[46] Michael L Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

[47] Harold N Gabow. Scaling algorithms for network problems. In *Foundations of Computer Science, 1983., 24th Annual Symposium on*, pages 248–258. IEEE, 1983.

[48] Robert G. Gallager, Pierre A. Humblet, and Philip M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and systems (TOPLAS)*, 5(1):66–77, 1983.

[49] Nishant M Gandhi and Rajiv Misra. Performance comparison of parallel graph coloring algorithms on bsp model using hadoop. In *Computing, Networking and Communications (ICNC), 2015 International Conference on*, pages 110–116. IEEE, 2015.

[50] Assefaw Hadish Gebremedhin and Fredrik Manne. Scalable parallel graph coloring algorithms. *Concurrency - Practice and Experience*, 12(12):1131–1146, 2000.

[51] Edgar N Gilbert. Random graphs. *The Annals of Mathematical Statistics*, 30(4):1141–1144, 1959.

[52] Andrew Goldberg, Serge Plotkin, and Gregory Shannon. Parallel symmetry-breaking in sparse graphs. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 315–324. ACM, 1987.

[53] Mark Goldberg and Thomas Spencer. Constructing a maximal independent set in parallel. *SIAM Journal on Discrete Mathematics*, 2(3):322–328, 1989.

[54] Mark K Goldberg. Parallel algorithms for three graph problems. *Congressus Numerantium*, 54(111-121):4–1, 1986.

[55] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI*, volume 12, page 2, 2012.

[56] Graph500Contributors. Graph 500 benchmark 1 ("search"), 2016.

[57] Oded Green and David A Bader. Faster clustering coefficient using vertex covers. In *Social Computing (SocialCom), 2013 International Conference on*, pages 321–330. IEEE, 2013.

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

[59] Oded Green, Pavan Yalamanchili, and Lluís-Miquel Munguía. Fast triangle counting on the gpu. In *Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms*, pages 1–8. IEEE Press, 2014.

[60] Douglas Gregor and Andrew Lumsdaine. Lifting sequential graph algorithms for distributed-memory parallel computation. In *ACM SIGPLAN Notices*, volume 40, pages 423–437. ACM, 2005.

[61] Yujie Han and Robert A Wagner. An efficient and fast parallel-connected component algorithm. *Journal of the ACM (JACM)*, 37(3):626–642, 1990.

[62] Jennie Hansen, Marek Kubale, Łukasz Kuszner, and Adam Nadolski. Distributed largest-first algorithm for graph coloring. In *European Conference on Parallel Processing*, pages 804–811. Springer, 2004.

[63] Pawan Harish and PJ Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *International Conference on High-Performance Computing*, pages 197–208. Springer, 2007.

[64] Harshvardhan, Adam Fidel, Nancy M. Amato, and Lawrence Rauchwerger. KLA: A New Algorithmic Paradigm for Parallel Graph Computations. In *Proc. 23rd Internat. Conf. on Parallel Architectures and Compilation*, pages 27–38. ACM, 2014.

[65] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2011.

[66] Daniel S. Hirschberg, Ashok K. Chandra, and Dilip V. Sarwate. Computing connected components on parallel computers. *Communications of the ACM*, 22(8):461–464, 1979.

[67] Dylan Hutchison. Distributed triangle counting in the graphulo matrix math library. In *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*, pages 1–7. IEEE, 2017.

[68] Kazuo Iwama and Yahiko Kambayashi. A simpler parallel algorithm for graph connectivity. *Journal of Algorithms*, 16(2):190–217, 1994.

[69] Chirag Jain, Patrick Flick, Tony Pan, Oded Green, and Srinivas Aluru. An adaptive parallel algorithm for computing connectivity. *arXiv preprint arXiv:1607.06156*, 2016.

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

[71] Donald B Johnson and Panagiotis Metaxas. Connected components in o (log3/2— v—) parallel time for the crew pram. In *32nd Annual IEEE Symposium on Foundations of Computer Science*, pages 688–697. Citeseer, 1991.

[72] Mark T Jones and Paul E Plassmann. A parallel graph coloring heuristic. *SIAM Journal on Scientific Computing*, 14(3):654–669, 1993.

[73] Thejaka Kanewala, Marcin Zalewski, and Andrew Lumsdaine. Distributed-memory fast maximal independent set. In *2017 IEEE High Performance Extreme Computing Conference*. IEEE, 2017.

[74] David R Karger, Noam Nisan, and Michal Parnas. Fast connected components algorithms for the erew pram. In *Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*, pages 373–381. ACM, 1992.

[75] Richard M Karp and Avi Wigderson. A fast parallel algorithm for the maximal independent set problem. *Journal of the ACM (JACM)*, 32(4):762–773, 1985.

[76] Mihail N Kolountzakis, Gary L Miller, Richard Peng, and Charalampos E Tsourakakis. Efficient triangle counting in large graphs via degree-based vertex partitioning. *Internet Mathematics*, 8(1-2):161–185, 2012.

[77] Václav Koubek and Jana Kršňáková. Parallel algorithms for connected components in a graph. In *International Conference on Fundamentals of Computation Theory*, pages 208–217. Springer, 1985.

[78] Fabian Kuhn, Thomas Moscibroda, Tim Nieberg, and Roger Wattenhofer. Fast deterministic distributed maximal independent set computation on growth-bounded graphs. In *International Symposium on Distributed Computing*, pages 273–287. Springer, 2005.

[79] Jérôme Kunegis. Konect: the koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*, pages 1343–1350. ACM, 2013.

[80] Matthieu Latapy. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical Computer Science*, 407(1-3):458–473, 2008.

[81] Charles E Leiserson and Tao B Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 303–314. ACM, 2010.

[82] Andrew Lenharth, Donald Nguyen, and Keshav Pingali. Priority Queues Are Not Good Concurrent Priority Schedulers. *The University of Texas at Austin, Department of Computer Sciences, Tech. Rep. TR-11-39*, 2011.

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

[84] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[85] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[86] M Leyzorek, RS Gray, AA Johnson, WC Ladew, SR Meaker Jr, RM Petry, and RN Seitz. Investigation of model techniques–first annual report–6 june 1956–1 july 1957–a study of model techniques for communication systems. *Case Institute of Technology, Cleveland, Ohio*, 1957.

[87] Rong-Hua Li, Jeffrey Xu Yu, and Rui Mao. Efficient core maintenance in large dynamic graphs. *IEEE Transactions on Knowledge and Data Engineering*, 26(10):2453–2465, 2014.

[88] Willie Lim, Ajit Agrawal, and Lena Nekludova. *A fast parallel algorithm for labeling connected components in image arrays*. Thinking Machines Corporation, 1986.

[89] Tze Meng Low, Varun Nagaraj Rao, Matthew Lee, Doru Popovici, Franz Franchetti, and Scott McMillan. First look: Linear algebra-based triangle counting without matrix multiplication. In *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*, pages 1–6. IEEE, 2017.

[90] 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.

[91] Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM journal on computing*, 15(4):1036–1053, 1986.

[92] Lijuan Luo, Martin Wong, and Wen-mei Hwu. An effective gpu implementation of breadth-first search. In *Proceedings of the 47th design automation conference*, pages 52–55. ACM, 2010.

[93] Nancy A Lynch. *Distributed algorithms*. Morgan Kaufmann, 1996.

[94] 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.

[95] Friedemann Mattern. Algorithms for distributed termination detection. *Distributed computing*, 2(3):161–175, 1987.

[96] William Mclendon Iii, Bruce Hendrickson, Steven J Plimpton, and Lawrence Rauchwerger. Finding strongly connected components in distributed graphs. *Journal of Parallel and Distributed Computing*, 65(8):901–910, 2005.

[97] Bruno Menegola. An external memory algorithm for listing triangles. 2010.

[98] Yves Métivier, John Michael Robson, Nasser Saheb-Djahromi, and Akka Zemmari. An optimal bit complexity randomized distributed mis algorithm. *Distributed Computing*, 23(5-6):331–340, 2011.

[99] Ulrich Meyer. Single-source shortest-paths on arbitrary directed graphs in linear average-case time. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 797–806. Society for Industrial and Applied Mathematics, 2001.

[100] Ulrich Meyer and Peter Sanders. Δ-Stepping: A Parallelizable Shortest Path Algorithm. *Journal of Algorithms*, 49(1):114–152, 2003.

[101] Daniele Miorandi and Francesco De Pellegrini. K-shell decomposition for dynamic complex networks. In *Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks (WiOpt), 2010 Proceedings of the 8th International Symposium on*, pages 488–496. IEEE, 2010.

[102] D Mizell and K Maschhoff. Early experiences with large-scale xmt systems. In *Proc. Workshop on Multithreaded Architectures and Applications (MTAAP09)*, 2009.

[103] Alberto Montresor, Francesco De Pellegrini, and Daniele Miorandi. Distributed k-core decomposition. *IEEE Transactions on parallel and distributed systems*, 24(2):288–300, 2013.

[104] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. Introducing the graph 500. *Cray Users Group (CUG)*, 19:45–74, 2010.

[105] Dhruva Nath and SN Maheshwari. Parallel algorithms for the connected components and minimal spanning tree problems. *Information Processing Letters*, 14(1):7–11, 1982.

[106] Jaroslav Nešetřil, Eva Milková, and Helena Nešetřilová. Otakar boruvka on minimum spanning tree problem translation of both the 1926 papers, comments, history. *Discrete mathematics*, 233(1):3–36, 2001.

[107] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A Lightweight Infrastructure for Graph Analytics. In *Proc. 24th ACM Symposium on Operating Systems Principles*, pages 456–471. ACM, 2013.

[108] Noam Nisan, Endre Szemeredi, and Avi Wigderson. Undirected connectivity in o (log n) space. 1997.

[109] Sindhuja Parimalarangan, George M Slota, and Kamesh Madduri. Fast parallel graph triad census and triangle counting on shared-memory platforms. In *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*, pages 1500–1509. IEEE, 2017.

[110] Ha-Myung Park and Chin-Wan Chung. An efficient mapreduce algorithm for counting triangles in a very large graph. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pages 539–548. ACM, 2013.

[111] Roger Pearce. Triangle counting for scale-free graphs at scale in distributed memory. In *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*, pages 1–4. IEEE, 2017.

[112] Roger Pearce, Maya Gokhale, and Nancy M Amato. Scaling techniques for massive scale-free graphs in distributed (external) memory. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 825–836. IEEE, 2013.

[113] Katerina Pechlivanidou, Dimitrios Katsaros, and Leandros Tassiulas. Mapreduce-based distributed k-shell decomposition for online social networks. In *2014 IEEE World Congress on Services*, pages 30–37. IEEE, 2014.

[114] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[115] Vibhor Rastogi, Ashwin Machanavajjhala, Laukik Chitnis, and Anish Das Sarma. Finding connected components in map-reduce in logarithmic rounds. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 50–61. IEEE, 2013.

[116] JH Reif. Optimal parallel algorithms for interger sorting and graph connectivity. technical report. Technical report, Harvard Univ., Cambridge, MA (USA). Aiken Computation Lab., 1985.

[117] John H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229 – 234, 1985.

[118] Georgios Rokos, Gerard Gorman, and Paul HJ Kelly. A fast and scalable graph coloring algorithm for multi-core and many-core architectures. In *European Conference on Parallel Processing*, pages 414–425. Springer, 2015.

[119] Semih Salihoglu and Jennifer Widom. Optimizing graph algorithms on pregel-like systems. *Proceedings of the VLDB Endowment*, 7(7):577–588, 2014.

[120] Carla Savage and Joseph Ja'Ja'. Fast, efficient parallel algorithms for some graph problems. *SIAM Journal on Computing*, 10(4):682–691, 1981.

[121] Thomas Schank. Algorithmic aspects of triangle-based network analysis. pages 26–37, 2007.

[122] Warren Schudy. Finding strongly connected components in parallel using o (log 2 n) reachability queries. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 146–151. ACM, 2008.

[123] Yossi Shiloach and Uzi Vishkin. An o (logn) parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57–67, 1982.

[124] Julian Shun, Laxman Dhulipala, and Guy Blelloch. A simple and practical linear-work parallel algorithm for connectivity. In *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*, pages 143–153. ACM, 2014.

[125] Julian Shun and Kanat Tangwongsan. Multicore triangle computations without tuning. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 149–160. IEEE, 2015.

[126] Natcha Simsiri, Kanat Tangwongsan, Srikanta Tirthapura, and Kun-Lung Wu. Work-efficient parallel union-find with applications to incremental graph connectivity. In *European Conference on Parallel Processing*, pages 561–573. Springer, 2016.

[127] David B Skillicorn, Jonathan Hill, and William F McColl. Questions and answers about bsp. *Scientific Programming*, 6(3):249–274, 1997.

[128] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th international conference on World wide web*, pages 607–614. ACM, 2011.

[129] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[130] Ancy Sarah Tom, Narayanan Sundaram, Nesreen K Ahmed, Shaden Smith, Stijn Eyerman, Midhunchandra Kodiyath, Ibrahim Hur, Fabrizio Petrini, and George Karypis. Exploring optimizations on shared-memory platforms for parallel triangle counting algorithms. In *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*, pages 1–7. IEEE, 2017.

[131] Daniel Tomkins, Timmie Smith, Nancy M Amato, and Lawrence Rauchwerger. Efficient, reachability-based, parallel algorithms for finding strongly connected components. Technical report, Technical report, Texas A&M University, 2015.

[132] Leslie G Valiant. *Bulk-synchronous parallel computers*. Harvard University, Center for Research in Computing Technology, Aiken Computation Laboratory, 1989.

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

[134] Uzi Vishkin. An optimal parallel connectivity algorithm. *Discrete Applied Mathematics*, 9(2):197–207, 1984.

[135] Howard T Welser, Eric Gleave, Danyel Fisher, and Marc Smith. Visualizing the signatures of social roles in online discussion groups. *Journal of social structure*, 8(2):1–32, 2007.

[136] Jeremiah James Willcock, Torsten Hoefler, Nicholas Gerard Edmonds, and Andrew Lumsdaine. AM++: A Generalized Active Message Framework. In *Proc. 19th Internat. Conference on Parallel Architectures and Compilation Techniques*, pages 401–410. ACM, 2010.

[137] Michael M Wolf, Mehmet Deveci, Jonathan W Berry, Simon D Hammond, and Sivasankaran Rajamanickam. Fast linear algebra-based triangle counting with kokkoskernels. In *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*, pages 1–7. IEEE, 2017.

[138] Yinglong Xia and Viktor K Prasanna. Topologically adaptive parallel breadth-first search on multicore processors. In *Proceedings of the 21st IASTED International Conference*, volume 668, page 91, 2009.

[139] Andy Yoo, Edmond Chow, Keith Henderson, William McLendon, Bruce Hendrickson, and Umit Catalyurek. A scalable distributed parallel breadth-first search algorithm on bluegene/l. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pages 25–25. IEEE, 2005.

[140] Kisun You, Jike Chong, Youngmin Yi, Ekaterina Gonina, Christopher J Hughes, Yen-Kuang Chen, Wonyong Sung, and Kurt Keutzer. Parallel scalability in speech recognition. *IEEE Signal Processing Magazine*, 26(6):124–135, 2009.

[141] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E Priebe, and Alexander S Szalay. Flashgraph: Processing billion-node graphs on an array of commodity ssds. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 45–58, 2015.

Thejaka Amila Kanewala

Skills Summary

- **Domains:** Web Services (SOAP & REST), Parallel & Distributed Graph Processing, Distributed in-memory graph storages, Cloud Systems, High performance computing, Service Oriented Architecture, Application Security, Fault Tolerance, Monitoring, Distributed Systems, Messaging (peer-to-peer, publish and subscribe, broadcast).
- **Programming Languages:** C, C++, Java, C#, Python, Javascript, Scheme, Agda, SQL.
- **Paradigms & Frameworks:** C++ Template meta-programming, Object Oriented Design, parallel programming with pthreads & OpenMP, MPI, C++ & GNU atomics, JDBC, JNDI, Servlets, JSP, Javascript, Web Services, SOAP, OSGI.
- **Tools:** Git, Svn, CppUnit, JUnit, Jenkins, Bamboo, Cruise Control, Tau, Perf, CrayPat, JProfiler, RDBMS (Oracle, MySQL), NoSQL, Gdb, Lldb, perf, valgrind, address/thread sanitizers, dtrace, tau, scorep, FindBugs, Jira, Maven, Docker.
- **Software Engineering:** Agile/Scrum, test driven development (TDD), unit testing, integration testing, performance testing, continuous build integration, code reviews, design documentation, issue tracking.
- **Open-Source Contributions:** Apache Axis2 (committer), Apache Rampart (committer), Apache WSS4J, and Apache Airavata (committer & Project Management Committee).

Education

Indiana University — Bloomington, Indiana, USA.
Ph.D. — *Aug '12 – September '18*

- Advisor : Professor Andrew Lumsdaine
- Thesis : Abstract Graph Machine(AGM) : Modeling Orderings in Distributed-Memory Parallel Asynchronous Graph Algorithms – AGM represents a distributed-memory parallel graph algorithm as a processing function and an ordering. The ordering is specified as a strict weak ordering. The model is further extended to explore orderings at spatial memory levels in a given architecture. The model and the extended model is implemented using MPI and available here.
- Major : Computer Science (Systems)
- Minor : Logic

Indiana University — Bloomington, Indiana, USA.
Master in Computer Science (GPA - 3.932/4.000) — *Aug '15*

University of Moratuwa — Moratuwa, Sri Lanka.
Master of Science in Computer Science — *Jan '06 – Sept '08*

University of Moratuwa — Moratuwa, Sri Lanka.
Honours Degree of Bachelor of Science and Engineering — *Jan '00 – Oct '04*

Awards

- Best student candidate paper at IEEE HPEC 2017.
- The most outstanding contributor (WSO2, 2012).
- The most outstanding onsite consultant (WSO2, 2011).
- International Mathematical Olympiad (Taejon, Korea, 2000).

Professional Experience

Advanced Computing, Mathematics and Data Division — Pacific Northwest National Laboratory, USA

Research Associate *March '17 – present*

Actively developing a high performance graph processing framework for the new processor in DARPA HIVE project.

- Developing framework features for distributed graph processing (multi-threading, network communication, termination detection, message aggregation etc.).
- Developing synchronous and asynchronous parallel and distributed graph algorithms.
- Designed and developed distributed data structures for graphs.

Center for Research in Extreme Scale Technologies — Indiana University, USA

Research Assistant *Jan '14 – Feb '17*

An active developer for NSFGraph project funded by NSF.

- Developed high performance distributed systems for processing large scale graphs.
- Designed and implemented scalable graph algorithms that can process large data sets.
- Incorporated features into HPX-5, HPX, and AM++ to process large graphs.

Science Gateways Research Center, Pervasive Technology Institute — Indiana University, USA

Research Assistant *Aug '12 – Dec '13*

An active developer for Apache Airavata and SciGap projects.

- Designed and implemented application security features (e.g., authentication, authorization, confidentiality etc.) necessary to secure the interaction between client programs and cloud based science gateways.
- Proposed, designed and implemented a novel method to manage computing resource credentials at the science gateway servers came up with an efficient, secure implementation.
- Proposed a fault tolerant design for the science gateway and implemented necessary fault tolerance features.
- For consecutive 3 years mentored GSoC projects of Apache Airavata project.

WSO2 — USA

Tech Lead *April '10 – July '12*

An active developer for WSO2 identity server.

- Integrated LDAP as a user store for the identity server.
- Designed and developed Web Services security features as part of Apache Rampart.
- Incorporated Kerberos authentication to Identity Server.
- Gathered requirements, designed and implemented security features for WSO2 products, with the required security aspects pertaining to authentication, authorization, non-repudiation, confidentiality, and auditing.
- Implemented role-based access control mechanisms and fine-grained authorization mechanisms using XACML.
- Member of the recruiting squad, managed a team of 2-4.
- Mentored and assisted team members when they needed, carried out necessary training/knowledge transfers for team members, guided them on bringing their novel ideas into the design discussion table and motivated them to improve those ideas and materialize them into final ontime deliverables.

FAST Search & Transfer (Microsoft Subsidiary) — Norway

Senior Software Engineer / Tech Lead *Aug '07 – Apr '10*

An active developer for FAST enterprise search engine.

- Researched and developed a SQL interface to query unstructured data from index storage.
- Involved in parsing the Abstract Syntax Tree (AST) of a SQL query and implementing necessary operations to retrieve indexed data.

- Involved in porting Java search engine implementation to C#.
- Played a main role in recruiting, mentioning and managing a team of 3-5, played the Scrum Master role.

Millennium Information Technologies R & D — Sri Lanka

Software Engineer / Senior Software Engineer — *Sep '04 – Jul '07*

An active back-end developer for a telecom billing and rating engine.

- Designed, prototyped and implemented a distributed Fault tolerance system for telecom billing engine.
- Involved in designing and developing highly configurable framework for telecom billing and rating.

Publications (selected)

1. **Kanewala, Thejaka Amila**, Zalewski, M., & Lumsdaine, A. (2018, June). Distributed, Shared-Memory Parallel Triangle Counting. In Proceedings of the Platform for Advanced Scientific Computing Conference. ACM.
2. **Kanewala, Thejaka**, Marcin Zalewski, and Andrew Lumsdaine. "Parallel Asynchronous Distributed-Memory Maximal Independent Set Algorithm with Work Ordering." 2017 IEEE 24th International Conference on High Performance Computing (HiPC). IEEE, 2017.
3. *Best Student Candidate Paper:* **Kanewala, Thejaka**, Marcin Zalewski, and Andrew Lumsdaine. "Distributed-memory fast maximal independent set." High Performance Extreme Computing Conference (HPEC), 2017 IEEE. IEEE, 2017.
4. **Kanewala, Thejaka Amila**, Marcin Zalewski, and Andrew Lumsdaine. "Families of Graph Algorithms: SSSP Case Study." European Conference on Parallel Processing. Springer, Cham, 2017.
5. Firoz, J. S., **Kanewala, Thejaka Amila**, Zalewski, M., Barnas, M., & Lumsdaine, A. (2018, September). "Synchronization-Avoiding Graph Algorithms." 2018 IEEE 25th International Conference on High Performance Computing (HiPC). IEEE, 2018 (*Accepted for publication*).
6. Firoz, J. S., **Kanewala, Thejaka Amila**, Zalewski, M., Barnas, M., & Lumsdaine, A. (2016, June). Context Matters: Distributed Graph Algorithms and Runtime Systems: A Case Study of Distributed Graph Traversals. In Proceedings of the Platform for Advanced Scientific Computing Conference (p. 12). ACM.
7. Firoz, J. S., **Thejaka Amila Kanewala**, Marcin Zalewski, Martina Barnas, and Andrew Lumsdaine. "Importance of Runtime Considerations in Performance Engineering of Large-Scale Distributed Graph Algorithms." In 1st Workshop on Performance Engineering for Large Scale Graph Analytics at EuroPar, August 2015. Springer.
8. Zalewski, Marcin, **Thejaka Amila Kanewala**, Jesun Sahariar Firoz, and Andrew Lumsdaine. "Distributed control: priority scheduling for single source shortest paths without synchronization." In Proceedings of the Fourth Workshop on Irregular Applications: Architectures and Algorithms, pp. 17-24. IEEE Press, 2014.
9. **Kanewala, Thejaka Amila**, Suresh Marru, Jim Basney, and Marlon Pierce. "A Credential Store for Multi-Tenant Science Gateways." In Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on, pp. 445-454. IEEE, 2014.
10. Basney, Jim, **Thejaka Amila Kanewala**, Jeff Gaynor, Suresh Marru, Rion Dooley, and Joe Stubbs. "Integrating Science Gateways with XSEDE Security: A Survey of Credential Management Approaches." (2014).
11. Pierce, Marlon, Suresh Marru, **Thejaka Amila Kanewala**, Lahiru Gunathilake, Raminder Singh, Saminda Wijeratne, Chathuri Wimalasena et al. "Apache Airavata: Design and Directions of a Science Gateway Framework." In Science Gateways (IWSG), 2014 6th International Workshop on, pp. 48-54. IEEE, 2014.

12. **Thejaka Amila Kanewala**, and S. Jayasena. "Persistent data structure library for C++ applications." In Parallel Distributed and Grid Computing (PDGC), 2010 1st International Conference on, pp. 356-361. IEEE, 2010.

13. **[Poster]** Firoz, J. S., **Thejaka Amila Kanewala**, Zalewski M., Barnas M., and Lumsdaine A. "POSTER: Distributed Control: The Benefits of Eliminating Global Synchronization via Effective Scheduling." In Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 441-442. ACM, 2017.

14. **Kanewala, Thejaka Amila**, Marcin Zalewski, and Andrew Lumsdaine. "Abstract Graph Machine." arXiv preprint arXiv:1604.04772 (2016).