

© Copyright 2020

Saranya Duraisamy

AGENT-BASED PARALLELIZATION OF BIOLOGICAL NETWORK MOTIF DETECTION

Saranya Duraisamy

A report

submitted in partial fulfillment of the
requirements for the degree of

Master of Science in Computer Science & Software Engineering

University of Washington, Bothell

2020

Project Committee:

Munehiro Fukuda, Chair

Wooyoung Kim, Member

Clark Olson, Member

Program Authorized to Offer Degree:

Computing and Software Systems

University of Washington, Bothell

Abstract

AGENT-BASED PARALLELIZATION OF BIOLOGICAL NETWORK MOTIF DETECTION

Saranya Duraisamy

Chair of the Supervisory Committee:
Dr. Munehiro Fukuda
Computing and Software Systems

Network motifs are subgraph patterns that occur frequently in biological networks and represent significant interaction between molecules. Discovering motifs reveal unidentified interactions that are of great importance to biological applications. However, motif detection is a computationally intense process due to the exponential growth of motif patterns with an increase in network or motif size. Due to the computational complexity, existing sequential tools impose a limitation on motif sizes, and larger network analysis takes unreasonable time. The performance issue of these tools resulted in a constant drive to improve the speed with parallel approaches. However, most approaches using MapReduce, OpenMPI, and previously implemented agent-based parallelization are limited to compute the frequency of candidate motifs and don't offer tools to detect significant motifs. Hence, this project implements parallel agent-based significant motif discovery using the MASS (Multi-Agent Spatial Simulation) library by crawling the reactive agents over the network distributed across multiple computing nodes. Additional Spark implementation helped in identifying strengths and enhancements to MASS to handle large-scale data. Compared to previous MASS agent-based implementation, the latest implementation gained at most 2x speedup and reduced memory usage by a factor of 2. Spark implementation attained almost 2x speedup compared to the sequential NemoLib tool. Although MASS implementation encountered memory limitation, both MASS and Spark implementations exhibited a higher level of parallelism with increased computing power and memory resources. Additionally, this work discusses the opportunity to parallelize graph algorithms with MASS in terms of development efforts and data reuse benefits.

TABLE OF CONTENTS

List of Figures	iii
List of Tables	iv
Chapter 1. Introduction	5
1.1 Problem Description	5
1.2 Parallel Frameworks	6
1.3 Research Goals.....	9
Chapter 2. Background	9
2.1 Network Motif Detection Process.....	10
2.2 Subgraph Enumeration Algorithm.....	11
Chapter 3. Related Works	12
3.1 Sequential Network Motif Detection Tools.....	12
3.2 Parallel Network Motif Detection Approaches.....	13
3.3 MASS-based Parallel Network Motif Analysis	14
Chapter 4. Parallelization of Biological Network Motif Detection	15
4.1 System Flow.....	15
4.2 MASS Implementation	18
4.3 Spark Implementation.....	21
Chapter 5. Results	25
5.1 Execution Environment	25

5.2	Input Datasets.....	25
5.3	Execution Performance.....	26
5.4	MASS Speedup and Memory Reduction.....	29
5.5	MASS Feature Evaluation	31
5.6	MASS versus Spark Analysis.....	33
Chapter 6. Conclusion & Future Work.....		37
Bibliography		38
Appendix A.....		42
Appendix B.....		43
Appendix C.....		46

LIST OF FIGURES

Figure 1.1. Non-Isomorphic Directed Motif Structures for Motif Size 3 ^[2]	5
Figure 1.2. MASS Architecture ^[3]	7
Figure 1.3. Spark Architecture ^[10]	8
Figure 2.1. Network Motif Detection Process.....	10
Figure 2.2. Subgraph Enumeration Algorithm ^[12]	11
Figure 2.3. Subgraph Enumeration Tree ^[12]	11
Figure 4.1. System Flow for Network Motif Detection.....	15
Figure 4.2. Graph6 and Canonical Labels for Motif Size 3.....	16
Figure 4.3. MASS Execution Result for Motif Size 5 in 1000 Random Graphs.....	17
Figure 4.4. MASS Agents Execution for Motif Size 3.....	18
Figure 4.5. MASS Place and Agent Data Structure.....	19
Figure 4.6. Spark Implementation of Subgraph Enumeration.....	22
Figure 4.7. Spark's RDD Lineage Re-evaluation for Spark Action.....	24
Figure 5.1. Dolphin Network.....	26
Figure 5.2. Undirected Real Networks Performance.....	27
Figure 5.3. Directed Real Networks Performance.....	28
Figure 5.4. Parallel I/O Graph.....	31
Figure 5.5. MASS Parallel Performance.....	33
Figure 5.6. Spark Parallel Performance.....	33

LIST OF TABLES

Table 1.1. Non-Isomorphic Subgraph Patterns in Graphs up to size 10^{21}	6
Table 5.2. Real Network Graph Properties.....	26
Table 5.3. Undirected Synthetic Graph Properties.....	26
Table 5.4. Enumerated Subgraphs Count in Undirected Real Graphs.....	28
Table 5.5. Enumerated Subgraphs Count in Directed Real Graphs.....	29
Table 5.6. New MASS Speedup in Undirected Synthetic Graphs.....	29
Table 5.7. Memory Reduction in MASS Implementation.....	30
Table 5.8. Input graph size for different formats.....	31
Table 5.9. MASS callAll vs doAll Performance.....	33
Table 5.10. Lines of Code (LoC) and Boilerplate Code Ratio.....	34
Table 5.11. Parallel Methods or Operations Count.....	35
Table 5.12. MASS Speedup for In-Memory Data Reuse.....	36

Chapter 1. INTRODUCTION

‘Network Motifs’ are defined as the recurrent and statistically significant patterns that are found more often in the biological networks than in randomized networks [1]. Network motif detection and analysis led to the discovery of unidentified biological interactions, detection of essential proteins, drug discovery, and disease diagnosis.

Biological data can be analyzed by modeling biological data as a graph with vertices representing molecules, and edges symbolizing molecular interaction. Network motif detection involves computationally intense subgraph enumeration, random graph generation, NP-complete subgraph isomorphic testing, and statistical testing. Existing sequential tools [5], [6] employ sampling methods (to find approximate results) to reduce computational complexity at the expense of detection accuracy.

Vertex-to-vertex communication and in-memory data analysis play a vital role to parallelize the network motif detection. Unlike other parallel frameworks, MASS offers direct communication between vertices by migrating agents, and Spark offers in-memory computation by reducing data movement. These features motivate us to use the MASS and Spark frameworks in this work.

This research exploits MASS [3] agent-based parallelization and Spark [4] parallelization to reduce computational complexity without compromising motif detection accuracy and enhance the scalability of motif detection. The main bottleneck in motif detection is subgraph enumeration, taking on average more than 95% of the whole execution time, as noted in [18]. In this work, computationally expensive subgraph enumeration step has been parallelized in both MASS and Spark implementations.

1.1 PROBLEM DESCRIPTION

The k -sized network motifs are k -vertices induced subgraphs that occur more frequently than any other k -vertices subgraphs in the target network.

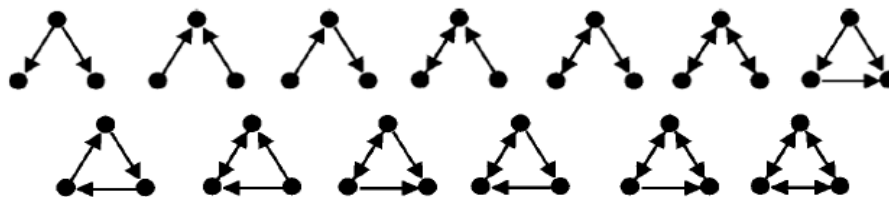


Figure 1.1. Non-Isomorphic Directed Motif Structures for Motif Size $3^{[2]}$.

Figure 1.1 depicts 13 non-isomorphic directed motif structures for motif size 3. As seen in Table 1.1, the number of non-isomorphic subgraph patterns increases exponentially with an increase in motif size ('k' vertices motif). For directed graphs, subgraph patterns increase enormously. Enumerating all k-sized subgraphs in a large graph is computationally intensive. Consequently, existing sequential tools [5], [6], and parallel works [14], [16] impose the maximum motif size limitation that can be detected using these tools.

Table 1.1. Non-Isomorphic Subgraph Patterns in Graphs up to size 10^{12} .

Vertices	Undirected	Directed
1	1	1
2	1	2
3	2	13
4	6	199
5	21	9364
6	112	1530843
7	853	880471142
8	11117	1792473955306
9	261080	13026161682466252
10	11716571	341247400399400765678

1.2 PARALLEL FRAMEWORKS

Due to the computational complexity described in section 1.1, sequential tools [5], [6], and [7] take a long time to detect large motif sizes or analyze large graph sizes, as these tools are restricted to use single machine compute and memory resources. This work attempts to speedup the motif detection by utilizing collective memory and compute power offered by multiple systems in the cluster environment. To realize the goal of improving motif detection speed, this work parallelized network motif detection process using the MASS and Spark framework described in this section.

1.2.1 *Multi-Agent Spatial Simulation (MASS) Library*

Multi-Agent Spatial Simulation (MASS) [3] is a parallel computing library built as an agent-based model intended to parallelize applications from physical, biological, social, and behavioral domains. Figure 1.2 portrays the high-level architecture of the MASS library. MASS comprises two major components, Places and Agents. Places represent simulation space, a multi-dimensional matrix dynamically allocated over the computing nodes in the cluster. Agents represent execution instances that can reside at a place, or migrate to another place in a local or remote computing node. Agents can access data at its residing place, communicate with other agents via inter-agent

broadcast [8], and are also capable of duplicating itself. Applications can be parallelized in the MASS library using the notion of places and agents.

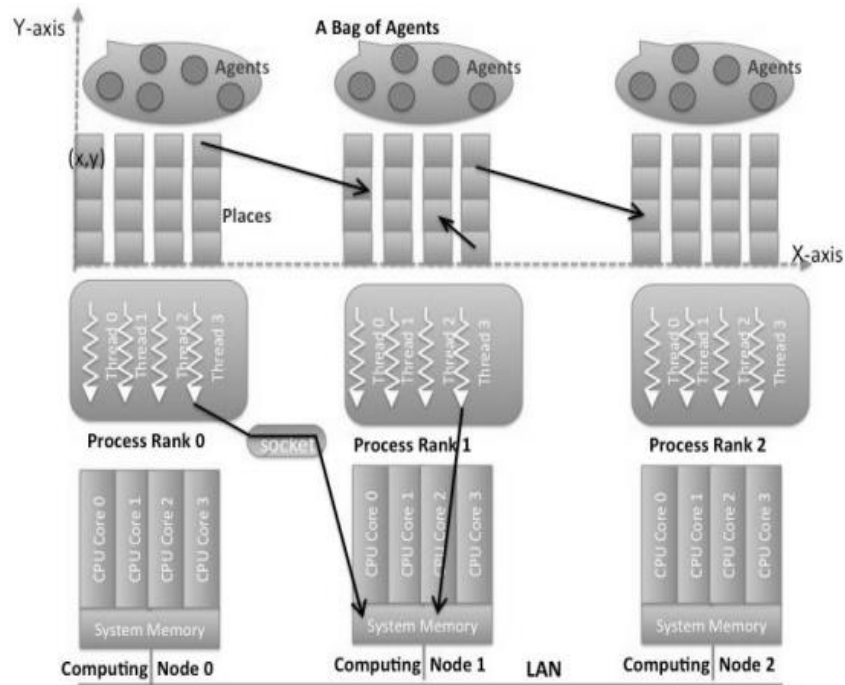


Figure 1.2. MASS Architecture^[3].

MASS spawns as many threads as the number of CPU cores and executes parallel operations on all places and agents using multiple threads. MASS uses sockets to communicate across the computing nodes utilized for the execution. MASS library completely hides the underlying parallel framework so that developers can focus only on their application.

1.2.2 Apache Spark Framework

Apache Spark is an open-source parallel framework that provides a unified computing engine for different data analytic tasks such as SQL querying, real-time streaming, machine learning, and big data applications. As seen from Figure 1.3, Spark [4] follows Master-Worker architecture with two main processes (driver and executor) and a cluster manager. Its driver's responsibility to translate spark application into actual spark jobs that run on the worker nodes. The cluster manager is responsible for allocating and deallocating resources to spark jobs.

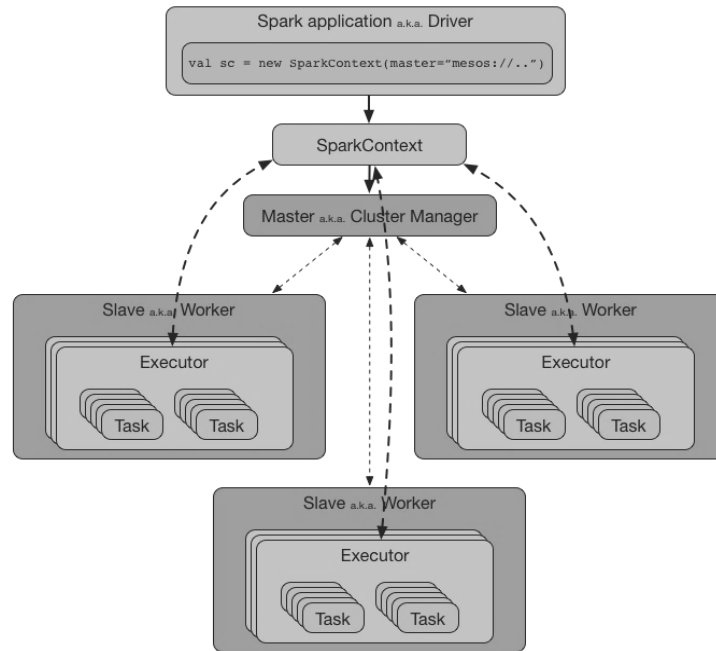


Figure 1.3. Spark Architecture^[10].

Spark provides a fault-tolerant abstraction termed as Resilient Distributed Datasets (RDD) [9], which are the immutable collection of elements partitioned over the distributed computing nodes. Spark provides two operations - transformations and actions, that can be performed on RDD partitions in parallel. Transformation operations such as map, filter, join build new RDDs by transforming parent RDDs. Action operations such as collect, reduce, count performs computation on RDDs, and return results to the driver. Spark lazily evaluates transformations by delaying execution until action is requested.

Transformations are further classified into narrow and wide based on the dependency involved in RDD creation. Objects residing in a single partition are dependent only on objects residing in a single partition of the parent RDD in narrow dependencies (map, filter). In contrast, objects are dependent upon objects residing in multiple partitions of the parent RDD in wide dependencies (join, reduceByKey). Thus, wide dependencies require costly shuffle operation that involves redistributing data across the worker machines.

Spark runtime splits the spark application into multiple jobs when it encounters spark action. Each job is divided into multiple stages when runtime encounters a wide-dependency transformation. Each stage, in turn, includes multiple tasks (pipelined narrow-dependencies) that are executed in parallel on RDD partitions. Thus, Spark builds an optimized execution plan to run parallel tasks.

1.3 RESEARCH GOALS

The research goals of this capstone work are as follows:

- *Improve Execution Speed* of the network motif detection process by parallelizing computation-intensive k-sized subgraph enumeration in the cluster environment.
- *Enhance the Scalability* of the network motif detection by detecting large size motifs and analyzing large graph sizes. Scalability can be enhanced by utilizing memory and compute power offered by all computing machines in the cluster environment.
- *Evaluate MASS Features*. MASS Parallel File I/O, Asynchronous Automatic Agent Migration features were developed to improve the parallel performance, and Agent Control Population feature was implemented to overcome the computation and memory overhead incurred by agent expansion for big data applications. This research evaluates these features for the network motif detection problem and identifies potential enhancements.
- *Identify Enhancements to MASS for Big Data Applications*. Comparing MASS and Spark implementations reveal possible enhancements to MASS to handle big data applications. This comparative analysis exposes MASS strengths to intuitively parallelize similar graph problems with lesser development efforts for biologists from the non-parallel computing background.

The rest of this document is organized as follows: Chapter 2 introduces the network motif detection process and algorithm employed to enumerate subgraphs. Chapter 3 reviews related works of sequential and parallel network motif detection tools. Chapter 4 describes in detail the system flow of the parallel motif detection, presents details of MASS and Spark implementations along with the performance enhancements. Chapter 5 provides a detailed experimental evaluation of MASS and Spark parallelization on different real and synthetic data sets. Finally, Chapter 6 concludes the paper with limitations and future enhancements.

Chapter 2. BACKGROUND

This chapter provides background on various steps involved in the network motif detection process and explains the algorithm used to enumerate subgraphs in the input and random networks.

2.1 NETWORK MOTIF DETECTION PROCESS

Network motif detection process involves the below three major steps, as depicted in Figure 2.1.

1. *Find Candidate Motif Frequencies in Input Graph.* Network-centric motif search approach finds candidate motifs in input graph 'G' by enumerating all k-sized motifs, groups isomorphic motifs, and then computes the frequency of the non-isomorphic motifs $F_G(m)$.
2. *Find Candidate Motif Frequencies in Random Graphs.* Motif search process generates hundreds or thousands of random graphs 'R' by preserving the topological properties of the input network such as the number of vertices and degree of each vertex. Then, it repeats enumeration and frequency computation of all non-isomorphic k-sized candidate motifs $F_R(m)$ in each of the random graphs.
3. *Statistical Significance Test.* Finally, the statistical significance test using Z-score and p-value is performed to discover candidate motifs that are significant network motifs.

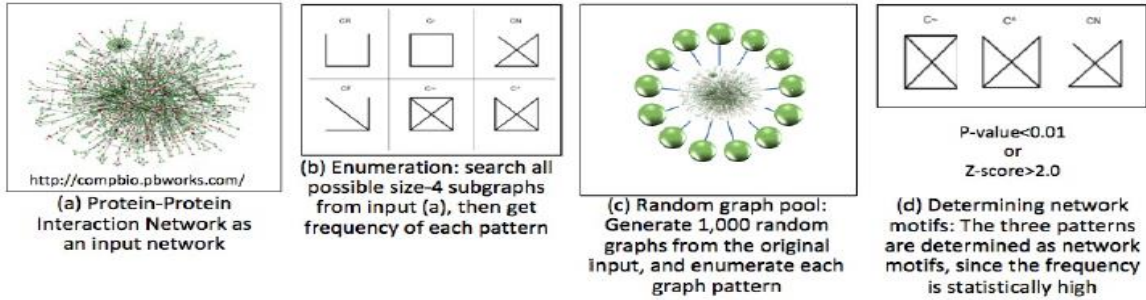


Figure 2.1. Network Motif Detection Process.

Z-score is the ratio of the difference between the original frequency and the mean random frequency to the standard deviation. Z-score may be undefined when the standard deviation is zero. The higher the Z-score, the more significant is the network motif.

$$Z(m) = \frac{F_G(m) - \text{Mean}(F_R(m))}{\text{StandardDeviation}(F_R(m))}$$

The p-value represents the number of random networks in which network motif occurred more often than in the original network, divided by the number of random networks 'N'. This value lies in the range from 0 to 1 inclusive. The smaller the p-value, the more significant is the network motif.

$$p(m) = \frac{1}{N} \sum_{n=1}^N c(n) \quad \text{where } c(n) = 1, \text{ if } F_R(m) \geq F_G(m)$$

Generally, candidates with $Z(m) > 2$ and $p(m) < 0.01$ are recognized as significant motifs [11].

2.2 SUBGRAPH ENUMERATION ALGORITHM

The Enumerate Subgraph (ESU) algorithm [12] is the fastest and efficient algorithm to enumerate all k -sized subgraphs in the input graph. Figure 2.2 shows the ESU algorithm that recursively enumerates subgraphs from each vertex. The main idea of this algorithm is to traverse only a limited set of neighbors (termed as ‘exclusive neighborhood’), whose identifier values are higher than the source vertex identifier of the current enumeration tree. This concept of the exclusive neighborhood facilitates the ESU algorithm to generate unique subgraphs.

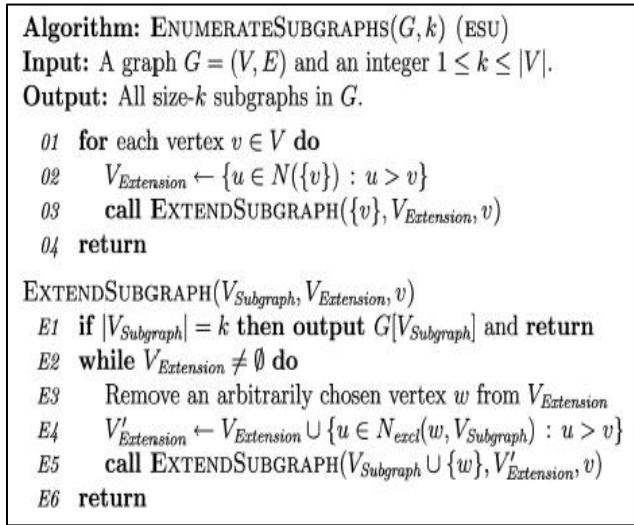


Figure 2.2. Subgraph Enumeration Algorithm^[12].

Figure 2.3 illustrates subgraph enumeration from each vertex is independent of each other. The unique and independent enumeration properties reveal the easily parallelizable potential and encourage to use this algorithm in this work.

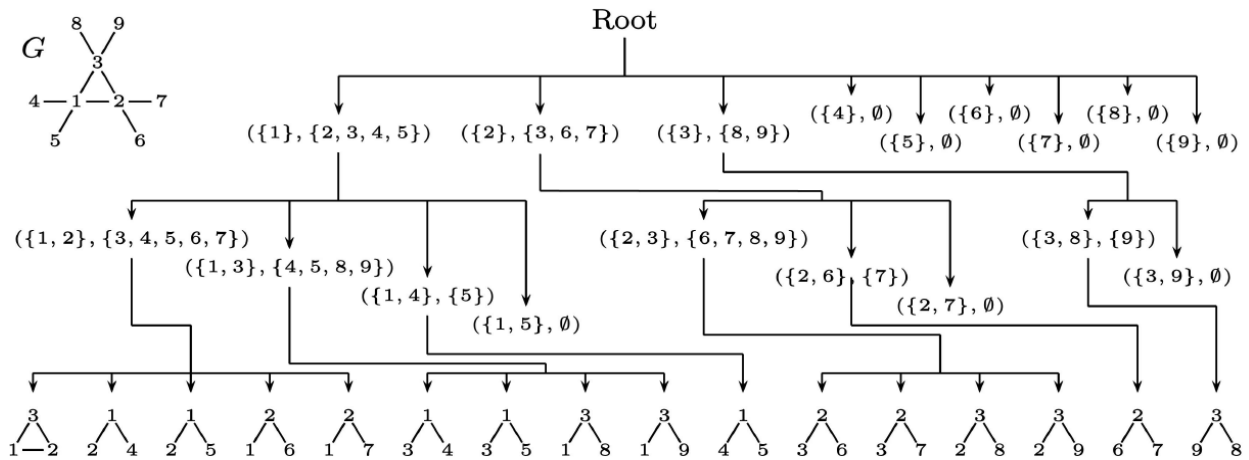


Figure 2.3. Subgraph Enumeration Tree^[12].

Chapter 3. RELATED WORKS

This chapter describes the existing sequential tools and parallel approaches for motif search using different parallel frameworks such as Message Passing Interface (MPI), Hadoop, and MASS. It discusses the limitations of the existing tools and highlights past research strategies adopted in this work.

3.1 SEQUENTIAL NETWORK MOTIF DETECTION TOOLS

Mfinder [5] enumerates all k -sized subgraphs starting from an edge in a brute force manner, which takes longer run time and consumes more memory. Mfinder's exhaustive enumeration version can detect small motifs up to size 4 in both directed and undirected graphs. Mfinder's sampling version implements the edge-sampling strategy proposed in [13] to reduce run time, but results in biased results by finding the same motif pattern repeatedly. Owing to the computational complexity, Mfinder's sampling version can only detect motifs up to size 6.

Fast Network Motif Detection (FANMOD) [6] tool implements ESU algorithm [12] explained in section 2.2. ESU algorithm adopts a vertex-sampling strategy with distinct vertex identifiers. Unlike Mfinder, ESU finds a motif only once and hence it is faster. In contrast to sampled Mfinder, Randomized ESU (RAND-ESU) yields unbiased results by sampling neighbors at each depth with identical visiting probability. FANMOD can detect motifs up to size 8 in undirected, directed, and colored networks.

Network Motif Library (NemoLib) [7] is a general-purpose network-centric approach library used for the detection and analysis of network motifs in undirected and directed networks. Similar to FANMOD, it uses ESU and RAND-ESU algorithms [12] to count motif frequencies referred to as NemoCount. Additionally, it offers NemoProfile that finds motifs concentration on each vertex and NemoCollect that retrieves all instances of input graph that match network motif patterns.

Though these sequential tools [5], [6] can detect motifs up to particular sizes, the motif size limitation imposed by these sequential tools implies the necessity to improve the detection process to discover large motifs and reduce the detection complexity.

3.2 PARALLEL NETWORK MOTIF DETECTION APPROACHES

Parallel Network Motif Extraction. Wang et al. proposed MPI based parallel motif detection [14], in which master process partitions network and broadcast to workers. Worker processes detect candidate motifs in parallel by constructing a Breadth-First Search (BFS) tree to depth $k-1$ for each vertex and finds all k -connected subgraphs. Eventually, the master process gathers results from all workers and deduces the actual motifs with isomorphism check. Although this parallelism performed faster for motifs up to size 4, Wang's parallel implementation was slower than the sequential random sampling method for large motif sizes (5 and 6).

Parallel G-tries. Riberio et al. proposed a data structure Graph reTRIEval (G-trie) [15] that provides an efficient way to store and search the collection of subgraphs. Similar to the prefix tree, G-trie is a multi-way tree where descendant nodes share a common subgraph structure. Later, Riberio et al. proposed a parallel g-tries [16] approach, which parallelized independent and recursive g-trie matching calls by distributing work evenly to different processes using receiver-initiated dynamic load balancing. This parallel subgraph count implementation using OpenMPI achieved almost linear speedup up to 128 processors for motifs of size at most 9.

Iterative Hadoop MapReduce ESU. Verma et al. [17] parallelized ESU algorithm using iterative Hadoop MapReduce. This parallelization consists of three MapReduce jobs, ESU job, Labeler job, and Combiner job. ESU job is executed repeatedly up to input motif size to enumerate all candidate motifs. Labeler job computes canonical labels for candidate motifs. Finally, the combiner job aggregates the candidate motif frequencies. This parallelism suffered from disk I/O overhead for smaller networks. Though parallel performance surpassed their sequential implementation for size 4 motif search on 25714-nodes network, it's slower than the sequential FANMOD [6] speed.

Parallel Network Motif Discovery. Parallel Network Motif Discovery [18] proposed by Riberio et al. parallelized motif search in input and random graphs simultaneously in different processes using OpenMPI. In contrast to the above parallel approaches, this approach parallelized the entire motif detection process using ESU. A distributed work-sharing strategy performed better than the master-worker strategy due to the optimal utilization of all CPU cores. It achieved almost linear speedup up to 128 processors for 1000 random graphs in 6 different networks for motif size 5-8.

3.3 MASS-BASED PARALLEL NETWORK MOTIF ANALYSIS

Agent and Spatial Parallelization of Network Motif Enumeration. Kipps et al. [19] parallelized biological network motif enumeration in three different ways, MASS agent-based, MASS place-based, and MPI based enumeration. Results showed consistent performance improvement for MASS place-based and MPI based enumeration with an increase in the number of threads and number of computing machines. But, MASS agent-based enumeration struggled to enumerate 5.5 million subgraphs for motif size 5 in a 2365-nodes network. This was caused by poor memory usage during agent explosion, and nearly all computing power spent to create and terminate agents.

MASS-based NemoProfile Construction. In research work [20], Andersen et al. extended MASS agent-based, MASS place-based, and MPI based motif enumerations [19] to construct NemoProfile [22], that determines motif concentration on the individual vertices. MASS place-based parallelization proved to be beneficial for NemoProfile, while MASS agent-based parallel implementation encountered memory exhaustion for motif size 4 in a 5193 network.

MASS Agent Management and Performance Features. To enhance performance and mitigate memory overhead issues reported by [19] and [20], the below features were added to MASS. *Parallel File I/O* feature to relieve the main program from the burden of distributing input data to all computing nodes in the cluster. *Asynchronous Automatic Agent Migration* feature to reduce communication with user application during each iteration of agent management functions. *Agent Population Control* feature to control the number of active agents by serializing agents that exceed population limit value. Serialized agents are activated once the current agent population drops below the population threshold. This work evaluates these features for motif detection problem.

Similar to tools [6], [7], [17], [18], [19], [20], this work employs ESU algorithms to enumerate motifs and relies on NautyTraces [21] program to test graph isomorphism. Sequential tools [6], [7] are used to prove correctness and evaluate performance, while previous MASS agent-based enumeration [19] serves as a parallel baseline for this work. Furthermore, existing parallel network motif implementations with OpenMPI [16], MapReduce [17], and MASS [19] are limited to motif frequency count and don't offer the entire network motif detection functionality. This project implements the complete motif detection process and identifies the statistically significant motifs.

Chapter 4. PARALLELIZATION OF BIOLOGICAL NETWORK MOTIF DETECTION

This chapter describes the overall system flow to detect network motifs of the desired size (≥ 3) in an undirected or directed target network. It also explains MASS and Spark approaches along with implementation-specific fine-tuning performed to improve speed and reduce memory use.

4.1 SYSTEM FLOW

The network motif detector comprises of five distinct modules, graph parser, target graph analyzer, random graph generator, random graph analyzer, and statistical analyzer. Figure 4.1 demonstrates the system flow that portrays the parallel or sequential execution of the modules. Both MASS and Spark implementations follow the identical execution pattern to develop compatible versions for consistent evaluation of parallelism.

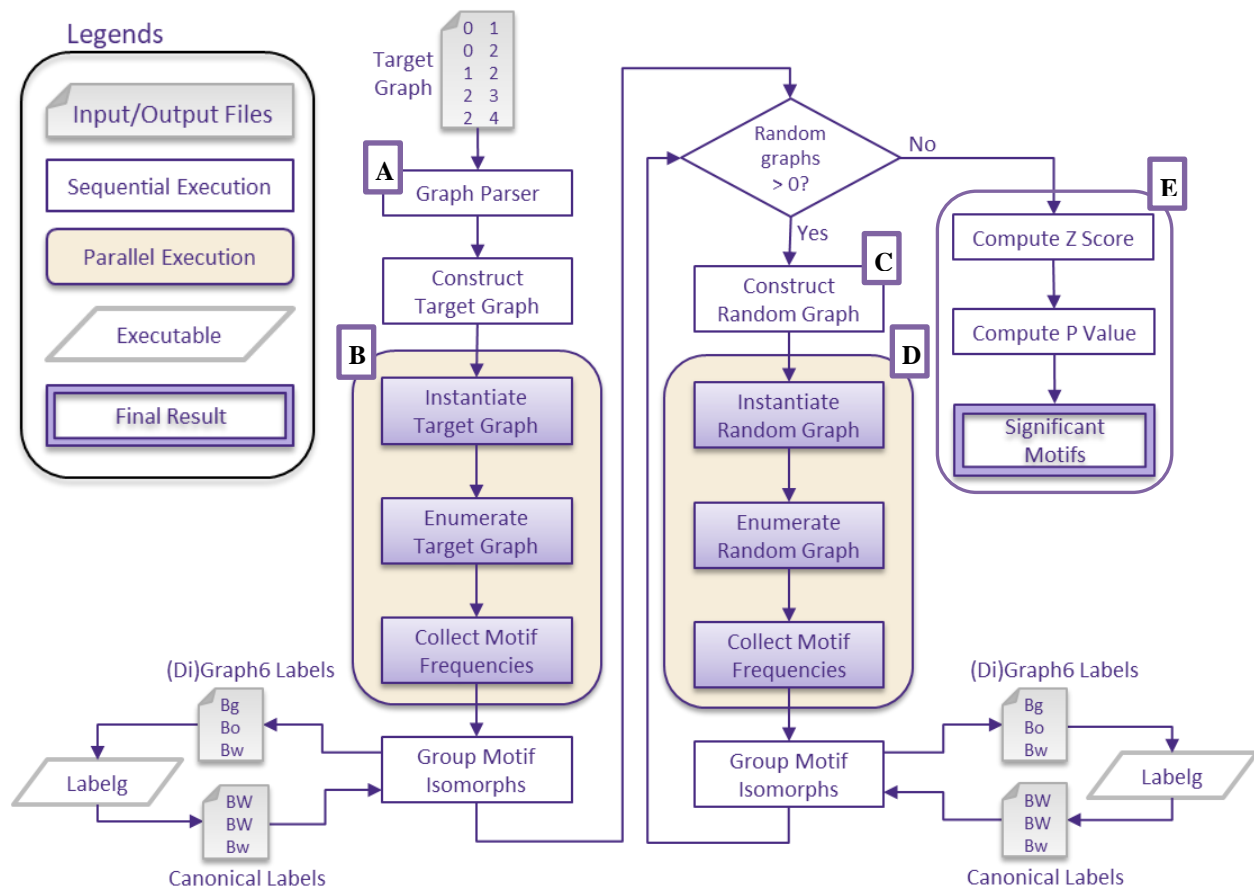


Figure 4.1. System Flow for Network Motif Detection.

Graph Parser (A in Figure 4.1). This module parses the target graph represented in the edge list format and constructs graph in adjacency list representation. With this representation, out and in neighbors' information for all vertices are initialized. Undirected graphs store an edge as 'out' neighbor in both vertices, while directed graphs store neighbor information based on the edge direction. Graph parser computes target graph's out-degree, in-degree sequences that are required to generate random graphs. During graph construction, graph parser eliminates self and parallel edges to avoid unnecessary computations. MASS parallel I/O feature evaluated in section 5.5.1 clarifies the decision to execute the graph parser module serially.

Target Graph Analyzer (B in Figure 4.1). It performs exhaustive enumeration of the target graph to identify all candidate motifs for the given motif size. Enumeration happens in parallel across all the computing machines utilized for the execution. The target graph analyzer first instantiates all vertices with their corresponding neighbor information obtained from graph parser. Then, it executes the ESU algorithm shown in Figure 2.2 simultaneously from all the vertices. Once the subgraph of input motif size is enumerated, compact graph6 or digraph6 representation is computed and returned. Appendix A describes the procedure to convert undirected and directed subgraph structures to graph6 and digraph6 formats respectively. At the termination of all parallel enumerations, the target graph analyzer gathers all motif sized subgraphs in graph6 or digraph6 format. Finally, isomorphic subgraph occurrences are grouped by providing graph6 or digraph6 representation to Labelg program [23], and resultant canonical labels of candidate motifs are saved along with respective frequencies. Figure 4.2 depicts isomorphism grouping, where the Labelg assigns the same canonical label (BW) for different graph6 labels (Bg, Bo) with identical structure.

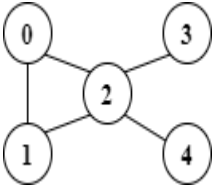
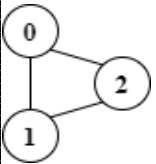
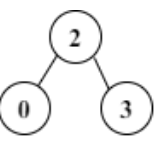
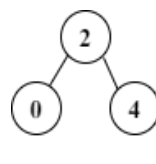
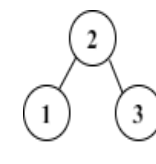
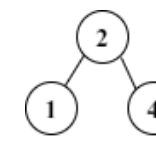
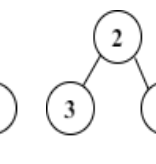
Input Graph	Enumerated Subgraphs for Motif Size 3						
							
Graph6 Label	Bw	Bg	Bg	Bg	Bg	Bo	
Canonical Label	Bw	BW	BW	BW	BW	BW	

Figure 4.2. Graph6 and Canonical Labels for Motif Size 3.

Random Graph Generator (C in Figure 4.1). Random graphs are generated from the input graph by preserving the degree distribution of the vertices in the input graph. This work generates degree-

preserving random graphs using the configuration model described in [24]. Random graph generator fetches degree distribution sequence from graph parser. Degree distribution sequence contains a list of vertex identifiers created by repeatedly adding each vertex identifier up to its degree value (number of neighbors). Random graph generator shuffles degree distribution sequence and repeatedly picks a random pair of vertices as an edge for the random graph. Consequently, the generated random graph may be a connected or disconnected graph with lesser degree distribution than expected, due to the exclusion of self edges and parallel edges.

Random Graph Analyzer (D in Figure 4.1). This module employs the RAND-ESU algorithm to perform approximate enumeration based on the input sampling probabilities. Instead of traversing all neighbors, it selectively traverses the limited set of neighbors at each ESU tree-level shown in Figure 2.3. RAND-ESU algorithm reduces the time taken to compute the frequency of candidate motifs in a large number of random graphs. Similar to the target graph analyzer, this module executes in parallel across all computing machines. Random graph analyzer filters non-candidate motifs enumerated in the randomized networks and gathers frequency of all candidate motifs.

Statistical Analyzer(E in Figure 4.1). As a final step, the statistical analyzer computes the Z-score and p-value for all the candidate motifs, using the mathematical relations stated in section 2.1. It executes sequentially in the master computing machine and displays the result to the user, as seen in Figure 4.3.

MASS Network Motif Detection					
Motif	Target Freq	Random Mean Freq	Random Std Dev	Z-Score	P-Value
DBw	3.1554 %	4.2259 %	0.00587369	-1.822	0.979
D`[10.6065 %	4.2580 %	0.00741453	8.562	0.000
Dd[1.0960 %	0.4702 %	0.00105330	5.942	0.000
DDw	30.4188 %	37.8394 %	0.01098371	-6.756	1.000
D?{	2.4968 %	3.6167 %	0.00398432	-2.811	0.998
DR{	0.6340 %	0.0552 %	0.00040524	14.283	0.000
DqK	0.6979 %	0.7847 %	0.00098705	-0.879	0.809
DN{	0.3195 %	0.0034 %	0.00007690	41.105	0.000
DJk	3.2734 %	0.5056 %	0.00204057	13.564	0.000
D@[4.2809 %	2.2804 %	0.00368407	5.430	0.000
DB{	2.9244 %	0.5024 %	0.00195762	12.372	0.000
DJ{	1.1255 %	0.0196 %	0.00036677	30.154	0.000
DF{	0.2359 %	0.0094 %	0.00011231	20.168	0.000
Dr{	0.0393 %	0.0017 %	0.00003945	9.537	0.000
DFw	0.0295 %	0.0783 %	0.00037224	-1.310	0.944
D`{	0.5406 %	0.1187 %	0.00052473	8.041	0.000
D~{	0.0147 %	0.0000 %	0.00000000	Infinity	0.000
D@s	28.6100 %	40.6115 %	0.01446442	-8.297	1.000
Dr{	0.1425 %	0.0265 %	0.00017176	6.758	0.000
D^{	0.1278 %	0.0001 %	0.00000808	158.116	0.000
DD[9.2303 %	4.5926 %	0.00805014	5.761	0.000

Figure 4.3. MASS Execution Result for Motif Size 5 in 1000 Random Graphs

4.2 MASS IMPLEMENTATION

MASS Places and Agents, introduced in section 1.2.1, are used to parallelize subgraph enumeration in target and random graphs. MASS places map to graph vertices that hold neighbors' information throughout the lifetime of the program. MASS agents are responsible for enumerating subgraphs of given motif size from each place (vertex) in parallel. Graph parser assigns zero-indexed vertex identifiers to map vertex identifier to the MASS place index effortlessly.

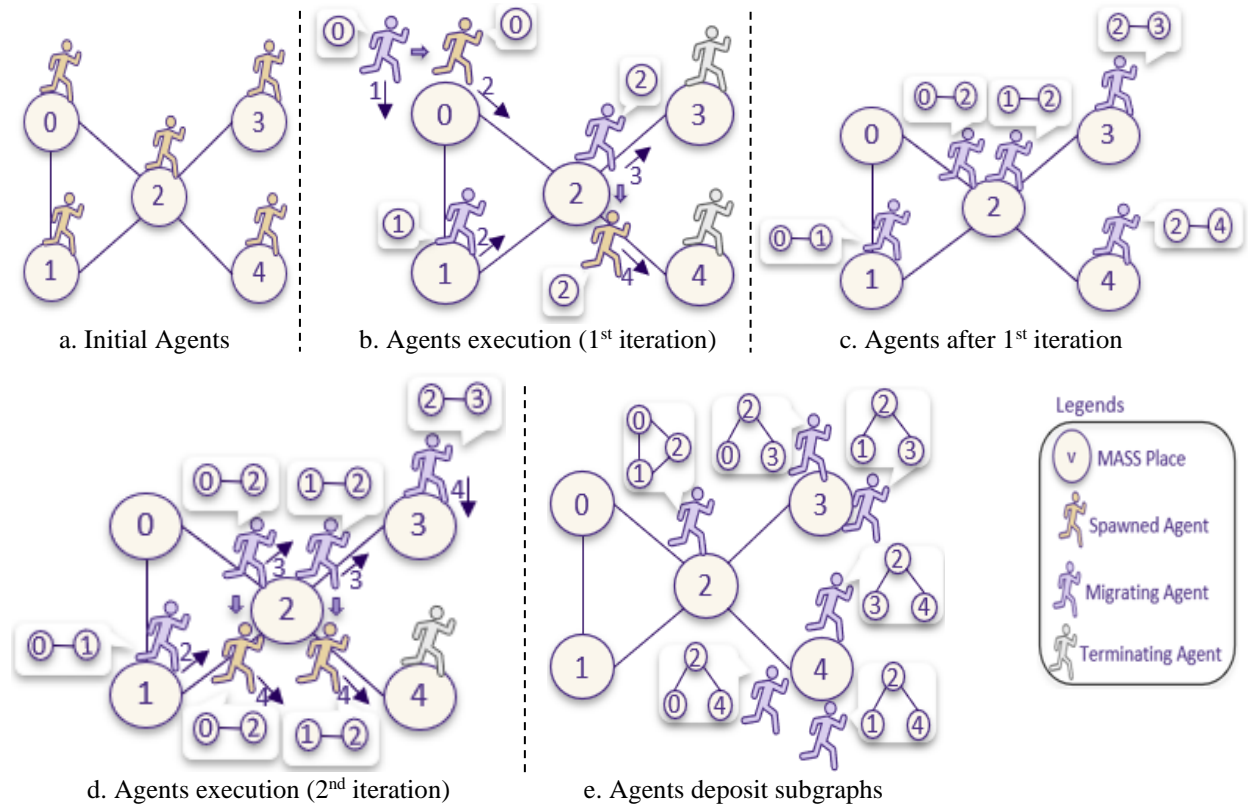


Figure 4.4. MASS Agents Execution for Motif Size 3

Initially, an agent is populated at every place, as seen in 4.4.a. At every iteration of agent function execution, agents examine the neighborhood and perform either of the following actions. If valid neighbors exist, it spawns child agents and migrates itself to the first valid neighbor or terminates otherwise. After repeating this procedure for motif size iterations, agents deposit enumerated subgraph structure at the current residing place (4.4.e) before terminating itself. Finally, the driver gathers subgraphs from all the places in parallel. Figure 4.4.d. demonstrates agent at vertex '3' migrates to vertex '4', although vertices '3' and '4' aren't direct neighbors (directly connected). MASS supports flexible agent movement, unlike Spark's graph processing library discussed later.

4.2.1 MASS Place and Agent Data Structure

MASS agent movement pattern described in the previous section is comparable to Kipps et al. [19] agent-based subgraph count implementation. As discussed in related works, MASS agent-based network motif analysis implementations [19], [20] faced memory overhead. Consequently, the main focus of this agent-based implementation is to reduce memory use at the application level.

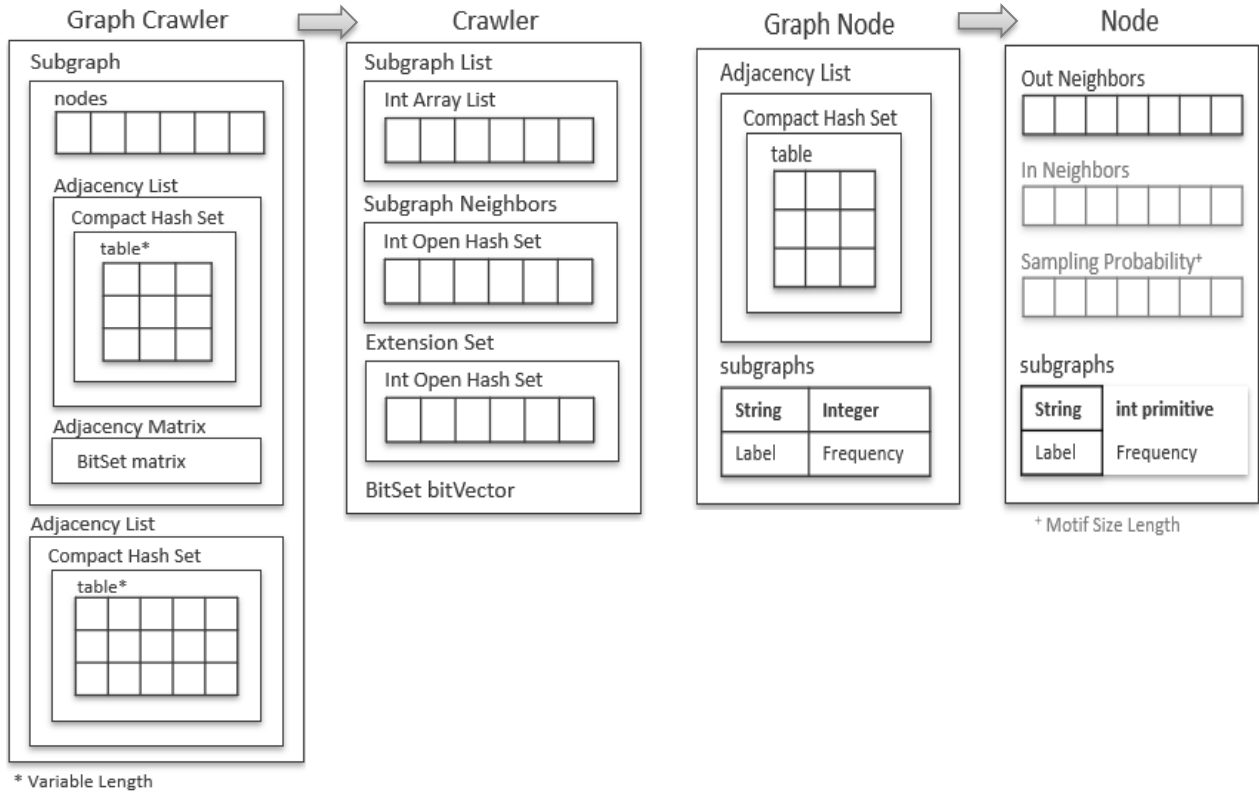


Figure 4.5. MASS Place and Agent Data Structure

To differentiate Kipps et al. implementation [19] from the current implementation, the former version termed as ‘Old MASS Network Motif’ and the latter version termed as ‘New MASS Network Motif’ throughout this paper. Figure 4.5 depicts the data structure transition from Old MASS Network Motif (Graph Crawler represents Agent and Graph Node represents Place) to New MASS Network Motif (Crawler represents Agent and Node represents Place). Additionally, New MASS network motif includes ‘In Neighbors’ and ‘Sampling probability’ fields to support directed graphs and RAND-ESU algorithm, which aren’t supported by old MASS network motif.

As seen in Figure 4.5, Old MASS network motif used wrapper objects and multiple levels of nested object references, while New MASS network motif version eliminated wrapper objects and used primitive types. The primitive type-specific collections are used instead of Java built-in collection (HashSet<Integer>) or user-defined collection (CompactHashSet) to avoid boxed

objects or multiple levels of embedded object references. Thus, minimal use of java objects enabled New MASS network motif version to consume less heap space and improve execution speed with reduced garbage collection pauses for memory management tasks.

New MASS network motif utilized Fastutil library [25] due to its compatibility with standard java library and easy to use well-documented APIs. More importantly, as per evaluations [26], [27], Fastutil performs consistently faster with less memory footprint when compared against other Java type-specific collection libraries such as Trove, Goldman Sachs Collections, Koloboke, and High-Performance Primitive Collections for Java (HPPC). Furthermore, Spark documentation [28] also recommends Fastutil for tuning data structures without wrapper objects and pointer-based data structures.

4.2.2 *MASS Performance Tuning*

New MASS network motif implementation intended to minimize data within each agent primarily to keep JVM heap memory utilization in control during agent expansion in order of millions. And additionally, reduce the time spent to serialize and deserialize agent data during migration across computing nodes. This section captures performance improvements incorporated in New MASS network motif implementation to reduce the overall memory footprint of the application and improve execution speed.

- *Preferred Primitive Types over Primitive-Wrapper Objects* to reduce memory space and avoid autoboxing/unboxing performed during primitive to non-primitive type conversions and vice versa.
- *Replaced Built-in Java Collection with Fastutil's Collection.* Built-in Java collections such as HashMap, HashSet, and ArrayList consume enormous memory with an increase in the collection size and tightly couples the internal data structure used. To reduce memory overhead and benefit from using different internal data structures such as array, AVL tree, RB tree, open hash, and custom hash, Fastutil [25] collections are used. In MASS implementation, the number of agents increases exponentially for large motif size and large graph size. With primitive type-specific collections, each agent contained considerably lesser data.

- *MASS Asynchronous Agent Migration.* Replaced Agent’s callAll followed by manageAll with doAll for ‘motif size’ iterations to reduce time incurred by returning control to the driver program in between the successive function calls for each iteration.
- *Reduced HashMap with String Key.* The initial version maintained data uniquely for each motif in multiple hash maps (8 hash maps) with motif’s canonical label string as a key. These hashmaps are restructured to ‘Motif’ class to reduce memory space occupied by the recurrent canonical label string key references stored across multiple maps.
- *Moved Agent’s data to Places.* Input motif size and sampling probabilities are stored in the agent initially. These input data occupied huge memory with the creation of millions of agents. Input data used by agents are stored in all places and agents fetch data from place upon arrival to the place, thereby reducing memory utilized during the agent expansion.
- *Eliminate redundant data in Agent.* Early implementation maintained source vertex, next migration vertex fields within each agent, which are later dropped and computed from subgraphList at run time.
- *Avoid passing unnecessary data during Agent duplication.* At the last iteration of the enumeration process, spawned agents migrate to the last vertex to discover its connectivity to all other vertices of the enumerated subgraph and deposit subgraph structure at last vertex. Hence, agents spawned in the last iteration require only subgraph data and not subgraph neighbors and extension data. This unnecessary data prevention reduced memory and time taken to duplicate the data from parent to child agents.

These performance tuning has reduced execution time significantly by minimizing agent size, reducing heap utilization, decreasing serialization-deserialization time, and notably lowering garbage collection cycles.

4.3 SPARK IMPLEMENTATION

Spark implementation to parallelize subgraph enumeration in target and random graphs uses basic Spark abstraction, Resilient Distributed Datasets (RDD). Graph vertices map to VertexRDD and enumerated subgraphs map to SubgraphRDD, as viewed in Figure 4.6. In contrast to MASS that allows vertex addition to enumerated subgraph (agent) dynamically, new SubgraphRDD has to be created every time to add a vertex to the enumerated subgraph, due to the immutable nature of Spark RDDs.

In Spark implementation, subgraph enumeration happens through a series of narrow-dependency transformations such as `mapToPair`, `values`, `flatMap`, followed by an expensive wide-dependency transformation (`reduceByKey`), that shuffles data across all the computing nodes. Finally, the driver program gathers the frequency of all enumerated subgraphs either in `graph6` or `digraph6` format via `collectAsMap` Spark action.

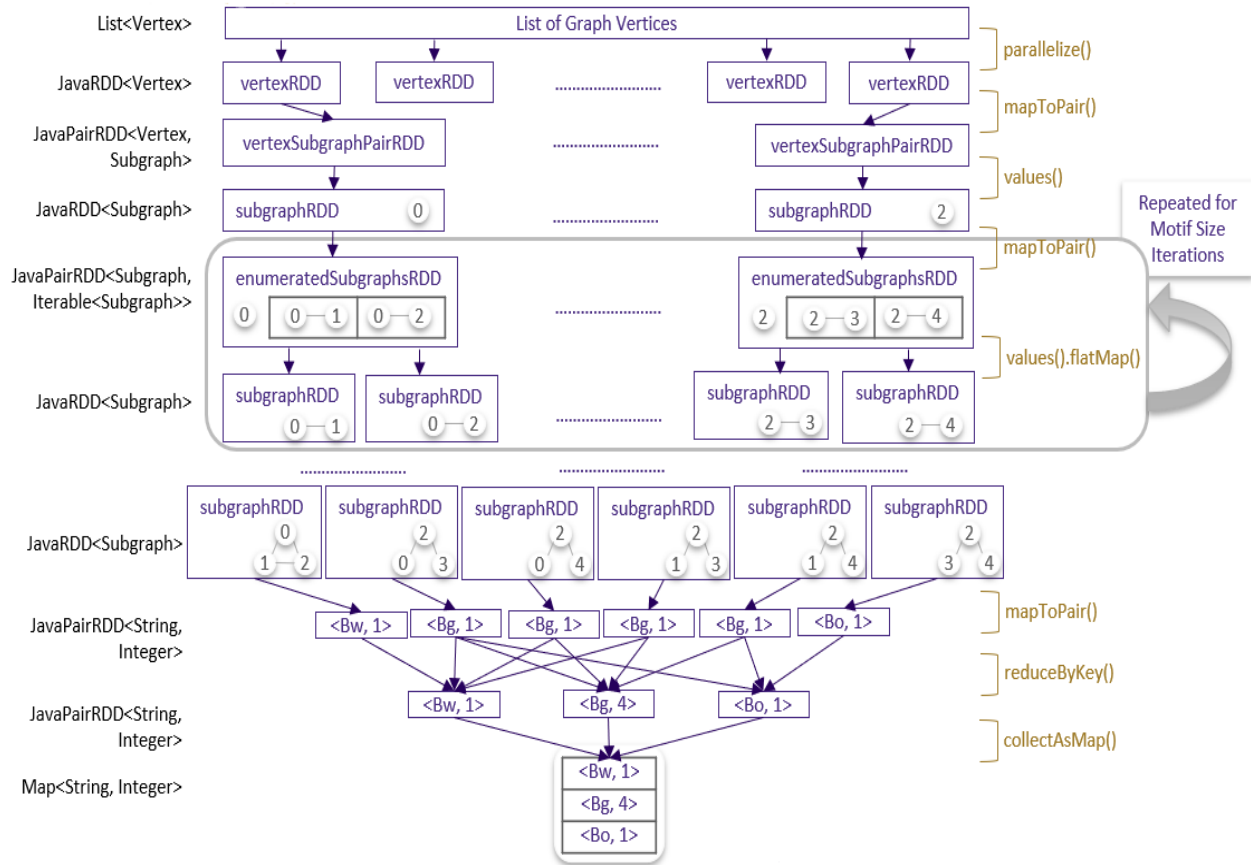


Figure 4.6. Spark Implementation of Subgraph Enumeration

Spark implementation doesn't utilize Graphx [29], Spark's graph processing library optimized for distributed graph operations. Pregel [30] is a vertex-centric graph processing model developed by Google for large-scale graph processing. Although Graphx provides an optimized Pregel like operator for iterative computations, restrictions imposed to optimize graph computation evolved as an obstacle to utilize for the subgraph enumeration algorithm. As pointed out in [31], Graphx Pregel operator prohibits direct communication between vertices that are not adjacent in the graph to reduce data movement. But, subgraph enumeration requires direct communication between non-adjacent vertices. Consequently, Spark implementation employed basic RDD instead of Graphx operators to enumerate subgraphs efficiently.

4.3.1 Spark Performance Tuning

The initial version of spark implementation performed slower than sequential tools [6], [7] because of the under-utilization of the cluster resources to execute tasks in parallel, and data serialization overhead that impacted network bandwidth.

- *Tune Default Parallelism Level.* In Spark, RDDs [32] are created either by parallelizing an existing collection from the driver program via `parallelize()` or by reading from the file system via `textFile()`. Later RDD creation splits input file into multiple partitions depending on file size so that Spark can run tasks in parallel across the partitions. But, Spark implementation depicted in Figure 4.6 creates initial `vertexRDDs` by parallelizing vertices collection. Automatic RDD partitioning done by Spark turns out to be inefficient in this scenario. Hence, it is crucial to split `vertexRDD` into multiple partitions to increase task parallelism of all narrow-dependency transformations applied successively on `vertexRDD`. Thus, the default parallelism level was fine-tuned by configuring `'spark.default.parallelism'` property to 3 tasks per CPU core in the cluster, as suggested in [28].
- *Tweak ReduceByKey Partitions.* `ReduceByKey` inherits the number of partitions either from the largest parent RDD or `'spark.default.parallelism'` property. It is essential to tweak the partition value for reduce tasks, that involve shuffle operation. Smaller partition count might cause out of memory issues when large RDD partitions don't fit in memory, while larger partition count might incur the repartition overhead. After experimenting with different partition values (that are multiples of the number of CPU cores in the cluster) on different graphs, optimal partition value has been identified to be 48 for the test cluster.
- *Eliminate unnecessary spark actions used for debugging.* Listing 1 shows the initial spark code that applied count action to find the number of subgraphs enumerated in every iteration. Following the spark's job execution model described in section 1.2.2, multiple jobs are created, one for each count action up to motif size.

```
1 for (int size = 1; size <= motifSize; size++) { // Iteratively enumerate subgraphs
2   System.out.println(subgraphRDD.count() + " subgraphs enumerated for motif size: " + size );
3
4   enumerateSubgraphsRDD = subgraphRDD.mapToPair(
5     (PairFunction<Subgraph, Subgraph, Iterable<Subgraph>>) subgraph -> {...});
6
7   subgraphRDD = enumerateSubgraphsRDD.values().flatMap(
8     (FlatMapFunction<Iterable<Subgraph>, Subgraph>) subgraphs -> {...});
9 }
```

Listing 1. Spark's code to get enumerated subgraphs count

Figure 4.7 depicts the re-evaluation of RDD lineage from the earliest parallelize transformation in each stage. Developers need to pay more attention while using spark actions to debug intermediate results. Thus, the removal of count action eliminated RDD lineage re-evaluation and improved execution speed. In case graph computation requires spark action, then RDDs can be cached in on-heap memory or persisted in disk or off-heap memory based on input storage level passed to persist() API to overcome the expensive re-evaluation of RDD lineage.

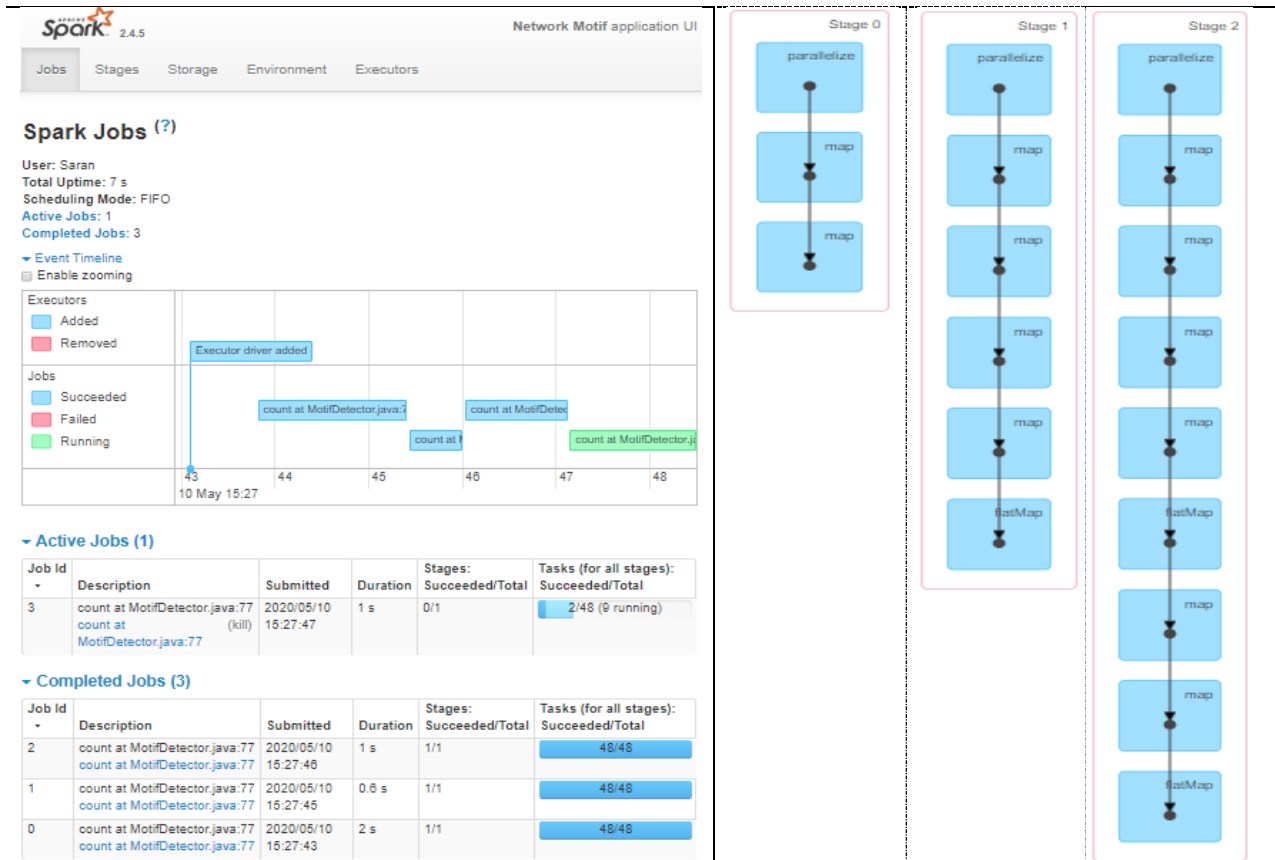


Figure 4.7. Spark's RDD Lineage Re-evaluation for Spark Action

- *Reduce Data Serialization Overhead.* Though default Java serialization is flexible, it results in large serialized data. But Kryo library serializes objects faster and at most 10x compact than default java serialized objects. The only downside of the Kryo serializer is to register user-defined classes. Spark [28] recommends Kryo serializer for network-intensive distributed applications. Thus, Spark implementation improved network performance by transferring compact objects serialized using the Kryo library.

These fine-tuning efforts enabled Spark implementation to outperform sequential tools [6], [7] for large motif sizes in small graphs, and even small motif sizes in large graphs.

Chapter 5. RESULTS

This chapter presents the experimental results of MASS and Spark parallel implementations discussed in chapter 4, and compares execution performance with sequential tools FANMOD [6], NemoLib [7], and parallel Kipps et al. [19] MASS version. Being a GUI based tool, FANMOD tests are carried out in Intel i7-8550U Windows laptop with 16 GB RAM. Except for FANMOD, all other implementations are tested in the environment described in 5.1. This chapter follows the same naming convention, 'Old MASS' refers to Kipps et al. [19] MASS version, and 'New MASS' refers to the MASS implementation described in 4.2. The correctness of the results is verified by comparing parallel output with the sequential output of motif detection presented in Appendix B.

5.1 EXECUTION ENVIRONMENT

Experiments are conducted in a cluster of 8 computing nodes made available by the University of Washington Bothell. Among 8 computing nodes, 4 nodes have 8-core 2.33GHz CPU (Intel Xeon E5410) with 16GB memory and the remaining 4 nodes have 4-core 2.66GHz CPU (Intel Xeon 5150) with 16GB memory. The latest stable version of software libraries used in this work are as follows, MASS Java [33] core version 1.2.1, NemoLib Java [34] version 2, Apache Spark [35] version 2.4.5, Nauty Traces [36] version 2.6 (r12), and Fastutil [37] version 8.3.1. NemoLib, Old MASS, and New MASS java applications are configured with 4GB initial heap and 12GB maximum heap space. The Spark version is configured to utilize 8GB memory for driver and 2GB memory for executor processes.

5.2 INPUT DATASETS

This section presents the graph properties of the undirected and directed real network datasets used for experiments. As stated in [2], typical biological networks are often sparse, with an average degree (ratio of the number of edges to the number of vertices) between 1 and 3. Hence, synthetic graphs are used to evaluate performance and memory usage of the implementations in dense networks.

5.2.1 Real Biological Network Datasets

Table 5.2 lists three undirected and two directed real datasets used to conduct experiments. These downloaded input datasets are in different graph formats such as Graph Modeling Language (GML), Pajek, and Weighted Edge-List format. Different input graphs formats are converted to the Edge-List format expected by the current implementation using a python script. This script uses open-source python library NetworkX [38], for format conversion and graph visualization. Figure 5.1 depicts the dolphin network visualized using python script.

Table 5.2. Real Network Graph Properties.

Real Datasets	Vertices	Edges	Directed?	Connected Components
Dolphin [39]	62	159	No	1
Power [40]	4,941	6,594	No	1
DIP [41]	26,695	73,085	No	1,204
Yeast [42]	688	1,078	Yes	11
Gnutella P2P [43]	6,301	20,777	Yes	2

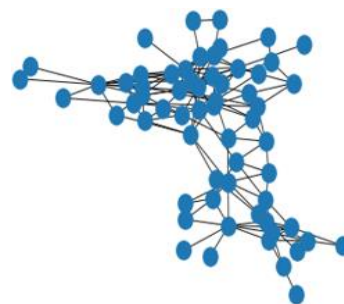


Figure 5.1. Dolphin Network

5.2.2 Undirected Synthetic Graphs

Undirected synthetic graphs (single connected component) are generated using NetworkX's API `fast_gnp_random_graph`, which implements an efficient version of the Erdős-Rényi random graph generation model [44]. Table 5.3 lists graph properties along with edge creation probability used to generate these graphs.

Table 5.3. Undirected Synthetic Graph Properties.

Vertices	Edges	Edge Creation Probability	Maximum Degree	Average Degree
1,024	52,116	0.1	134	50.89
2,048	83,925	0.04	118	40.97
4,096	125,668	0.015	90	30.68
8,192	234,678	0.007	86	28.64

5.3 EXECUTION PERFORMANCE

This section compares the performance of sequential tools (FANMOD, NemoLib) against parallel implementations (Old MASS, New MASS, and Spark) for target graph enumeration. Listing 2 shows execution time measured with `System.currentTimeMillis()` uniformly in all versions (except FANMOD). Both MASS versions utilized 8 computing nodes and 4 threads per computing node.

```

1 long startTime = System.currentTimeMillis();
2
3 // Execute Network Motif Detection Application
4
5 System.out.println("Overall execution time = " + (System.currentTimeMillis() - startTime) + " milliseconds");

```

Listing 2. Execution time measurement code snippet

5.3.1 Real graphs performance comparison with increasing motif sizes

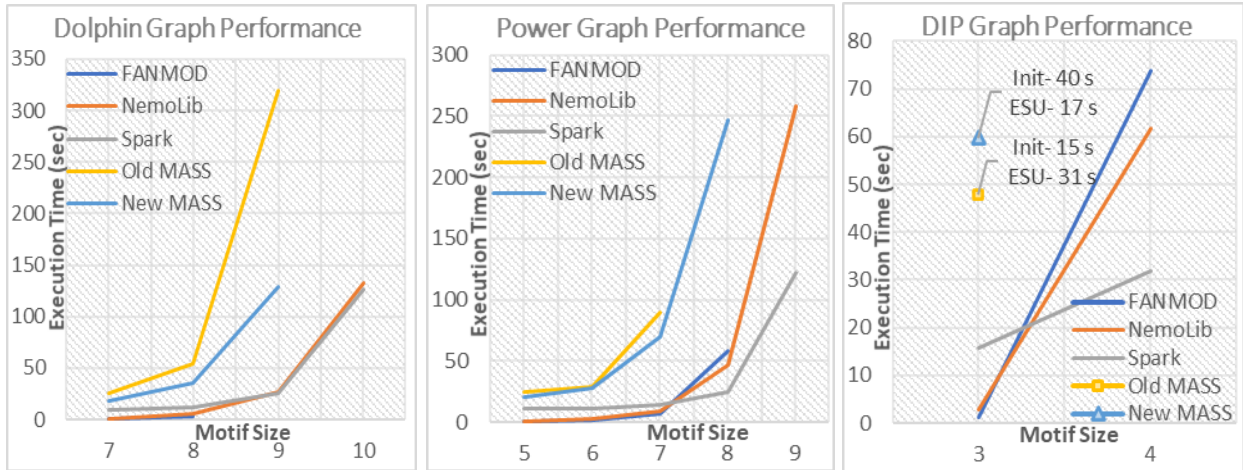


Figure 5.2. Undirected Real Networks Performance.

Undirected Real Graphs Performance. Figure 5.2 shows New MASS version performs consistently faster than the Old MASS version for small (Dolphin) and medium (Power) sized undirected graphs. But in large (DIP) graph, Old MASS runs faster than New MASS implementation for small motif size (3). The phase-wise execution time analysis reveals the parallel subgraph enumeration (ESU) happens faster in New MASS, but New MASS takes more time to initialize the target graph compared to Old MASS implementation. Due to additional data structures incorporated in the New MASS version to support directed graphs and approximate enumeration (presented in 4.2.1), New MASS takes more time to initialize larger graphs.

Though New MASS improved over Old MASS, MASS versions lag behind Spark and sequential tools in all experiments. Spark performance significantly improves with an increase in motif size as well as graph size. Figure 5.2 depicts the parallel Spark version attained almost 2x speedup in medium-sized power graph (motif size 9) and large DIP network (motif size 4). FANMOD faced maximum motif size limitation (> 8), and MASS versions encountered memory overhead to enumerate larger subgraphs. Old MASS can enumerate up to 12.4 million subgraphs while New MASS can enumerate up to 33.4 million subgraphs. In conclusion, spark parallelization improved speed and enhanced scalability over the sequential tools for undirected real graphs.

Table 5.4. Enumerated Subgraphs Count in Undirected Real Graphs.

Target Graph	Motif Size	#Subgraphs	Target Graph	Motif Size	#Subgraphs	Target Graph	Motif Size	#Subgraphs
Dolphin	7	550,428	Power	5	268,694	DIP	3	1,859,101
Dolphin	8	2,683,740	Power	6	1,260,958	DIP	4	89,371,477
Dolphin	9	12,495,833	Power	7	6,340,413			
Dolphin	10	55,824,707	Power	8	33,494,650			
			Power	9	183,453,978			

Table 5.4 lists the number of subgraphs enumerated for increasing motif sizes in undirected real datasets. Dolphin graph performance shows New MASS speedup (compared to Old MASS) increased from 1.5x to 2.4x with increased enumerated subgraphs (agents) from 2.6 million to 12.4 million. With compact agent structure implementation, New MASS minimized CPU time spent on memory management tasks, while operating with millions of agents.

Directed Real Graphs Performance. This subsection compares performance for directed real graphs. Tests were not conducted in Old MASS, as it doesn't support directed graphs.

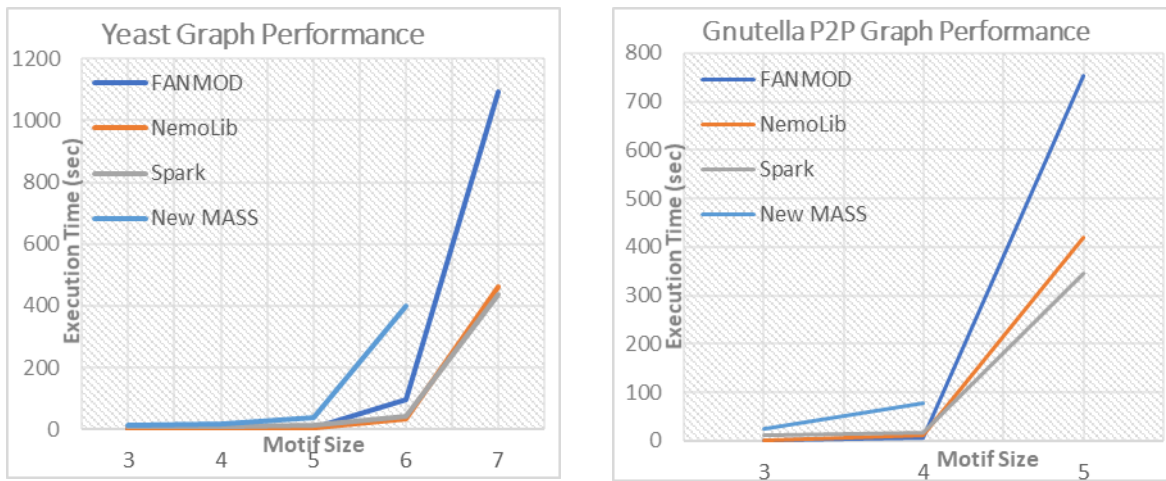


Figure 5.3. Directed Real Networks Performance.

Figure 5.3 illustrates New MASS performs comparably equivalent to Spark for small motif size (3). But for motifs (≥ 4), agent creation and management cost increases with an exponential increase in agent population and consequently impacts MASS performance. This is because a large number of subgraphs are enumerated even for small motif size in directed graphs (for instance, 10 million subgraphs are enumerated for motif size 4 in P2P), as seen in Table 5.5. Similar to undirected graphs, MASS version encountered memory limitation beyond motif size 6 and 4 in Yeast and P2P graphs respectively. Figure 5.3 shows performance only upto certain motif sizes because parallel implementations support subgraph enumeration up to maximum integer value of

$2^{31}-1$ (2,147,483,647). NemoLib, the sequential baseline, also exhibits the same limitation. Figure 5.2 and Figure 5.3 depict parallel Spark implementation outperformed sequential tools, NemoLib and FANMOD for large motif sizes in both undirected and directed real graphs.

Table 5.5. Enumerated Subgraphs Count in Directed Real Graphs.

Target Graph	Motif Size	#Subgraphs	Target Graph	Motif Size	#Subgraphs
Yeast	3	13,150	P2P	3	341,267
Yeast	4	183,174	P2P	4	10,031,003
Yeast	5	2,508,149	P2P	5	391,618,916
Yeast	6	32,883,898			
Yeast	7	416,284,878			

5.4 MASS SPEEDUP AND MEMORY REDUCTION

This section evaluates the impact of MASS performance improvement by comparing speedup and memory reduction on dense synthetic graphs generated using the technique described in 5.2.2. All tests used 8 computing nodes and 4 threads per computing node and enumerated motif size 3.

5.4.1 Synthetic graphs performance comparison with increasing graph sizes

Table 5.6. New MASS Speedup in Undirected Synthetic Graphs.

Graph	Subgraphs	Old MASS	New MASS	Speedup
1,024	4,949,229	87.022	43.381	2.00
2,048	6,693,448	94.884	51.472	1.843
4,096	7,629,910	117.717	53.525	2.19
8,192	13,389,518	303.443	79.674	3.80

Table 5.6 demonstrates New MASS achieved an average of 2.4x speedup than Old MASS for dense synthetic graphs. Fine-tuning MASS version explained in 4.2.2 improved performance of New MASS over Old MASS. Although application-level optimization enhanced New MASS speed, increasing motif size beyond the value presented in this section affects the MASS execution. This is due to the memory exhaustion caused by millions of agents that fill up the heap space. In this scenario, CPU resources are utilized for garbage collection rather than algorithm computation, which impacts performance. Thus, current MASS implementation is limited by the maximum heap availability on the cluster machines.

5.4.2 Memory Reduction

This section measures the memory usage of Old MASS and New MASS implementations and evaluates reduction achieved by the New MASS version. Listing 3 shows memory consumption measured by finding the difference between total memory and available free memory.

```

1 long MemAtStart = Runtime.getRuntime().totalMemory()-Runtime.getRuntime().freeMemory();
2
3 // Execute Network Motif Detection Application
4
5 long MemAtExit = Runtime.getRuntime().totalMemory()-Runtime.getRuntime().freeMemory();
6 long actualMemUsed = MemAtExit-MemAtStart;
7 // Memory Utilized
8 System.out.println("Memory Used: "+ actualMemUsed + " Bytes (" + actualMemUsed/1048576 + " MB");

```

Listing 3. Memory utilization measurement code snippet

Table 5.7 displays memory usage (in MB) of MASS versions to detect motif size 3 in undirected synthetic graphs. As per Java SE documentation [45], approximate values returned by totalMemory() and freeMemory() methods (in Listing 3) vary over time depending on garbage collection execution in the host. Owing to the unreliable nature of this measurement, a series of experiments (5 executions at different times) are conducted to find the overall memory usage trend.

Table 5.7. Memory Reduction in MASS Implementation

Graph Version	1024V_52116E		2048V_83925E		4096V_125668E		8192V_234678E	
	Old MASS	New MASS	Old MASS	New MASS	Old MASS	New MASS	Old MASS	New MASS
Execution 1	1165	487	2115	347	2442	906	2687	1013
Execution 2	1161	472	2163	368	2336	942	2509	1070
Execution 3	1048	483	2119	350	2382	927	2651	1048
Execution 4	1223	465	2167	369	2402	983	2599	1180
Execution 5	1129	448	2017	397	2426	927	2417	955
Min (in MB)	1048	448	2017	347	2336	906	2417	955
Avg (in MB)	1145.2	471	2116.2	366.2	2397.6	937	2572.6	1053.2
Max(in MB)	1223	487	2167	397	2442	983	2687	1180
Reduction	2.431422505		5.778809394		2.558804696		2.442650968	

The memory usage trend exhibits variation in the range of 39~270 MB across the executions. Based on average memory usage, it can be concluded that the New MASS version achieved at least 2.4 times memory reduction in dense graphs. Compact data structure and minimal data carried by agents in New Mass version reduced memory usage, which in turn decreased garbage collection pauses and enhanced performance. Consequently, reducing memory footprint at the application level can significantly improve speed and enhance scalability to deal with large-scale data.

5.5 MASS FEATURE EVALUATION

This section evaluates three MASS features, parallel file I/O, agent population control, and asynchronous agent migration features that are tried in New MASS implementation.

5.5.1 MASS Parallel I/O Feature

MASS Parallel I/O feature expects each line in the input graph to have the same alignment so that the file can be partitioned and read in parallel from the computing nodes. To meet even alignment constraint, each neighbor data need to be filled with -1 and spaces up to maximum neighbors as seen in Figure 5.4 and end up creating a huge input file. As visualized from Table 5.8, MASS Parallel I/O drastically increases the input file size for large graphs. Additionally, it expects the number of input lines to be an exact multiple of the number of computing nodes to partition the input file accurately. Although MASS Parallel I/O provides great parallelization potential for complete graphs wherein every vertex has a connection to all other vertices in the graph. Current MASS Parallel I/O is not well suited for biological networks that exhibit network property of fewer vertices with high degree and more vertices with low degree. Hence, MASS implementation preferred sequential graph parser described in 4.1 over parallel I/O.

Table 5.8. Input graph size for different formats

Dataset	Edge List File Size	Parallel I/O File Size
Dolphin	1 KB	8 KB
Power	62 KB	923 KB
DIP	761 KB	75,370 KB

1	2	3	4
0	2	5	6
0	1	7	-1
0	-1	-1	-1
0	-1	-1	-1
1	-1	-1	-1
1	-1	-1	-1
2	-1	-1	-1

Figure 5.4. Parallel I/O Graph

5.5.2 MASS Agent Population Control Feature

This feature controls the active agent population by serializing agents that are spawned beyond the maximum limit and deserialize agents once the current population drops below the maximum limit. Although this feature caches inactive agent data in a serialized form (that are much smaller than raw agent objects), serialized agent data consumes significant heap space when inactive agents grow exponentially in order of millions. Due to highly imbalanced agents in computing nodes, even if a single computing node (whose heap is filled by active agents and serialized inactive agents) triggers full garbage collection, then execution takes unreasonable time. To reduce heap

utilization and avoid garbage collection pauses, inactive agents are stored to disk in GZIP compressed format to minimize disk space usage as seen in Listing 4. Though this implementation worked for small graphs, it couldn't be tested in the current Network File System (NFS) cluster environment for larger graphs. This is because when MASS execution creates millions of files (for serialized inactive agents) parallelly from all computing nodes, remote connection to the test cluster gets aborted. Although this application couldn't benefit from this feature, it could be useful for applications in which agent expansion happens linearly between successive iterations.

```

1 public byte[] serializeAgent(Agent agent) {                public String serializeAgent(Agent agent) {
2   ByteArrayOutputStream baos =                            // Use UUID as filename to store serialized agents in disk
3       new ByteArrayOutputStream();                       // Compress serialized agent in .gz format to save disk space
4   ObjectOutputStream oos =                                String serializedAgent = UUID.randomUUID().toString() + ".gz";
5       new ObjectOutputStream(baos);                     FileOutputStream fos = new FileOutputStream(serializedAgent);
6   oos.writeObject(agent);                                GZIPOutputStream gz = new GZIPOutputStream(fos);
7   oos.flush();                                          ObjectOutputStream oos = new ObjectOutputStream(gz);
8   oos.close();                                          oos.writeObject(agent);
9   byte[] serializedAgent = baos.toByteArray();          oos.close();
10  return serializedAgent;                                return serializedAgent;
11 }                                                       }
12
13 public Agent deserializeAgent(byte[] serializedAgent) {  public Agent deserializeAgent(String serializedAgent) {
14   ByteArrayInputStream bais =                            FileInputStream fin = new FileInputStream(serializedAgent);
15       new ByteArrayInputStream(serializedAgent);        GZIPInputStream gis = new GZIPInputStream(fin);
16   ObjectInputStream ois = new ObjectInputStream(bais);   ObjectInputStream ois = new ObjectInputStream(gis);
17   Agent deserializedAgent = (Agent) ois.readObject();   Agent deserializedAgent = (Agent) ois.readObject();
18   ois.close();                                          ois.close(); // close stream and delete agent data file
19   return deserializedAgent;                             deleteFile(serializedAgent);
20 }                                                       return deserializedAgent;
21 }                                                       }

```

Listing 4. Agent serialize and deserialize code snippets (Left: Existing code that maintains inactive agents in heap, Right: Modified code to store inactive agents in disk)

5.5.3 MASS Asynchronous Agent Migration Feature

```

1 // Synchronous Agent Migration                1 // Asynchronous Agent Migration
2 crawler.callAll(Crawler.enumerateExhaustive_); 2 crawler.doAll(new int[] {Crawler.enumerateExhaustive_}, null, motifSize);
3 crawler.manageAll();

```

Listing 5. Synchronous vs Asynchronous agent migration code snippet

MASS asynchronous agent migration feature reduces communication and synchronization overhead across computing nodes by repeatedly performing agent function execution and migration for specified iterations without returning control to the driver program. To evaluate this feature, the execution time for synchronous and asynchronous agent migrations shown in Listing 5, are measured for Dolphin graph in which large motif size (9) can be detected. Table 5.9 shows

performance gain slowly increases with the number of iterations (motif size). Consequently, graph algorithms that execute for a higher number of iterations will benefit from this feature.

Table 5.9. MASS callAll vs doAll Performance

Motif Size	#Subgraphs	Execution Time (sec)		doAll Speedup
		callAll	doAll	
6	107,775	17.623	17.851	0.987
7	550,428	18.246	18.111	1.007
8	2,683,740	39.372	35.548	1.107
9	12,495,833	150.832	129.185	1.167

5.6 MASS VERSUS SPARK ANALYSIS

This section analyzes MASS and Spark parallel implementations and evaluates the fitness of MASS for similar graph problems that handle large-scale data.

5.6.1 Parallelism Analysis

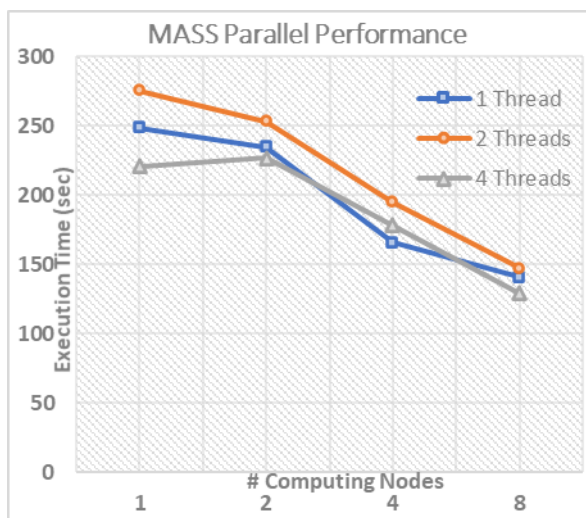


Figure 5.5. MASS Parallel Performance

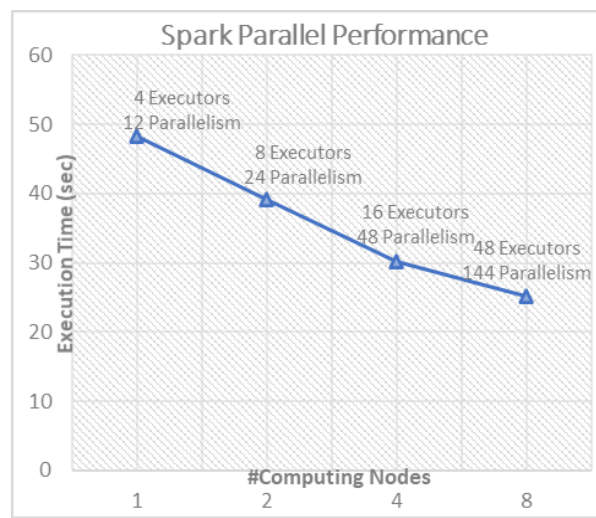


Figure 5.6. Spark Parallel Performance

To assess MASS and Spark parallelism, New MASS version tested with different threads and computing nodes, and Spark version tested with different executors and parallelism configurations for motif size 9 in Dolphin graph. As evident from Figure 5.5, MASS performance improved with an increase in the number of computing nodes and maximum threads (4) utilized. 2-threads performed slower than single thread execution because threads synchronization time nullified the parallel performance gained by 2-threads. But 4-threads gained more parallel performance than 2-threads such that threads synchronization time had no negative impact on performance.

Analogous to MASS, Figure 5.6 depicts Spark also exhibits similar parallel performance gain with an increase in the number of computing nodes, executors, and default parallelism value. MASS achieved 1.925x speedup, and Spark gained 1.924x speedup from the minimum to maximum parallel configuration. This signifies both MASS and Spark parallel implementations can execute faster in a sophisticated cluster environment with enormous computing power and large physical memory capacity.

5.6.2 Programmability Analysis

This section presents quantitative and qualitative programmability analysis of the parallel versions. The quantitative analysis focuses on the boilerplate code ratio, lines of code (LoC), and the number of parallel methods or operations. The quantitative analysis targets development efforts, data representation, flexible communication, ease of parallelism, and performance improvements.

Quantitative Analysis. Boilerplate code represents the lines to code that are intended to configure the parallelization framework and are not related to application implementation. Boilerplate code ratio measures the percentage of boilerplate code to the total number of lines of code. The smaller boilerplate code ratio symbolizes easiness to set up the parallel environment. Lines of code measurement presented in this section omitted comment and blank lines and both parallel versions followed standard java coding convention (80 characters code width). Table 5.10 shows both MASS and Spark implementations have a significantly lesser boilerplate code ratio. But, the boilerplate code ratio for Spark is slightly higher than the MASS. This is because the Spark version has to register all user-defined classes to utilize Kryo serializer, as discussed in section 4.3.1.

Table 5.10. Lines of Code (LoC) and Boilerplate Code Ratio

Modules	MASS LoC	Spark LoC
Graph Parser	170	
Labeler & Statistical Test	302	
Boilerplate code	12	16
Framework specific code	661	463
Total LoC	1145	951
Boilerplate Code Ratio	1.04%	1.68%

Table 5.10 indicates MASS has a higher total LoC than Spark due to the difference in parallel paradigms. MASS version includes 3 classes (TargetNwArgs2Places, RandomNwArgs2Places, Args2Agents) with parameterized constructor alone that serve the purpose of passing arguments to specific agent or place during the invocation of parallel methods. Such framework-specific implementation increases total LoC, but enhancing MASS features will relieve the application developers from the burden of implementing common functionalities. For instance, the graph parser module can be eliminated for the MASS version if the enhanced MASS Parallel I/O feature eliminates constraints described in 5.5.1. This elimination makes MASS total LoC (975) roughly equivalent to Spark total LoC (951). Thus, excluding code that performs common tasks such as graph parsing reduces the total LoC, and simplifies application parallelization using MASS.

Table 5.11. Parallel Methods or Operations Count

MASS Parallel Methods		Spark Parallel Operations	
MASS Place	3	Transformations	8
MASS Agent	2	Actions	1
Total	5	Total	9

Table 5.11 presents programmability analysis based on the number of parallel methods used in MASS and the number of parallel operations used in Spark version. As seen in Table 5.11, Spark implements more parallel operations than the MASS version. MASS version has three parallel methods in place to initialize (target and random) network and gather (collect subgraphs) results, and two methods in agent for algorithm implementation (exhaustive and random enumeration). But Spark version applies eight transformations (one to initialize network and seven to implement algorithm) and one final action to gather results. This reveals the MASS version closely resembles sequential java implementation and developers can easily parallelize applications using the MASS library by identifying methods that can be operated in parallel from places and agents.

Qualitative Analysis. Unlike Spark’s RDD flat representation, MASS provides spatial graph representation to easily discover graph structure and intuitively parallelize sequential graph algorithms. Developing a parallel application using MASS is less complex and requires fewer development efforts for biologists, while Spark implementation involves longer development time to change sequential algorithms to fit into Spark's data-parallel model of transformations and actions. Factors such as choosing efficient transformation/action (reduceByKey instead of

groupByKey), and deciding on RDD partitioning and persistence play an important role to guarantee optimal utilization of the cluster resources and improve performance.

MASS offers a simple API to alter parallelism whereas Spark requires deep dive into default configurations to fine-tune parallelism described in 4.3.1. Consequently, developers from non-parallel backgrounds spent more time and effort to achieve maximum parallelism with Spark. Additionally, MASS allows flexible communication between graph vertices, while Spark GraphX restricts direct communication between non-adjacent vertices. Hence, graph algorithms that require direct communication between non-adjacent vertices benefit from MASS’s flexibility.

5.6.3 In-Memory Data Reuse Analysis

To assess in-memory data analysis, MASS and Spark versions are tested for a 50% sampled subgraph enumeration in 1000 random graphs. As discussed in section 4.3, Spark streams data or recreates vertexRDD for each random graph, while MASS updates neighbor information in Place structure instead of deleting and creating vertex structure for every graph. Table 5.10 shows the benefit of retaining data structure in memory (MASS places) for multiple iterations so that different operations can be performed over the same data structure (holding the same or modified data for each iteration). Though Spark executed 3.2-3.5 times faster than MASS for single target graph enumeration, MASS performed 1.6 times faster than Spark for 1000 random graphs enumeration. In conclusion, iterative graph algorithms that perform different operations on the same or modified graph structure can attain higher speedup using MASS.

Table 5.12. MASS Speedup for In-Memory Data Reuse

Target Graph	Undirected Power (Motif Size = 4)		Directed Yeast (Motif Size = 3)	
	MASS	Spark	MASS	Spark
Target Subgraphs	63,401	63,401	13,150	13,150
Random Subgraphs	39,983,983	39,962,071	5,389,395	5,391,802
Total Subgraphs	40,047,384	40,025,472	5,402,545	5,404,952
Target Enumeration	29.288	8.912	22.273	6.221
Random Enumeration	705.126	1191.824	266.653	427.241
Total Time (sec)	734.543	1202.804	288.997	435.467
MASS Speedup	1.637		1.506	

Chapter 6. CONCLUSION & FUTURE WORK

Agent-based and Spark network motif implementations detect high-order motifs (> 8) that are infeasible with sequential tools Mfinder, FANMOD. This work verified the correctness of parallel results by comparing it with sequential results from NemoLib, and FANMOD. It also evaluated performance for different sparse real networks and dense synthetic graphs. Unlike old MASS implementation, New MASS version detects significant motifs and also supports directed graph analysis as well as the sampled version of subgraph enumeration to improve the speed.

Compared with the Old MASS version, New MASS version achieved 2x speedup on higher-order motifs in real networks, and on average, 2x memory reduction in dense networks. New MASS version optimized data structures by eliminating nested objects and utilized primitive type-specific collections. These optimized structures consumed less heap space, which in turn reduced garbage collection time and improved performance. Light-weight agents reduced data serialization cost during agent migration and object creation overhead during agent duplication.

This work proved that developing a memory-optimized application enhances MASS scalability to deal with millions of agents with a moderate performance impact. But increasing more than tens of millions of agents cause memory exhaustion, which needs to be addressed by fine-tuning data structures within the library or by modifying the memory model to use off-heap memory or disk space, that are not managed by the garbage collector.

Parallel Spark version achieved at most 2x speedup than sequential NemoLib for high-order motif enumeration in target graphs. But MASS gained 1.5 times speedup than Spark for low-order significant motif detection. This project demonstrates graph algorithms that require flexible communication between vertices irrespective of connectivity, and those that perform different computations on the same data will benefit from MASS.

As for future work, MASS network motif application can be extended to support NemoProfile and NemoCollect functionalities, and also visualize detected significant motifs in Cytoscape using the recently developed MASS Cytoscape plugin. Due to memory overhead, current implementation enumerates random graphs sequentially. In future, random graph enumeration can be parallelized along with target graph enumeration to significantly improve the performance.

The source code for MASS and Spark implementations are available in MASS Java applications repository (https://bitbucket.org/mass_application_developers/mass_java_appl).

BIBLIOGRAPHY

- [1] Milo, Ron, Shai Shen-Orr, Shalev Itzkovitz, Nadav Kashtan, Dmitri Chklovskii, and Uri Alon. Network motifs: simple building blocks of complex networks. *Science* 298, no. 5594, pp. 824-827, 2002.
- [2] Junker, Björn H., and Falk Schreiber. *Analysis of biological networks*. Vol. 2. John Wiley & Sons, 2011.
- [3] Fukuda, Munehiro. MASS: Parallel-computing library for multi-agent spatial simulation. *Distributed Systems Laboratory, Computing & Software Systems, University of Washington Bothell, Bothell, WA* (2010).
- [4] Apache Spark, <https://spark.apache.org/docs/latest/index.html>
- [5] Mfinder, <https://www.weizmann.ac.il/mcb/UriAlon/download/network-motif-software>
- [6] Wernicke, Sebastian, and Florian Rasche. FANMOD: a tool for fast network motif detection. *Bioinformatics* 22, no. 9 (2006): 1152-1153.
- [7] Andersen, Andrew, and Wooyoung Kim. NemoLib: A Java Library for Efficient Network Motif Detection. In *International Symposium on Bioinformatics Research and Applications*, pp. 403-407. Springer, Cham, 2017.
- [8] Matthew Sell, Munehiro Fukuda, Agent Programmability Enhancement for Rambling over a Scientific Dataset, to appear in *PAAMS 2020*, October 7-9, L'Aquila, Italy
- [9] Zaharia, Matei, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th (USENIX) Symposium on Networked Systems Design and Implementation (NSDI 12)* pp. 15-28. 2012.
- [10] The Internals of Apache Spark, <https://books.japila.pl/apache-spark-internals/apache-spark-internals/2.4.4/spark-architecture.html>
- [11] Erciyes, Kayhan. *Distributed and sequential algorithms for bioinformatics*. Vol. 23. Cham: Springer, 2015.
- [12] Wernicke, Sebastian. Efficient detection of network motifs. *IEEE/ACM transactions on Computational Biology and Bioinformatics*, vol. 3, no. 4, pp. 347-359, 2006.

- [13] Kashtan, Nadav, Shalev Itzkovitz, Ron Milo, and Uri Alon. Efficient sampling algorithm for estimating subgraph concentrations and detecting network motifs. *Bioinformatics* 20, no. 11, pp. 1746-1758, 2004.
- [14] Wang T, Touchman JW, Zhang W, Suh EB, Xue G (2005) A parallel algorithm for extracting transcription regulatory network motifs. In *Proceedings of the IEEE international symposium on bioinformatics and bioengineering*, IEEE Computer Society Press, LosAlamitos, CA,USA, pp 193–200.
- [15] Ribeiro, Pedro, and Fernando Silva. G-tries: an efficient data structure for discovering network motifs. In *Proceedings of the 2010 ACM symposium on applied computing*, pp. 1559-1566. 2010.
- [16] Ribeiro, Pedro, Fernando Silva, and Luis Lopes. Efficient parallel subgraph counting using g-tries. In *2010 IEEE International Conference on Cluster Computing*, pp. 217-226. IEEE, 2010.
- [17] Verma, Vartika, Paul Park Kwon, Anand Joglekar, and Wooyoung Kim. Network motif analysis in clouds-subgraph enumeration with iterative hadoop mapreduce. *vol. 4*,pp. 28-4, 10 2016.
- [18] Ribeiro, Pedro, Fernando Silva, and Luís Lopes. Parallel discovery of network motifs. *Journal of Parallel and Distributed Computing* 72, no. 2, pp. 144-154, 2012.
- [19] Matthew Kipps, Wooyoung Kim, and Munehiro Fukuda. Agent and Spatial Based Parallelization of Biological Network Motif Search. In *Proc. 17th IEEE International Conference on High Performance Computing and Communications - HPCC 2015*, pages 786–791, New York, NY, August 2015.
- [20] Andrew Andersen, Wooyoung Kim, and Munehiro Fukuda. Mass-based nemoprofile construction for an efficient network motif search. In *IEEE International Conference on Big Data and Cloud Computing in Bioinformatics - BDCloud 2016*, pp. 601–606, Atlanta, GA, October 2016.
- [21] McKay, Brendan D., and Adolfo Piperno. Practical graph isomorphism, II. *Journal of Symbolic Computation* 60, pp. 94-112, 2014.
- [22] Kim, Wooyoung, and Lynnette Haukap. NemoProfile as an efficient approach to network motif analysis with instance collection. *BMC bioinformatics* 18, no. 12, p.423, 2017.
- [23] Nauty labelg Man Page, <https://www.mankier.com/1/nauty-labelg>
- [24] Newman, M. E. (2003). The structure and function of complex networks. *SIAM review*, 45(2), 167-256.
- [25] Fastutil: Fast & compact type-specific collections for Java, <http://fastutil.di.unimi.it/>

- [26] Costa, Diego, Artur Andrzejak, Janos Seboek, and David Lo. Empirical study of usage and performance of java collections. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, pp. 389-400. 2017.
- [27] <http://java-performance.info/hashmap-overview-jdk-fastutil-goldman-sachs-hppc-koloboke-trove-january-2015/>
- [28] Tuning - Spark 2.4.5 Documentation, <https://spark.apache.org/docs/latest/tuning.html>
- [29] Gonzalez, Joseph E., Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pp. 599-613. 2014.
- [30] Malewicz, Grzegorz, Matthew H. Austern, Aart JC Bik, James C. Dehnert, Ian 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*, pp. 135-146. 2010.
- [31] Ryza, Sandy, Uri Laserson, Sean Owen, and Josh Wills. Advanced analytics with spark: patterns for learning from data at scale, Second Edition, O'Reilly Media, Inc., 2017.
- [32] RDD Programming Guide, <https://spark.apache.org/docs/latest/rdd-programming-guide.html>
- [33] MASS Java Core, https://bitbucket.org/mass_library_developers/mass_java_core/src/master/
- [34] NemoLib Java version 2, <https://github.com/Kimw6/NemoLib-Java-V2>
- [35] Downloads - Apache Spark, <https://spark.apache.org/downloads.html>
- [36] Nauty Traces - Home, <http://pallini.di.uniroma1.it/>
- [37] Maven Repository: Fastutil, <https://mvnrepository.com/artifact/it.unimi.dsi/fastutil/8.3.1>
- [38] Hagberg, Aric, Pieter Swart, and Daniel S Chult. Exploring Network Structure, Dynamics, and Function using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy2008)*, Gael Varoquaux, Travis Vaught, and Jarrod Millman (Eds.), (Pasadena, CA USA), pp. 11–15.
- [39] Lusseau, David, Karsten Schneider, Oliver J. Boisseau, Patti Haase, Elisabeth Slooten, and Steve M. Dawson. The bottlenose dolphin community of Doubtful Sound features a large proportion of long-lasting associations. *Behavioral Ecology and Sociobiology*, 54, no. 4 (2003), pp. 396-405.
- [40] Watts, Duncan J., and Steven H. Strogatz. Collective dynamics of ‘small-world’ networks. *Nature* 393, no. 6684 (1998), pp. 440-442.

- [41] Xenarios, Ioannis, Lukasz Salwinski, Xiaoqun Joyce Duan, Patrick Higney, Sul-Min Kim, and David Eisenberg. DIP, the Database of Interacting Proteins: a research tool for studying cellular networks of protein interactions. *Nucleic Acids Research*, 30, no. 1 (2002), pp. 303-305.
- [42] Gene regulation network (Transcription interaction of yeast *S. cerevisiae*), <https://www.weizmann.ac.il/mcb/UriAlon/download/collection-complex-networks>
- [43] Ripeanu, Matei, Ian Foster, and Adriana Iamnitchi. Mapping the Gnutella Network: Properties of Large-Scale Peer-to-Peer Systems and Implications for System Design. *IEEE Internet Computing Journal*, (2002).
- [44] Batagelj, Vladimir, and Ulrik Brandes. Efficient generation of large random networks. *Physical Review E* 71, no. 3, 2005.
- [45] Runtime (Java SE 8), <https://docs.oracle.com/javase/8/docs/api/java/lang/Runtime.html>

APPENDIX A

Graph6 and Digraph6 formats used by NautyTraces utility. NautyTraces tool is commonly used to test graph isomorphism with canonical labels. In this work, it is used to group isomorphic subgraphs. This tool requires an input graph to be in a specific compact encoded format described here. Undirected and directed graphs have to be encoded in ‘graph6’ and ‘digraph6’ formats respectively to use labelg executable of the NautyTraces utility.

graph6 format: N(n) R(x)

digraph6 format: & N(n) R(x)

where N(n), R(x) denotes bit vector representation of graph size ‘n’ and graph structure, and ‘&’ is used to distinguish directed graphs.

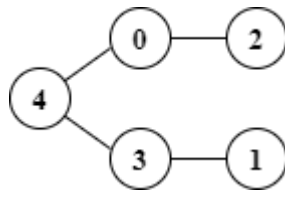
Bit vector representation. In this representation, all bytes have a value in the range 63-126 that are printable ASCII characters (A-Z, a-z, ?, @, [, \,], ^, _, ` , {, |, }, ~). The below steps explain the procedure to create a bit vector representation for value ‘35612’.

Step 1: Convert value to binary	100010 110001 1100
Step 2: Pad ‘0’ to right to make digits exact multiple of ‘6’	100010 110001 110000
Step 3: Add 63 to each group of 6 bits	(34+63) (49+63) (48+63)
Step 4: Store each group of 6 bits value in a byte	97 112 111

Bit vector representation N(n) for graph size ‘n’:

‘n’ range	N(n) size	N(n) bit vector representation
$0 \leq n \leq 62$	1 byte	n+63
$63 \leq n \leq 258047$	4 bytes	126 R(n), R(n): 18-bit bitvector representation of n
$258048 \leq n \leq 68719476735$	8 bytes	126 126 R(n), R(n): 36-bit bitvector representation of n

Bit vector representation R(x) for graph structure. Graph6 format encodes the upper triangle of the adjacency matrix of an undirected graph as a bit vector x of length ‘n(n-1)/2’, while digraph6 encodes the adjacency matrix of a directed graph as a bit vector x of length ‘n²’ row by row.



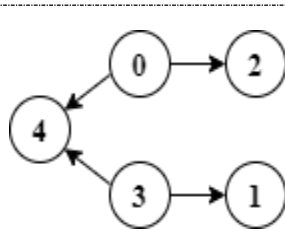
	0	1	2	3	4
0	0	0	1	0	1
1	0	0	0	1	0
2	1	0	0	0	0
3	0	1	0	0	1
4	1	0	0	1	0

n = 5, N(n) = 5+63 = 68

R(x) = 010010 1001 => 010010 1001**00** (Padding)

= (18+63) (36+63) => 81 99

Graph6: N(n) R(x): 68 81 99 => **‘DQe’**



	0	1	2	3	4
0	0	0	1	0	1
1	0	0	0	0	0
2	0	0	0	0	0
3	0	1	0	0	1
4	0	0	0	0	0

n = 5, N(n) = 5+63 = 68

R(x) = 001010 000000 000010 010000 000000

= (10+63) (0+63) (2+63) (16+63) (0+63)

Digraph6: & N(n) R(x): 38 68 73 63 65 79 63

=> **‘&DI?AO?’**

APPENDIX B

Appendix B presents the execution results of the sequential tool (NemoLib) and implemented parallel versions (MASS and Spark). The correctness of the result can be verified by comparing the target graph enumeration results of all implementations. All executions used 1000 random graphs and 50% sampling (1,1,...1,0.5) for randomization. The following list of values associated with random graphs are expected to be different in each execution.

- Number of subgraphs enumerated in random graphs,
- Random Mean Frequency,
- Random Standard Deviation,
- Z-score, and
- P-value

Directed Yeast Graph Execution Results.

```
[saranyad@hermes01 sequentialApp]$ ./run.sh /home/NETID/saranyad/InputGraph/YeastDirected.txt 3 1000 1
Filename = /home/NETID/saranyad/InputGraph/YeastDirected.txt
Parsing target graph...
Analyzing target graph...
13150 subgraphs enumerated in the target network.
Generating 1000 random graph...
530790 subgraphs enumerated in the random networks.
Label RelFreq RandMeanFreq Z-Score P-Value
6B7o 0.000% 0.005% -0.426 1.000
6B9o 0.000% 0.034% -1.089 1.000
6BC 23.795% 21.952% 2.756 0.004
6B5o 0.000% 0.031% -0.970 1.000
6B8o 0.000% 0.021% -0.768 1.000
6B0o 0.000% 0.040% -1.065 1.000
6Bp 0.000% 0.074% -1.848 1.000
6B8o 0.000% 1.627% -2.255 1.000
6B0o 31.650% 30.417% 1.209 0.117
6Bco 0.548% 1.007% -2.681 0.996
6B5 0.000% 0.918% -2.232 1.000
6B0 0.000% 0.000% -0.071 1.000
6B7o 44.008% 43.874% 0.176 0.420
SubgraphCount Complete
Overall execution time = 23647 milliseconds
Memory Used: 576634496 Bytes (549 MB)
```

NemoLib Execution Output

```
[saranyad@hermes01 SparkNetworkMotif]$ ./run.sh /home/NETID/saranyad/InputGraph/YeastDirected.txt 3 1000 1 0
Note: Subgraph.java uses unchecked or unsafe operations.
Note: Recompile with -XInt:unchecked for details.
added manifest
adding: CanonicalLabeler.class (in = 4845) (out= 2592) (deflated 46%)
adding: Graph.class (in = 4312) (out= 2265) (deflated 47%)
adding: Main.class (in = 4925) (out= 2578) (deflated 47%)
adding: Motif.class (in = 1963) (out= 1029) (deflated 47%)
adding: MotifDetector.class (in = 12750) (out= 5809) (deflated 54%)
adding: Subgraph.class (in = 6927) (out= 3626) (deflated 47%)
adding: Vertex.class (in = 1444) (out= 696) (deflated 51%)
-----
Target Graph Enumeration
-----
13150 subgraphs from 4 classes enumerated in target network
Target enumeration took 6221 milliseconds
-----
Random Graphs Enumeration
-----
Random enumeration took 427241 milliseconds
13150 subgraphs were enumerated in the target network.
5301802 subgraphs were enumerated in the random networks.
5404952 subgraphs were enumerated in all networks.
Overall execution time = 435467 milliseconds
Memory Used: 990002120 Bytes (944 MB)
-----
Spark Network Motif Detection
-----
Motif |Target Freq |Random Mean Freq|Random Std Dev |Z-Score |P-Value
-----|-----|-----|-----|-----|-----
6B7o | 23.7947 % | 22.5678 % | 0.00705551 | 1.739 | 0.038
6B0o | 0.5475 % | 1.0501 % | 0.00182733 | -2.750 | 1.000
6BC | 23.7947 % | 22.6113 % | 0.00729423 | 1.622 | 0.058
6B7o | 44.0076 % | 45.1480 % | 0.00801200 | -1.423 | 0.928
-----
```

Spark Execution Output

Another difference is that NemoLib displays non-candidate motifs (motifs that aren't present in target graph but found in random graphs) that are insignificant in addition to candidates (motifs that are present in target graph), whereas parallel implementations display only candidate motifs.

```
[saranyad@hermes01 NetworkMotif]$ ./run.sh /home/NETID/saranyad/InputGraph/YeastDirected.txt 3 1000 1 0
MPProcess on hermes02.uwb.edu run with command: /usr/bin/java -Xms4g -Xmx12g -cp /home/NETID/saranyad/AgentCompression/mass_java_app/Applications/NetworkMotif/target
1 8 1 58050 /home/NETID/saranyad/AgentCompression/mass_java_app/Applications/NetworkMotif/target
MPProcess on hermes03.uwb.edu run with command: /usr/bin/java -Xms4g -Xmx12g -cp /home/NETID/saranyad/AgentCompression/mass_java_app/Applications/NetworkMotif/target
2 8 1 58050 /home/NETID/saranyad/AgentCompression/mass_java_app/Applications/NetworkMotif/target
MPProcess on hermes04.uwb.edu run with command: /usr/bin/java -Xms4g -Xmx12g -cp /home/NETID/saranyad/AgentCompression/mass_java_app/Applications/NetworkMotif/target
3 8 1 58050 /home/NETID/saranyad/AgentCompression/mass_java_app/Applications/NetworkMotif/target
MPProcess on hermes05.uwb.edu run with command: /usr/bin/java -Xms4g -Xmx12g -cp /home/NETID/saranyad/AgentCompression/mass_java_app/Applications/NetworkMotif/target
4 8 1 58050 /home/NETID/saranyad/AgentCompression/mass_java_app/Applications/NetworkMotif/target
MPProcess on hermes06.uwb.edu run with command: /usr/bin/java -Xms4g -Xmx12g -cp /home/NETID/saranyad/AgentCompression/mass_java_app/Applications/NetworkMotif/target
5 8 1 58050 /home/NETID/saranyad/AgentCompression/mass_java_app/Applications/NetworkMotif/target
MPProcess on hermes07.uwb.edu run with command: /usr/bin/java -Xms4g -Xmx12g -cp /home/NETID/saranyad/AgentCompression/mass_java_app/Applications/NetworkMotif/target
6 8 1 58050 /home/NETID/saranyad/AgentCompression/mass_java_app/Applications/NetworkMotif/target
MPProcess on hermes08.uwb.edu run with command: /usr/bin/java -Xms4g -Xmx12g -cp /home/NETID/saranyad/AgentCompression/mass_java_app/Applications/NetworkMotif/target
7 8 1 58050 /home/NETID/saranyad/AgentCompression/mass_java_app/Applications/NetworkMotif/target
MASS.init: done
40 milliseconds to parse and construct target network
21479 milliseconds to initialize target network data
27 milliseconds to instantiate agents at all graph vertices
-----
Target Graph Enumeration
-----
#Agents: 688
#Agents: 12978
649 milliseconds to enumerate all subgraphs in target network
77 milliseconds to collect all subgraphs from all places
13150 subgraphs from 4 non-isomorphic classes enumerated in target network
Target enumeration took 22273 milliseconds
-----
Random Graphs Enumeration
-----
Random enumeration took 266653 milliseconds
13150 subgraphs were enumerated in the target network.
5389395 subgraphs were enumerated in the random networks.
5402545 subgraphs were enumerated in all networks.
Overall execution time = 288997 milliseconds
Memory Used: 4383824264 Bytes (4180 MB)
-----
MASS Network Motif Detection
-----
Motif |Target Freq |Random Mean Freq|Random Std Dev |Z-Score |P-Value
-----|-----|-----|-----|-----|-----
6B7o | 23.7947 % | 22.5678 % | 0.00705551 | 1.739 | 0.038
6B0o | 0.5475 % | 1.0501 % | 0.00182733 | -2.750 | 1.000
6BC | 23.7947 % | 22.6113 % | 0.00729423 | 1.622 | 0.058
6B7o | 44.0076 % | 45.1480 % | 0.00801200 | -1.423 | 0.928
-----
```

MASS Execution Output

Undirected Dolphin Network Execution Results.

```
[sarayadhermes01 sequentialApp]$ ./run.sh /home/NETID/sarayad/InputGraph/Dolphin.txt 5 1000 0
filename = /home/NETID/sarayad/InputGraph/Dolphin.txt
Parsing target graph...
Analyzing target graph...
20346 subgraphs enumerated in the target network.
Generating 1000 random graph...
13751742 subgraphs enumerated in the random networks.
Label RelFreq RandMeanFreq Z-Score P-Value
D*( 0.128% 0.000% 139.443 0.000
D-( 0.015% 0.000% 0.000 0.000
D? 2.497% 3.602% -2.725 1.000
Dk 3.273% 0.507% 13.135 0.000
Dr 0.039% 0.002% 10.567 0.000
DR 0.634% 0.055% 14.362 0.000
Dr 0.143% 0.028% 6.275 0.000
DM 0.319% 0.083% 41.831 0.000
DJ 1.126% 0.021% 29.312 0.000
DFw 0.029% 0.078% -1.320 0.932
DDW 30.419% 37.804% -6.762 1.000
Des 28.610% 40.594% -8.312 1.000
DBw 3.155% 4.199% -1.832 0.970
DFc 0.236% 0.009% 21.003 0.000
DD 9.230% 4.632% 5.580 0.000
Dd 1.096% 0.470% 5.837 0.000
DBc 2.924% 0.502% 12.625 0.000
D 10.607% 4.293% 8.290 0.000
Dq 4.201% 2.297% 5.320 0.000
D 0.541% 0.126% 7.607 0.000
DqK 0.698% 0.784% -0.883 0.805

SubgraphCount Complete
Overall execution time = 34903 milliseconds
Memory Used: 44704896 Bytes (42 MB)
```

NemoLib Execution Output

```
[sarayadhermes01 SparkNetworkMotif]$ ./run.sh /home/NETID/sarayad/InputGraph/Dolphin.txt 5 1000 0
Notes: Subgraph.java uses unchecked or unsafe operations.
Notes: Recompile with -Xlint:unchecked for details.
added manifest
adding: CanonicalLabeler.class(in = 4845) (out= 2592)(deflated 46%)
adding: Graph.class(in = 4312) (out= 2265)(deflated 47%)
adding: Main.class(in = 4925) (out= 2578)(deflated 47%)
adding: Motif.class(in = 1963) (out= 1029)(deflated 47%)
adding: MotifDetector.class(in = 12750) (out= 5809)(deflated 54%)
adding: Subgraph.class(in = 6927) (out= 3026)(deflated 47%)
adding: Vertex.class(in = 1444) (out= 696)(deflated 51%)

-----
Target Graph Enumeration
-----
20346 subgraphs from 21 classes enumerated in target network
Target enumeration took 6396 milliseconds

-----
Random Graphs Enumeration
-----
Random enumeration took 354326 milliseconds
20346 subgraphs were enumerated in the target network.
13866085 subgraphs were enumerated in the random networks.
13886431 subgraphs were enumerated in all networks.
Overall execution time = 362729 milliseconds
Memory Used: 1361543128 Bytes (1298 MB)
```

Spark Network Motif Detection

Motif	Target Freq	Random Mean Freq	Random Std Dev	Z-Score	P-Value
D*(0.1278 %	0.0001 %	0.00001281	99.646	0.000
D-(0.0147 %	0.0000 %	0.00000000	Infinity	0.000
D?	2.4968 %	3.6470 %	0.00406634	-2.828	1.000
Dk	3.2734 %	0.4958 %	0.00199243	13.941	0.000
Dr	0.0340 %	0.0020 %	0.00039280	14.799	0.000
DR	0.1425 %	0.0265 %	0.00017907	6.478	0.000
Dr	0.0393 %	0.0017 %	0.00004074	9.225	0.000
DM	0.3195 %	0.0029 %	0.00007052	44.895	0.000
DJ	1.1255 %	0.0185 %	0.00034988	31.640	0.000
DFw	0.0295 %	0.0773 %	0.00036180	-1.321	0.944
DDW	30.4188 %	37.7987 %	0.01080771	-6.828	1.000
Des	28.6100 %	40.7867 %	0.01477791	-8.240	1.000
DBw	3.1554 %	4.2275 %	0.00571387	-1.876	0.976
DFc	0.2359 %	0.0088 %	0.00010540	21.551	0.000
DD	9.2303 %	4.5280 %	0.00842683	5.580	0.000
Dd	1.0960 %	0.4640 %	0.00107574	5.875	0.000
DBc	2.9244 %	0.4953 %	0.00197902	12.274	0.000
D	10.6065 %	4.2037 %	0.00756681	8.462	0.000
Dq	4.2009 %	2.2577 %	0.00379662	5.329	0.000
D	0.5406 %	0.1144 %	0.00052301	8.150	0.000
DqK	0.6979 %	0.7926 %	0.00099307	-0.953	0.827

Spark Execution Output

```
[sarayadhermes01 NetworkMotif]$ ./run.sh /home/NETID/sarayad/InputGraph/Dolphin.txt 5 1000 0
MProcess on hermes02.uwb.edu run with command: /usr/bin/java -Xms4g -Xmx12g -cp /home/NETID/sarayad/
1 8 1 58050 /home/NETID/sarayad/AgentCompression/mass_java_app/Agents/NetworkMotif/target
MProcess on hermes03.uwb.edu run with command: /usr/bin/java -Xms4g -Xmx12g -cp /home/NETID/sarayad/
2 8 1 58050 /home/NETID/sarayad/AgentCompression/mass_java_app/Agents/NetworkMotif/target
MProcess on hermes04.uwb.edu run with command: /usr/bin/java -Xms4g -Xmx12g -cp /home/NETID/sarayad/
3 8 1 58050 /home/NETID/sarayad/AgentCompression/mass_java_app/Agents/NetworkMotif/target
MProcess on hermes05.uwb.edu run with command: /usr/bin/java -Xms4g -Xmx12g -cp /home/NETID/sarayad/
4 8 1 58050 /home/NETID/sarayad/AgentCompression/mass_java_app/Agents/NetworkMotif/target
MProcess on hermes06.uwb.edu run with command: /usr/bin/java -Xms4g -Xmx12g -cp /home/NETID/sarayad/
5 8 1 58050 /home/NETID/sarayad/AgentCompression/mass_java_app/Agents/NetworkMotif/target
MProcess on hermes07.uwb.edu run with command: /usr/bin/java -Xms4g -Xmx12g -cp /home/NETID/sarayad/
6 8 1 58050 /home/NETID/sarayad/AgentCompression/mass_java_app/Agents/NetworkMotif/target
MProcess on hermes08.uwb.edu run with command: /usr/bin/java -Xms4g -Xmx12g -cp /home/NETID/sarayad/
7 8 1 58050 /home/NETID/sarayad/AgentCompression/mass_java_app/Agents/NetworkMotif/target
MASS.init: done
13 milliseconds to parse and construct target network
15403 milliseconds to initialize target network data
25 milliseconds to instantiate agents at all graph vertices
```

```
-----
Target Graph Enumeration
-----
#Agents: 62
#Agents: 5153
1020 milliseconds to enumerate all subgraphs in target network
73 milliseconds to collect all subgraphs from all places
20346 subgraphs from 21 non-isomorphic classes enumerated in target network
Target enumeration took 16561 milliseconds

-----
Random Graphs Enumeration
-----
Random enumeration took 454342 milliseconds
20346 subgraphs were enumerated in the target network.
13783252 subgraphs were enumerated in the random networks.
13803598 subgraphs were enumerated in all networks.
Overall execution time = 470947 milliseconds
Memory Used: 1770566856 Bytes (1688 MB)
```

MASS Network Motif Detection

Motif	Target Freq	Random Mean Freq	Random Std Dev	Z-Score	P-Value
DBw	3.1554 %	4.2259 %	0.00587369	-1.822	0.979
D?	10.6065 %	4.2580 %	0.00741453	8.562	0.000
Dd	1.0960 %	0.4702 %	0.00105330	5.942	0.000
DDW	30.4188 %	37.8394 %	0.01098371	-6.756	1.000
D?c	2.4968 %	3.6167 %	0.00390432	-2.011	0.998
DR	0.6340 %	0.0552 %	0.00040524	14.283	0.000
DqK	0.6979 %	0.7847 %	0.00098705	-0.879	0.809
DNc	0.3195 %	0.0034 %	0.00007690	41.105	0.000
DD	3.2734 %	0.5056 %	0.00204057	13.564	0.000
De	4.2009 %	2.2804 %	0.00368407	5.430	0.000
DBc	2.9244 %	0.5024 %	0.00195762	12.372	0.000
DDc	1.1255 %	0.0196 %	0.00036677	30.154	0.000
DFc	0.2359 %	0.0094 %	0.00011231	20.168	0.000
Dr	0.0393 %	0.0017 %	0.00003945	9.537	0.000
DFw	0.0295 %	0.0783 %	0.00037224	-1.310	0.944
D?	0.5406 %	0.1187 %	0.00052473	8.041	0.000
D-(0.0147 %	0.0000 %	0.00000000	Infinity	0.000
Des	28.6100 %	40.6115 %	0.01446442	-6.297	1.000
Dr	0.1425 %	0.0265 %	0.00017176	6.758	0.000
D?	0.1278 %	0.0001 %	0.00000808	158.116	0.000
DD	9.2303 %	4.5926 %	0.00805014	5.761	0.000

MASS Execution Output

Undirected Power Network Execution Results.

```
[saranyad@hermes01 sequentialApp]$ ./run.sh /home/NETID/saranyad/InputGraph/Power.txt 4 1000 0
filename = /home/NETID/saranyad/InputGraph/Power.txt
Parsing target graph...
Analyzing target graph...
63401 subgraphs enumerated in the target network.
Generating 1000 random graph...
39971271 subgraphs enumerated in the random networks.
Label  RelFreq      RandMeanFreq  Z-Score P-Value
CF      31.271%      32.382%      -4.114 1.000
CN      8.063%       0.060%      228.596 0.000
C^      0.607%       0.000%      2321.925 0.000
C-      0.142%       0.000%      =      0.000
CR      59.434%      67.547%      -30.470 1.000
Cr      0.511%       0.011%      103.563 0.000

SubgraphCount Complete
Overall execution time = 112089 milliseconds
Memory Used: 1356837416 Bytes (1293 MB)
```

NemoLib Execution Output

```
[saranyad@hermes01 SparkNetworkMotif]$ ./run.sh /home/NETID/saranyad/InputGraph/Power.txt 4 1000 0 0
Note: Subgraph.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
added manifest
adding: CanonicalLabeler.class(in = 4845) (out= 2592)(deflated 46%)
adding: Graph.class(in = 4312) (out= 2265)(deflated 47%)
adding: Main.class(in = 4926) (out= 2578)(deflated 47%)
adding: Motif.class(in = 1963) (out= 1029)(deflated 47%)
adding: MotifDetector.class(in = 12750) (out= 5809)(deflated 54%)
adding: Subgraph.class(in = 6927) (out= 3626)(deflated 47%)
adding: Vertex.class(in = 1444) (out= 696)(deflated 51%)

-----
Target Graph Enumeration
-----
63401 subgraphs from 6 classes enumerated in target network
Target enumeration took 8912 milliseconds

-----
Random Graphs Enumeration
-----
Random enumeration took 1191824 milliseconds
63401 subgraphs were enumerated in the target network.
39962071 subgraphs were enumerated in the random networks.
40025472 subgraphs were enumerated in all networks.
Overall execution time = 1202804 milliseconds
Memory Used: 697069656 Bytes (664 MB)

-----
Spark Network Motif Detection
-----
Motif |Target Freq |Random Mean Freq|Random Std Dev |Z-Score |P-Value
-----|-----|-----|-----|-----|-----
CF |31.2708 % |32.3942 % |0.00272085 | -4.129 |0.999
CN |8.0346 % |0.0601 % |0.00034951 | 228.160 |0.000
C^ |0.6072 % |0.0000 % |0.00000305 | 1992.771 |0.000
C- |0.1420 % |0.0000 % |0.00000000 | Infinity |0.000
CR |59.4344 % |67.5352 % |0.00270003 | -30.003 |1.000
Cr |0.5110 % |0.0105 % |0.00005031 | 99.502 |0.000
```

Spark Execution Output

```
[saranyad@hermes01 NetworkMotif]$ ./run.sh /home/NETID/saranyad/InputGraph/Power.txt 4 1000 0 0
MProcess on hermes02.uwb.edu run with command: /usr/bin/java -Xms4g -Xmx12g -cp /home/NETID/saranyad/
1 8 1 58050 /home/NETID/saranyad/AgentCompression/mass_java_app/Applications/NetworkMotif/target
MProcess on hermes03.uwb.edu run with command: /usr/bin/java -Xms4g -Xmx12g -cp /home/NETID/saranyad/
2 8 1 58050 /home/NETID/saranyad/AgentCompression/mass_java_app/Applications/NetworkMotif/target
MProcess on hermes04.uwb.edu run with command: /usr/bin/java -Xms4g -Xmx12g -cp /home/NETID/saranyad/
3 8 1 58050 /home/NETID/saranyad/AgentCompression/mass_java_app/Applications/NetworkMotif/target
MProcess on hermes05.uwb.edu run with command: /usr/bin/java -Xms4g -Xmx12g -cp /home/NETID/saranyad/
5 MProcess on hermes05.uwb.edu 4 8 1 58050 /home/NETID/saranyad/AgentCompression/mass_java_app/Applicat
MProcess on hermes06.uwb.edu run with command: /usr/bin/java -Xms4g -Xmx12g -cp /home/NETID/saranyad/
/NETID/saranyad/AgentCompression/mass_java_app/Applications/NetworkMotif/target
MProcess on hermes07.uwb.edu run with command: /usr/bin/java -Xms4g -Xmx12g -cp /home/NETID/saranyad/
/NETID/saranyad/AgentCompression/mass_java_app/Applications/NetworkMotif/target
MProcess on hermes08.uwb.edu run with command: /usr/bin/java -Xms4g -Xmx12g -cp /home/NETID/saranyad/
7 8 1 58050 /home/NETID/saranyad/AgentCompression/mass_java_app/Applications/NetworkMotif/target
MASS_init: done
97 milliseconds to parse and construct target network
27789 milliseconds to initialize target network data
44 milliseconds to instantiate agents at all graph vertices

-----
Target Graph Enumeration
-----
#Agents: 4941
#Agents: 34779
1156 milliseconds to enumerate all subgraphs in target network
250 milliseconds to collect all subgraphs from all places
63401 subgraphs from 6 non-isomorphic classes enumerated in target network
Target enumeration took 29288 milliseconds

-----
Random Graphs Enumeration
-----
Random enumeration took 705126 milliseconds
63401 subgraphs were enumerated in the target network.
39983983 subgraphs were enumerated in the random networks.
40047384 subgraphs were enumerated in all networks.
Overall execution time = 734543 milliseconds
Memory Used: 6508893776 Bytes (6207 MB)

-----
MASS Network Motif Detection
-----
Motif |Target Freq |Random Mean Freq|Random Std Dev |Z-Score |P-Value
-----|-----|-----|-----|-----|-----
CN |8.0346 % |0.0601 % |0.00032202 | 230.539 |0.000
C^ |0.6072 % |0.0000 % |0.00002660 | 2332.389 |0.000
C- |0.5110 % |0.0108 % |0.00005252 | 95.247 |0.000
CR |59.4344 % |67.5497 % |0.00276851 | -29.313 |1.000
CF |31.2708 % |32.3793 % |0.00278613 | -3.979 |1.000
C- |0.1420 % |0.0000 % |0.00000000 | Infinity |0.000
```

MASS Execution Output

APPENDIX C

MASS execution results with increased computing nodes. This appendix section compares the MASS execution results using 11 (cssmpiNh) and 20 computing nodes (cssmpiNh and hermes0N) against results using 8 computing nodes (hermes0N) discussed in chapter 5. 11 computing nodes testing in cssmpiNh machines excluded cssmpi9h machine due to the reverse DNS lookup issue (which was resolved later).

Table 1. MASS speedup with increased computing nodes for real networks.

Graphs	Motif Size	Subgraphs	MASS Execution Time (sec)			11 Nodes Speedup	20 Nodes Speedup
			8 Nodes	11 Nodes	20 Nodes		
Dolphin	7	550,428	18.042	4.589	15.86	3.932	1.138
Dolphin	8	2,683,740	35.548	14.062	27.458	2.528	1.295
Dolphin	9	12,495,833	129.185	63.315	48.29	2.04	2.675
Dolphin	10	55,824,707	*	354.017	215.918	-	-
Power	5	268,694	20.798	5.69	14.922	3.655	1.394
Power	6	1,260,958	27.657	11.149	15.235	2.481	1.815
Power	7	6,340,413	69.367	25.362	22.251	2.735	3.117
Power	8	33,494,650	246.977	115.124	95.775	2.145	2.579
DIP	3	1,859,101	59.702	21.112	25.614	2.828	2.331
DIP	4	89,371,477	*	*	557.156	-	-
Yeast	3	13,150	12.377	1.649	6.76	7.506	1.831
Yeast	4	183,174	19.167	2.99	10.995	6.41	1.743
Yeast	5	2,508,149	39.434	15.081	14.766	2.615	2.671
Yeast	6	32,883,898	399.557	198.537	171.033	2.013	2.336
P2P	3	341,267	23.689	7.266	6.584	3.26	3.598
P2P	4	10,031,003	77.027	29.571	35.488	2.605	2.171

* Execution terminated due to the memory exhaustion.

Table 1 depicts MASS gained 2x speedup with increased computing nodes from 8 to 11 and 20 machines. Highlighted rows in the table indicate 8 nodes MASS execution faced memory exhaustion, and at max, it can enumerate up to 33.4 million subgraphs. But, with 11 computing nodes, the MASS version can enumerate up to 55.8 million subgraphs, and using 20 computing nodes it can enumerate up to 89.3 million subgraphs. Consequently, additional computing nodes improved MASS execution speed and enhanced scalability.

Table 2. MASS Initialization and Enumeration Time.

Graphs	Motif Size	Initialization Time		Enumeration Time	
		11 Nodes	20 Nodes	11 Nodes	20 Nodes
Dolphin	7	0.271	10.4	3.394	4.562
Dolphin	8	0.258	14.4	11.15	10.988
Dolphin	9	0.265	0.292	45.717	33.254
Dolphin	10	0.267	8.434	262.148	215.918
Power	5	3.387	12.486	1.661	1.833
Power	6	4.343	8.336	5.571	5.409
Power	7	4.475	3.515	18.657	16.42
Power	8	4.351	12.49	96.461	72.43
DIP	3	11.619	14.156	8.494	10.574
DIP	4	*	14.526	*	541.145
Yeast	3	0.942	5.844	0.538	0.76
Yeast	4	0.952	8.807	1.845	2.003
Yeast	5	0.938	0.748	13.706	13.636
Yeast	6	0.922	9.027	196.433	160.73
P2P	3	5.007	3.8	1.709	2.388
P2P	4	5.062	7.841	23.35	26.453

* Execution terminated due to the memory exhaustion.

Table 3. Sequential tools and Spark Execution Time.

Graphs	Motif Size	Subgraphs	FANMOD	NemoLib	Spark
Dolphin	7	550,428	0.786	1.347	9.942
Dolphin	8	2,683,740	3.648	5.994	12.195
Dolphin	9	12,495,833	*	26.756	25.135
Dolphin	10	55,824,707	*	133.011	126.395
Power	5	268,694	0.259	0.767	10.558
Power	6	1,260,958	1.383	2.865	10.899
Power	7	6,340,413	7.151	8.426	14.399
Power	8	33,494,650	42.283	46.784	24.91
DIP	3	1,859,101	1.296	2.916	15.745
DIP	4	89,371,477	73.913	61.731	31.807
Yeast	3	13,150	0.016	0.464	8.416
Yeast	4	183,174	0.291	0.711	8.492
Yeast	5	2,508,149	3.934	3.747	11.364
Yeast	6	32,883,898	96.26	33.063	42.637
P2P	3	341,267	0.26	1.048	13.437
P2P	4	10,031,003	6.935	10.348	16.773

* Infeasible due to FANMOD's motif size limitation (≤ 8).

Table 1 illustrates for small motif size MASS performance improved from 8 nodes to 11 nodes. But, further addition of nodes (11 nodes to 20 nodes), impacts performance negatively due to the time incurred for the network communication. Table 2 indicates that enumeration in 20 nodes execution faster or comparable to the 11 nodes execution, but initialization in 20 nodes execution takes longer time than 11 nodes execution in most cases.

Table 3 lists the execution time for sequential tools (FANMOD, NemoLib) and Spark version (using 8 nodes) depicted in Figure 5.2 and Figure 5.3 (Results chapter). Although extra computing nodes improved MASS performance, Table 1 and Table 3 reveals MASS performance (with 11 & 20 nodes) still lags behind sequential tools and Spark performance (with 8 nodes).