Network Motif Analysis in Clouds - Subgraph Enumeration with Iterative Hadoop MapReduce

Vartika Verma, Paul Park Kwon, Anand Joglekar, and Wooyoung Kim

Computing and Software Systems
School of Science, Technology, Engineering, and Mathematics
University of Washington Bothell
Bothell, WA 98011-8246
Email: \*vartika.nitt@gmail.com, †ppk0907@uw.edu, ‡anandj@uw.edu, §kimw6@uw.edu

Abstract
Finding network motifs in biological networks is a computationally intensive task as it involves traversing through a large network to enumerate all possible subgraphs of a given size, and then determining their statistical uniqueness by sampling subgraphs from a large number (more than 1,000) of random graph pools. There have been parallelization efforts in the past to mitigate the computational intensity for finding network motifs. However, they are either more for the frequent subgraphs in networks rather than network motifs, or require complex manipulations on message passing or designs. Additionally, most of the parallel algorithms are unavailable as tools for use. Here, we introduce a project of ‘complete network motif parallelization.’ This project aims to improve performance of serial algorithms for finding all subgraph patterns with given size in PPI networks by parallelizing them using iterative Hadoop MapReduce on Google Cloud, and then determining their uniqueness through explicit or direct random graph generation to determine network motifs. In this paper, we describe the parallelization of subgraph enumeration and McKay’s canonical algorithms and present the experimental results with a significant improvement in performance, up to 37 times speedup. We are continuing to parallelize network motif significance as a next step, and expecting the completion of the project in near future.

Keywords: Cloud, MapReduce, Network Motifs

1. INTRODUCTION
Protein-protein interaction (PPI) networks are one of the biomolecular networks [1] that have gathered much attention in recent research for cellular data [2]. They constitute physical interactions that occur between organisms or proteins, and it has been found that biological networks including PPI networks have scale-free and small world properties. These properties do not exist in random graphs. It also has been proven beneficial to study such properties, and extract biologically significant information that propels research forward in areas like treating chronic or terminal diseases such as breast cancer [3].

Graph Theory, as one of mathematical disciplines, is used to study those biological networks [2] including PPI networks, where proteins are represented as vertices and edges as their interactions. Understandably, many graph algorithms have been used to analyze PPI networks, such as graph clustering, tree traversal, network centrality, or network motifs. This paper focuses on the network motif analysis and aims to improve the computational performance by parallelizing the process.

Network motifs are recurring patterns which are frequent and unique subgraph types in a network. For example, out of total 6 non-isomorphic graph types of size 4 shown in Figure 1, one experiment has determined frequent and unique patterns as CN, C~, C^, which are network motifs, and they are used to predict essential proteins within a PPI network [4].

![Fig. 1. Possible subgraph types of size 4 for a graph. The patterns are labeled as CR, Cr, CN, CF, C~, C^.

Discovering biological network motifs involves exhaustive subgraph enumeration, isomorphic testing and statistical significance testing procedures. Performance is affected negatively because the number of subgraphs to be found increases exponentially with increase in the size of network as shown in Figure 2a, and with increase in the size of motif as shown in Figure 2b. Therefore, we introduce a project of complete parallelization of network motif analysis based on Iterative MapReduce using Hadoop.
This project includes the following three parallel components: 1) parallel enumeration and graph labeling in the target network; 2) parallel random graph generation and sampling of subgraphs; and 3) parallelization of statistical testing. In this paper, we show the first parallelization task with Iterative Hadoop MapReduce based method. For comparison purpose, we implemented Wernicke’s ESU algorithm [5] for subgraph enumeration and McKay’s Nauty [6] for canonical label generation to discover frequent subgraphs in serial solution. Then we parallelize it using Iterative Hadoop MapReduce and show that more substantial improvement in performance is achieved with larger graph and larger motif size. The second and third parallelization tasks are still in progress and we expect the completion of this project in the near future.

Spark [7] is another framework which can be used for running MapReduce algorithms. It provides an abstraction over map and reduce jobs along with a shared memory which can be queried by the program running on a cluster. We will discuss results of running ESU algorithm on a Spark cluster in Section 2 which was done in a parallel effort as part of a bigger parent project.

Apart from MapReduce, there are other parallelization techniques such as MPI/OpenMP [8] and MASS [9] which overlays data in a distributed array through which a program can access on different nodes. Along with distributed multidimensional array (which are called Places in MASS), MASS library also provides a concept of agents which are set of execution instances that can reside at a single place, access its public data/method members, migrate to other places, spawn child agents, and interact with other agents. We will also discuss results of running ESU algorithm using MPI and MASS which were presented by Kipps et al. [10] as part of the parent project in Section 2.

The rest of paper is organized as follows. Section 2 describes the related work in the field which we evaluated or used for this paper. Section 3 describes serial implementation of subgraph enumeration and Iterative Hadoop MapReduce implementation. We tested the method with several PPI networks and reported the results in Section 4. Conclusion and future works are described in Section 5.

2. RELATED WORK

According to a review by Kim et al. [3], serial network motif detection algorithms are categorized into network-centric or motif-centric methods. Network-centric algorithms search subgraphs in the target network, while motif-centric algorithms collect the instances matching with a query pattern. Most of existing algorithms are, however, infeasible for large (more than 8) motif size due to huge intermediate computations involved.

![Fig. 3. Execution time of Spark implementation for enumerating sub-graphs of size 4 using increasing number of cores.](image-url)
reduce) and process in parallel and collect results to summarize final output. Map task takes in a block of the input and distributes it to the reducers who in turn collect the outputs to produce final result. They have also been proven to work efficiently for graph-like problems. Hadoop MapReduce [12] is open-source software known for its fault tolerance, scalability and ease of use.

Liu et al. [13] has introduced Map Reduce-Based Pattern Finding Algorithm (MRPF) which is a parallel solution for finding frequent subgraph patterns to improve upon the performance. But the statistical significance testing has not been implemented in this algorithm. Hence, it remains inapplicable for the conventional concept of network-motifs.

The Spark framework [14] can also be used for implementing simple MapReduce architecture. The advantage of using Hadoop are easy setup of infrastructure and more control to fine tune the performance of individual map and reduce tasks. On the other hand, the advantages of using Spark are simpler programming model due to shared memory and ease of prototyping on development machine.

A parallel implementation of ESU algorithm was evaluated by a member of our research group at UWB using Spark framework on PPI data with 5173 nodes and 24627 edges. The execution time of enumerating all the sub-graphs of size 4 in the mentioned graph was recorded against the number of cores available to the Spark. As can be seen in Figure 3, the runtime improved with increasing the number of cores which proves the usability of MapReduce algorithm in scaling the ESU algorithm. Along with that, the execution time with increasing size of sub-graph to be detected was also recorded as seen in Figure 4.

In non-MapReduce based approaches, Kipps et al. [10] evaluated the MASS library [9] against MPI based approach. They showed that MASS places-based implementation as well as MPI-based implementation, see Figures 6 and 7, showed good parallelization characteristics by improving runtime with increasing number of available computing cores. Though the MASS agent based implementation struggled to show advantage over serial approach as in Figure 5, because of an explosion of agents, which results in poor memory optimization. It was noted that an agent-based implementation is intuitive but it requires improvements in execution performance.
which is useful when finding motifs of higher order by reusing them.

Iterative MapReduce involves execution of map and reduce tasks in an incremental fashion where the subsequent iteration depend on the output of their predecessors. It is used in various applications like social network analysis, data mining and webpage ranking. Through analyses of the existing solutions, we realize that iterative MapReduce would be more suitable for motif search than simple MapReduce. It is because the intermediate results, from size 2 to \( k \), can be stored and utilized while iterating, unlike other programs providing the final \( k \)-size subgraph counts alone.

![Algorithm 1: ENUMERATE SUBGRAPHS EXTRACTED FROM [5]](image)

**Input:** Graph \( G = (V,E) \), and an integer \( 1 \leq k \leq |V| \)

**Output:** All size-\( k \) subgraphs in \( G \).

1. for each vertex \( v \in V \) do
2. \( V_{Extension} \leftarrow \{ u \in N(v); u > v \} \)
3. call ExtendSubgraph \((\{v\}, V_{Extension}, v)\)
4. return

5. ExtendSubgraph \((V_{Sub}, V_{Extension}, v)\)
6. if \( |V_{Sub}| = k \) then
7. output \( V_{Sub} \), return
8. while \( V_{Extension} != \emptyset \) do
9. get \( w \) from \( V_{Extension} \)
10. \( V'_{Extension} \leftarrow V_{Extension} \cup \{ u \in N(w), u > v \} \)
11. call ExtendSubgraph \((V_{Sub} \cup \{w\}, V'_{Extension}, v)\)
12. return

### 3. METHOD

Parallel network motif detection process can be divided into three subtasks. One is a parallel subgraph enumeration and parallel labeling in the original network. Two is a parallel generation and sampling of random graphs. Three is the parallel computation of statistical testing, such as Z-score or P-value. In this paper, we designed the algorithms of the first task and implemented for testing. We describe the definition of the network motif and the implementation of serial algorithm, and then move to the Iterative Hadoop MapReduce based subgraph enumeration algorithm.

#### 3.1 Network Motif

Given graph \( G = (V,E) \), where \( V \) is a set of vertices and \( E \) is a set of edges, a **network motif** \( m \) is defined as an overly represented connected graph pattern of size \( k \).

**Definition.** Let \( G = (V,E) \) be a graph, and \( 3 \leq k \ll |V| \).

A **network motif** \( m \) is a connected subgraph of size \( k \) in \( G \), which is frequent and unique.

To determine the frequency and uniqueness of motif, \( m \), often more than 1,000 random graphs are generated that preserve the degree sequence of the original network. From the randomly generated graphs, an appropriate number of size \( k \) connected subgraphs are sampled. The frequency of the subgraph \( m \) in the random graph \( R \), \( f_R(m) \), is estimated to obtain a p-value as shown in Equation (1) and Z-score in Equation (2).

\[
p\text{value}(m) = \frac{1}{N} \sum_{n=1}^{N} c(n) \\
\text{where } c(n) = \begin{cases} 
1, & \text{if } f_R(m) \geq f_C(m) \\
0, & \text{otherwise} 
\end{cases}
\]

\[
Z\text{score}(m) = \frac{f_C(m) - \text{average}(f_R(m))}{\text{std}(f_R(m))} 
\]

Here, \( f_R(m) \) and \( f_C(m) \) are the frequencies of \( m \) in the target graph \( G \) and a random graph \( R \), respectively. \( N \) is the number of random graphs and \( \text{average}(f_R(m)) \) and \( \text{std}(f_R(m)) \) refer to the average and standard deviation of frequencies in random graphs, correspondingly. Generally, a graph pattern with p-value less than 0.01 or Z-score greater than 2.0 is considered as a network motif.

#### 3.2 The Serial Solution

**ESU Algorithm.** Wernicke’s ESU algorithm [5] is an exact counting algorithm that has been used for enumerating subgraphs as it is highly efficient. Its efficiency has been proved in the freely available FANMOD tool which implements it [15]. The pseudo-code for the ESU algorithm is given in Algorithm 1 where the input parameters \( G \) represent the input graph and \( k \) represents the subgraph size to be enumerated. The ESU algorithm builds ESU search tree and searches subgraph instances by increasing one node at each level as described in Figure 8.

**McKay’s Canonical Labeling Algorithm.** For finding frequency of the subgraph patterns by identifying isomorphic graphs, McKay’s canonical labeling algorithm [6] has been used. This algorithm is efficient and practical and there is a program that implemented the algorithm named Nauty, which is known as the fastest isomorphism testing program. We imported the ‘Labelg’ module into our program for the canonical labeling.

**Architecture.** Figure 9 depicts the flow of control in the serial program. In the Parser module, input file which is essentially a list of two adjacent nodes is converted into a mapping object which contains the adjacency list of all the nodes. The output from this module is then used by ESU and Labeler modules. The ESU module enumerates subgraphs of input size from the mapping.
object and calls Labeler module to determine intermediate labeling, g6 representation, which is necessary for an efficient canonical labeling at the end by quickly merging them with the external canonical labeling program. The intermediate results are written into files and then sent to the Labeler module which determines their canonical labels by calling the Labelg executable program. In the end, each subgraph is canonically labeled, and each unique pattern obtains the frequencies as the number of corresponding instances.

The GraphLabel class in labeler module receives a list of edges and vertices in the subgraphs generated by the ESU generator module. The list is sent to GraphFormat to convert them into an adjacency matrix, and then into g6 intermediate labeling, and finally to McKay’s implementation of Nauty libraries [6].

**Improvements on the Serial Solution.** The serial solution architecture aforementioned has improved the existing ESU algorithm implementations, and the activities and benefits of the improved serial solution include:

- Memory footprint has been reduced significantly to fit within the default JVM heap size which is 2 GB.
- Efficient data structures such as ArrayLists and LinkedLists replaced the HashTable and HashSet which are too costly when the number of subgraphs increased.
- Improved garbage collection by reducing instantiation of new objects and reusing old ones. This resolved the problems where garbage collector consumed 100% CPU, which should have been just 5%.
- Improved performance by implementing neighborhood caching which stores the neighboring vertices in a cache so that no more recomputing is required in the following iterations.
- Saved substantial memory space by exporting intermediate data into external files instead of keeping in memory. Due to intermediate files, the memory footprint was reduced to around 250 MB from 6GB for subgraph size 4. Also, the run time for size 3 was reduced to 4.9 seconds from 17 seconds.

However, the improved serial solution still has some limitations that necessitate the parallelization of the process. The tight coupling between ESU and Labeler modules easily maximized the memory space by filling out all the subgraph instances. In addition, although the serial solution works swiftly for small subgraph size, a single machine is very limited in terms of computing resources, resulting hours of execution time for relatively large network size or motif size. Therefore, to resolve the

---

Fig. 8. Given the labeled graph $G$, the above ESU-tree extends subgraphs by 1 node in each iteration [5]. The tree has 16 leaves which correspond to 16 size 3 subgraphs of $G$, out of which 1 belongs to type BW and the rest belong to type Bw.

Fig. 9. Flowchart of the serial solution
limitations on the serial solution, we designed and implemented a parallel subgraph enumeration and its labeling task.

3.3 The Parallel Solution

**Architecture.** The parallel solution has been modularized into 3 separate MapReduce jobs: ESU Job, Labeler Job and Combiner Job, as shown in Figure 11. The ESU job is responsible for enumerating subgraphs iteratively by using ESU algorithm where in each step the size of the subgraph is enhanced by one. The steps are repeated until the input size of the subgraph is reached. Then all subgraphs are sent to the Labeler Job which determines their canonical labels. The Combiner job then calculates the frequency of each pattern.

### ESU Job.

ESU Map Reduce Jobs (Figure 12) are initiated by ESU Job which creates a file containing list of all the vertices of the graph. ESU Mapper receives the input file as blocks. In the first iteration, size 2 subgraphs are generated along with their g6 representations. The key-value pairs of subgraphs-g6 representations are passed to the labeler job as input.

### Labeler Job.

The Labeler mappers (Figure 13) switch the input key-value pairs of subgraph and g6 representation to key-value pairs of g6 representation and integer value one, which are then summed up by the Labeler reducers. Labeler reducers also call Labelg executable to determine the canonical labels of the g6 representations. Hence, the final output comprises of a file containing key-value pairs of g6 representation-canonical label and frequencies.

---

**Fig. 10.** Various jobs running one after another in Hadoop cluster. Here we see three ESU jobs corresponding to different iterations, followed by a Combiner job and finally Labeler Job.

**Fig. 11.** Flowchart of the parallel solution.

**ESU Job.** ESU Map Reduce Jobs (Figure 12) are initiated by ESU Job which creates a file containing list of all the vertices of the graph. ESU Mapper receives the input file as blocks. In the first iteration, size 2 subgraphs are generated along with their g6 representations. The key-value pairs of subgraphs-g6 representations are then sent to the reducers to remove any duplicate subgraphs generated. This list then acts as an input to the ESU mappers of the next iteration. The size of subgraph increases by one in each iteration, therefore the iterations continue until the input size of subgraph is reached. The key-value pairs of subgraphs-g6 representations are passed to the labeler job as input.
Combiner Job. The Combiner (Figure 14) takes in the output of Labeler Job as the input. Combiner mappers count the total frequency of canonical labels and output a file containing key-value pairs of canonical labels-total count.

Fine tuning of number of map and reduce tasks. The number of map and reduce tasks are decided by the Hadoop framework based on the size of the input. Idea is that each map task process a certain amount of data which is further passed on to the reduce task. E.g. an input of 1 GB might get divided into 10 map tasks of about 100 MB each if Hadoop is configured to process 100 MB of data at max in each task. At first glance, it might be lucrative to set the limit as low as possible so that large number of tasks are spawned in parallel resulting a huge speedup. But the maximum number of parallel tasks that can run at a time is restricted by the number of computing cores available. Also, while experimenting, we found that as we increase the number of tasks, the amount of data that need to synchronized between those tasks also increase a lot resulting in large overhead. Thus, the solution required some fine tuning to find the right balance of parallelization.

Another problem that we have to tackle was due to the iterative nature of the solution. It should be noted that the number of sub-graphs enumerated in each successive iteration grew exponentially. For example, on one of the network graph, the number of sub-graphs enumerated of size 2, 3, 4, and 5 were in the order of 22 thousand, 763 thousand, 42 million, and 2 billion respectively. As a result, number of tasks appropriate for first iteration were not adequate for last iteration.

Advantages and Limitations. The iterative MapReduce based graph enumeration has the following advantages.

• Well defined components with ESU, labeling and counting jobs separated from each other.
• Using iterations in ESU job, we can enumerate subgraphs of all the sizes until the target size. This will avoid the regenerating process of smaller than size k network motifs once size k motif is determined.
• With distributed data nodes, it is possible to save all enumerated subgraphs which was not possible on a single node.
• Scaling is easy and hence performance gains are enormous. Scaling is done by increasing the number of MapReduce tasks with the increase in size of subgraph to be found or network to be input.

However, overhead is present for small subgraph sizes in small networks due to extra work done by underlying platform for handling jobs, merging and sorting between mappers and reducers. Therefore, the parallel solution works best with fairly large data size or large motif size.
4. EXPERIMENTS

4.1 Platforms Used

The experiments were carried out on 3 platforms which are listed in Table I along with their configurations. It was observed that for one node and PPI network with 5,132 vertices, UWB machines performed the best while the performance on Google Cloud node and Dell XPS machine was comparable as shown in Figure 16. The Hadoop MapReduce framework also provided monitoring tool through browser which was used to monitor the progress of various jobs. Figure 10 shows the different jobs: ESU, Labeler, and Combiner, running one after another. Figure 15 shows multiple map tasks running in parallel for a single ESU job.

4.2 Input Dataset

The experiments were carried out on different PPI networks obtained from DIP database [16]. We experimented with 5 PPI networks as shown in Table II. The subgraph size to be found varied from size 3 to size 7. The size 3 and 4 subgraphs were found for all the networks but the maximum size varied, because visible performance gains depends on the size of motif and size of network. For example, for PPI network with 25,714 vertices the maximum size experimented with was 4 and for the PPI network with 2,365 vertices, the maximum size experimented with was size 7.
4.3 Constraints
Some constraints that were observed while experimenting on different platforms. Although the UWB cluster was the fastest, there was a limitation of disk space. Consequently, it was not possible to test for higher subgraph sizes in the networks larger than 5,132 vertices. This is where the disk space on Google Cloud came in handy. Another constraint is Hadoop configuration. One of the nodes on UWB cluster was not working properly for Hadoop. Hence, the maximum capacity cluster that could be obtained consisted of 1 name node and 14 data nodes instead of 1 name node and 15 data nodes with 4 CPUs per data node. This limited the maximum attainable parallelization on UWB clusters as 56-way instead of 60-way parallelization. In addition, some experiments could not provide fair comparison with serial solution. For example, the experiment on PPI network of 25,714 vertices with size 4 motif search, took days to execute days in serial, and eventually failed to get the final run time. Another case is the experiment on PPI network of 2,365 vertices and size 7 subgraphs.

4.4 Performance Analysis
The parallel solution has been evaluated with various assessment methods, such as, computational time trends with the increased number of nodes, time trends with increased motif size, speedup, efficiency and overhead.

Time vs Computing Nodes. This metric helped us analyze the improvement in time obtained when the nodes were increased for the parallel solution testing. 4 experiments were carried out in total on UWB clusters of 1, 4, 8 and 12 nodes for PPI network with 25,714 vertices and clusters of 1, 4, 8, 12 and 14 nodes for PPI network with 5,132 vertices. The results are shown in Figure 17. It was found that performance for large PPI network with 25,714 vertices improved on adding more nodes even for smaller subgraph sizes 3 and 4. But for medium sized PPI network with 5,132 vertices, although there was an obvious overhead as more nodes were added for smaller subgraph sizes 3 and 4, the performance for the latter case improved for subgraph size 5 onwards.

Time vs Size of Subgraph. This metric helped us analyze the improvement in time observed with respect to the size of the subgraph to be found. 5 experiments were carried out in total, each with PPI networks of 25,714 vertices, 5,132 vertices, 2,365 vertices, 846 vertices, and 900 vertices. There was a common observation in all these experiments. As can be seen in the Figure 18, time taken by the serial solution increases exponentially with increase in size of subgraph to be found. In all the experiments, time taken by the parallel solution increases sub-linearly instead of exponentially and it performs substantially better than the serial solution for larger subgraph sizes.

Speedup. It is defined as the ratio of the runtime of a serial solution to a problem to the runtime of a parallel solution [8].

\[ S(n, p) = \frac{T(n)}{T(n, p)} \]

where \( S(n, p) \) denotes the speedup of the parallel program, \( T(n) \) denotes the runtime of the serial solution and \( T(n, p) \) denotes the runtime of the parallel solution with \( p \) processes. It was found that, from Table II, there was a substantial speedup for large PPI network with 25,714 vertices, which was more than 37 times. For medium PPI network with 2,365 vertices, it was more than 8.7 times. Due to constraints aforementioned, the exact number is unavailable but it is expected that it would be much better than these two figures. It was also observed that, in the cases where number of vertices were comparable, the speedup was affected not only by the number of vertices but also by the number of edges in the subgraph. For example, for PPI network with 846 vertices and 1,607 edges, speedup was observed to be 3.9 times and for PPI network with 900 vertices and 2,262 edges, speedup was 1.33 times. The density of network affects the performance. The trend should be clearer with further experiments in the future for larger subgraph sizes.

### TABLE II

<table>
<thead>
<tr>
<th>Network Size</th>
<th>Subgraph Size</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>846</td>
<td>7</td>
<td>3.92</td>
</tr>
<tr>
<td>900</td>
<td>6</td>
<td>1.33</td>
</tr>
<tr>
<td>2,365</td>
<td>7</td>
<td>&gt;= 8.78</td>
</tr>
<tr>
<td>5,312</td>
<td>5</td>
<td>1.9</td>
</tr>
<tr>
<td>25,714</td>
<td>4</td>
<td>&gt;= 37.81</td>
</tr>
</tbody>
</table>

### TABLE III

<table>
<thead>
<tr>
<th>Network Size</th>
<th>Subgraph Size</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>5,132</td>
<td>3</td>
<td>Overhead____ = (1+12.6)*5.3 = 7.3</td>
</tr>
<tr>
<td>5,132</td>
<td>4</td>
<td>Overhead____ = (1+7.43)*7.19 = 0.24</td>
</tr>
</tbody>
</table>

Efficiency. It is defined as the amount of work done by a serial program with respect to that by the parallel programming including idle time [8].

\[ E(n, p) = \frac{W(n)}{W(n, p)} \]

where, \( E(n, p) \) is the efficiency of parallelization. If \( E(n, p) = 1 \), then the program is exhibiting a linear speedup which is ideal and rarely the case. Two experiments were carried out for determining the efficiency of the parallel solution,
the results of which are shown in Figures 19 and 20. In an experiment with PPI network of 25,714 vertices, it was found that the efficiency decreases initially with increase in the number of nodes for smaller subgraph sizes, but the rate of decrease in efficiency becomes lesser as the nodes are increased. For example, for subgraph size 3, the rate of decrease in efficiency is the greatest from 4 to 8 nodes but is zero between 8 to 12 nodes. The trend is similar in another experiment with PPI network of 5,132 vertices for subgraph size 4.

**Overhead.** It is defined as the increase in the total amount of work done by the parallel program over the amount of work done by a serial program, that is, the amount of work done by the parallel program not done by the serial program [8].

\[ To(p,n) = W(n,p) - W(n) = pT(p,n) - T(n) \]

Possible sources of overhead are communication between nodes, idle time when one node may be working harder than the other node which is sitting idle, extra computation done by the underlying framework like sort and merge operations. For PPI network of 5,132 vertices and subgraph sizes 3 and 4, overhead was calculated for serial solution without MapReduce vs serial solution with MapReduce solution at 1 data node and 1 name node. As shown in Table III, it was found that the overhead decreased with increase in subgraph size. The performance improvement was actually seen in the case of subgraph size 5 where overhead was negligible with respect to the amount of work done by the nodes, but as serial and parallel solutions were run on different platforms due to time constraints, calculation of overhead did not seem fair.

**Fig. 17.** Significant performance improvement can be seen with increase in nodes for large networks in 17a and 17b. There is overhead present for smaller networks in 17c and 17d, though at 12 nodes there is slight improvement.

In summary, we were able to see the performance improvement:

1. When the PPI networks are dense with high number of vertices or edges. For example, PPI network of 25,714 vertices as shown in Figures 17a and 17b.

2. When the motif size is large, for size 5 subgraph in PPI network of 5,132 vertices. For smaller subgraphs, parallelization only added overhead as shown in Figures 17c and 17d.

3. When the underlying platform was faster. For example, UWB cluster with 3.4 GHz CPUs performed...
faster than Google Cloud cluster with 2.5GHz CPUs Figure 16 and Table I.

5. CONCLUSION AND FUTURE WORK

We implemented iterative Hadoop MapReduce to efficiently enumerate subgraphs in biological networks, which will eventually lead to efficient parallel network motif detection. The experimental results show that we could achieve more than 37 times of speedup. Also, we were able to compare the performance in different distribution platforms, including UWB cluster nodes and Google Cloud cluster nodes, to suggest an ideal computing environment for the task of network motif detection.

Future works include completing the project with completion of all subtasks mentioned in Section 1, conducting further tests to discover the exact speedup and overhead numbers, and using different approaches and activities, such as, testing without iterations, and combining network-centric and motif-centric methods.

In current implementation, the overhead of IO between mapreduce tasks cause poor performance for smaller network. The Spark-based and MASS-based approaches which we discussed in Section 2 show good parallelization potential. In Spark-based approach, sharing of mapping object across nodes helped improve the performance. Though MASS-based approach was implemented in UWB clusters, its results show that the approach will perform well on cloud as well. Since both the approaches count only the number of subgraphs generated, they need to be extended to cover the remaining steps in motif detection to gauge their actual performance.

6. ACKNOWLEDGMENTS

We thank University of Washington, Bothell for funding the project and providing all the needed resources. We would like to express the deepest appreciation to professor William Erdly for guiding us through detailed processes. In addition, a thank to professor Munehiro Fukuda who provided basic knowledge of Distributed and Parallel computing necessary for the project. We would also like to thank Matt Kipps for providing results for MASS based approach. Finally, all other hard working team members including Zalak Shah for the support.
7. REFERENCES


Authors

Vartika Verma has 2 years of industrial experience working as a software development engineer and is currently working at Caradigm USA. She received her MS degree in Computer Science and Software Engineering from the University of Washington Bothell, Washington, USA where she also worked as a teaching assistant. She also received her Master of Computer Applications and Bachelor of Computer Applications degrees from the National Institute of Technology, Tiruchirappalli, Tamil Nadu, India and the Panjab University, Chandigarh, India. She is interested in machine learning, distributed and parallel computing.

Paul Park Kwon works as a full-stack software development engineer at Vertafore and has also worked for Adobe in the past. He received his Bachelor of Science degree in Computer Science and Software Engineering from the University of Washington Bothell, Washington, USA where he also worked as a research assistant. He is interested in software development engineering and has good experience working with Hadoop, performance improvement and test suite development.

Anand Joglekar is a software generalist with development experience in messaging and finance domains. He has more than 20 years of experience working in different roles in the software industry. He also has 10 years of experience working as a finance officer and finance executive. He received his MS degree in Computer Science and Software Engineering from the University of Washington Bothell, Washington, USA. He also got his
MBA degree from Institute of Management Development and Research, Pune, Maharashtra, India. His skill set includes object oriented analysis and design, relational database design, implementing end to end solutions in J2EE stack and setting up software development teams and processes.

**Wooyoung Kim** is an acting assistant professor in the Department of Computing and Software Systems (CSS) at University of Washington Bothell, Washington, USA. She received her PhD degree in computer science from Georgia State University, Atlanta, Georgia, USA in 2012, and her MS degree in computer science in 2007 from Georgia Institute of Technology, Atlanta, Georgia, USA. She also obtained BS and MS degrees in Mathematics from Yonsei University, Seoul, Korea in 1996 and 1998, respectively. Her research is bioinformatics and machine learning. As a Co-PI, she received an NSF funding for a project, "RUI: Characterization of Coevolution, Adaptation and Diversification in a Model Microbial Mutualism" in 2012 and currently working with the professor at the Biological Science division at UWB.