Parallelizing and Distributing the Random Graph Generation Algorithms

Vanam Vinay Kumar

A Thesis Submitted to

Indian Institute of Technology Hyderabad

In Partial Fulfillment of the Requirements for

The Degree of Master of Technology

भारतीय प्रौद्योगिकी संस्थान हैदराबाद

Indian Institute of Technology Hyderabad

Department of Computer Science and Engineering

June 2015

Declaration

I declare that this written submission represents my ideas in my own words, and where ideas or words of others have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be a cause for disciplinary action by the Institute and can also evoke penal action from the sources that have thus not been properly cited, or from whom proper permission has not been taken when needed.

[illegible]

(Signature)

(Vanam Vinay Kumar)

CS12M1015

(Roll No.)

Approval Sheet

This Thesis entitled Parallelizing and Distributing the Random Graph Generation Algorithms by Vanam Vinay Kumar is approved for the degree of Master of Technology from IIT Hyderabad

[illegible]

U. RAMAKRISHNA
(———-) Examiner
Dept. of Computer science and Eng
IITH

[illegible]

M. V. Panduranga Rao
(———-) Examiner
Dept. of Computer science and Eng
IITH

N. R. Aravind

(Dr. N R Aravind) Adviser
Dept. of Computer science and Eng
IITH

[illegible]

Manish Singh
(———-) Chairman
Dept. of Computer science and Eng
IITH

Acknowledgements

.

Dedication

Abstract

Random Graphs evolved as a major tool for modeling the complex networks. Random Graphs have wide range of applications.Random Graph can be defined as a probability distribution over graph. Erdos Renyi Random Graph generation model is one of the most popular and best studied models of a network. Erdos Renyi Random Graph model $G(n, p)$ generates random graph with n vertices where each edge appears with probability p. Despite the fact that the evolution of random graphs as data representation and modelling tool, the previous research hasn't focused on the efficiency in generating random graphs. The Random Graph generation algorithms perform poor when generating massively large graphs and fails to use the parallel processing capabilities of modern hardware. The goal of my Thesis work is to parallelize the Random Graph generation models using GPGPU (General Purpose Graphics Processing Unit)to improve the performance.

In this thesis work we have studied the the Erdos Renyi random graph model and we implemented the sequential and parallel algorithms given by the authors of the journal [1]. We have further improved the performance of the parallel algorithms. In addition, we also propose a novel distributed algorithm (DistER) for basic Erdos Renyi algorithm. We also study the Watts and Strogatz model[2] which generates the random graph with real world properties such as high clustering and average path lengths. We propose a novel parallel algorithm for the Watts and Strogatz model.

Contents

Declaration ii
Approval Sheet iii
Acknowledgements iv
Abstract vi

Nomenclature **viii**

1 Introduction **1**
1.1 Random Graphs 1
1.2 Erdos Renyi Random Graph 1
1.3 Watts and Strogatz Model 2

2 Parallelizing ER Model **4**
2.1 Sequential Algorithms 4
2.1.1 Basic ER Algorithm 4
2.1.2 Skipping Edges 5
2.1.3 ZER 6
2.1.4 PreLogZER 6
2.1.5 PreZER 7
2.2 Parallel Algorithms 7
2.2.1 PER 9
2.2.2 PZER 9
2.2.3 PPreZER 11

3 Distributing the Erdos Renyi Random Graph Generation **12**
3.1 Hadoop Map Reduce 12
3.2 Distributed ER Algorithm DistER 13

4 Parallelizing Watts and Strogatz Model **15**
4.1 Sequential Watts Strogatz Algorithm 16
4.2 Parallel Watts Strogatz Algorithm 16

5 Experimental Setup and Results **20**
5.1 Experimental Setup 20
5.2 Erdos Renyi Model 20
5.2.1 Speedup assessment 20

5.2.2 Results 20
5.3 Watts Strogatz Model 23

6 Conclusion **28**

References **29**

Chapter 1

Introduction

1.1 Random Graphs

Random Graphs are recognized as one of the most important concepts in modern Theoretical Computer Science. The theory of random graphs is the combination of the probability theory and the Graph Theory, so random graph theory lies in the intersection between them. In mathematics, distribution of the probability over the graph is defined as a random graph. Large graphs appear in many contexts such as the Biological networks, Social networks, Internet and in many other networks. The Random Graphs evolved as a major tool for analysis and modeling of complex networks.[3][4] $G(n, p)$ is random graph with n vertices labeled from $\{0, ...n - 1\}$ where (a, b) is an edge between vertex a and vertex b with some probability p [5]. Inclusion of an edge is independent of other edges.In mathematics random graphs [6] are used to study the properties of the graph and to answer questions about the properties. The applications of it are found in the complex networks which requires modeling.

1.2 Erdos Renyi Random Graph

The Erdos Renyi Random Graph[7] generation method has wide range of applications. We live in a world which is composed of complex networks. Examples include the Internet, networks of telephone users, airline networks, power grids, etc. The network structure has a profound effect on behavior of the network and performance of the network.[8] In the field of biology, the Erdos Renyi algorithm is accepted as a basic model to study biological networks[9]. Random graphs are used to model the interactions among biological system components, like genes, proteins, or metabolites[10]. From their global topology and organization one can learn non-trivial, systemic properties of organisms[11]. In the field of Mathematics, the Erdos Renyi random graphs are used in graph coloring problem and generating spanning trees randomly[12]. In the field of Networks, uniform random graphs based on Erdos Renyi are utilized to explore data search and replication in peer to peer networks[13]. Theory of random geometric graphs have emerged as essential tools in the analysis and design of large Wireless Networks. In Ad-hoc sensor networks, Random graphs are used to model the network[14][15].

The basic Erdos Renyi random graph generation implementation doesn't scale to very large graphs. It performs poor when generating massively large graphs and fails to use the parallel processing capabilities of modern hardware. The goal of my Thesis work is to implement a parallel algorithm for the basic Erdos Renyi algorithm using GPGPU (General Purpose Graphics Processing Unit) later extending my work to other methods.

The authors of journal paper [1] implements the basic ER algorithm and then proposes modifications to the basic ER algorithm to improve the efficiency.The first algorithm is basic ER algorithm. In the second algorithm ZER, instead of checking for probability of every edge we just calculate the skip value i.e. the expected number of edges that can be skipped to include next edge. ZER calculates logarithm values repeatedly, So in third algorithm PreLogZER we calculate and stores the logarithm values prior to computation which improves the performance of ZER. In the final algorithm PreZER, we avoid the logarithm computation overhead by using cumulative probability function. Then finally we implement the parallel versions of the algorithms ER, ZER and PreZER named as PER, PZER and PPreZER respectively.

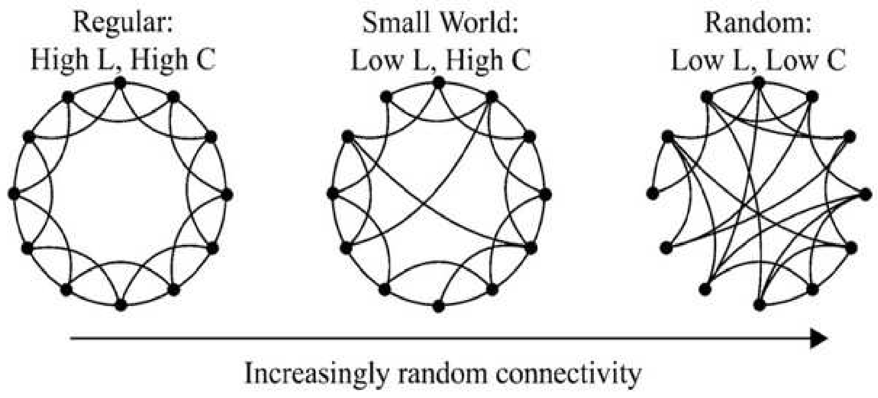
In this thesis work, we implemented the sequential algorithms namely ER, ZER, PreLogZER, Prezer and parallel algorithms PER,PZER and PPreZER of the ER model respectively. In the paper [1], the authors didn't use a real pseudo random number generator in parallel algorithms and the paper doesn't describe about the space efficiency. In our work, we used latest CUDA[16] cuRAND library framework for generating random numbers parallely on GPU. All the algorithms were space efficient as we represent each edge with a single bit.

In addition to parallel algorithms for ER model, we also propose a novel distributed algorithm i.e. DistER for basic ER algorithm using Hadoop distributed framework.

1.3 Watts and Strogatz Model

The Watts and strogatz model[17] generates the random graph with small world properties such as high clustering and average path lengths. The real world networks Internet, Social networks, Biological networks and other real world networks tend to have the small world property of high clustering and short path lengths. The ER model lacks high clustering and it follows unrealistic Poisson distribution.[18]

The Watts and Strogatz model $WS(n,k,p)$ where n is the number of nodes and k is the degree of each vertex and p is the rewiring probability. $WS(n,k,p)$ initially constructs a regular lattice with each node is connected to k neighbors such that $k/2$ nodes on each side. Then we take every edge in the regular lattice and rewire it with a probability p. Here rewiring means replacing an edge (i,j) with (i,r) where r is a uniform random value chosen from $\{0,1,2...n-1\}$. The below figure depicts the Watts and Strogatz algorithm.



In this thesis work, We studied and implemented the basic watts and strogatz algorithm. We propose a novel parallel algorithm i.e. PWattsStrog which improves the performance of the watts and strogatz algorithm.

Chapter 2

Parallelizing ER Model

In this chapter initially we introduce the basic Erdos Renyi algorithm. We deduce a skip value using mathematical formulas as ER algorithm employs the Bernoulli process in selecting the edges. Using the skip value we will bypass some edges in computation which improves the performance.We improve the basic Erdos Renyi successively by presenting the improved sequential algorithms. After presenting the improved ER sequential algorithms namely ER,ZER and PreZER, then we devise the parallel algorithms for each sequential ER algorithms namely PER,PZER and PPreZER respectively. As the inclusion of an edge in the graph is independent of other edges, the selection of the edges can be done concurrently. In all the algorithms in this thesis we used adjacency matrix representation to represent the random graph.

2.1 Sequential Algorithms

Graph G(n,p) has n vertices and p is the probability that an edge exists between a pair of vertices. Total number of edges depends on the structure of the graph. A directed graph with self-loops can have a maximum of n^2 edges and in case of undirected graph with no self-loops, the maximum possible number of edges are n(n-1)/2. In this work, we considered directed graph with self loops.

2.1.1 Basic ER Algorithm

The algorithm 1 generates a directed random graph with self loops. Graphs can be distinguished by using the maximum number of edges in that graph for example a regular graph with mean degree k (i.e. k-regular graph) will have $n \times k$ number of edges . It is possible to implement a generalized algorithm for $G(n, p)$ model which can produce desired graph type using number of edges.Consider the maximum number of edges E for a certain graph, then its is sufficient enough to generate index values from $0 to E - 1$ to represent the graph. These indices i.e. from $0 to E - 1$ can be decoded with respect to the particular desired graph type. For example consider an undirected graph of 6 nodes (labeled from 0 to 5) and without self loops. The edge number 10 represents an edge between node 3 and node 4. c. For a directed graph with self loops the edge number decoding can done as followed.

$$i = \lfloor \frac{ind}{n} \rfloor \;\; and \;\; j = ind \; mod \; n$$

Algorithm 1 Basic ER algorithm

```
Input:      $\eta$ : Number of nodes
            $\rho$ : Edge occurrence Probability
Result:     Random Graph G(n,p)

Intialization;
$G = \phi$
for $p = 0$ to $\eta - 1$ do
    for $q = 0$ to $\eta - 1$ do
        $\gamma$ is a random number $\in$ [0,1) which is generated uniformly.
        if $\gamma < \rho$ then
            $G \leftarrow (p, q)$;
        end if
    end for
end for
```

where (i, j) is a directed edge from i to j.

The generalized ER algorithm can be written as

Algorithm 2 ER algorithm

```
Input:      E : Number of Edges
            $\rho$ : Edge occurrence probability
Result:     Random Graph G(n,p)

Initialization;
$G = \phi$
for $i = 0$ $to$ $E - 1$ do

    $\gamma$ is a random number $\in$ [0,1) which is generated uniformly.
    if $\gamma < \rho$ then
        $G \leftarrow i$;
    end if
end for
```

2.1.2 Skipping Edges

As the ER algorithm successively selects the edges with probability p. The number of selected edges has a Binomial distribution B(E,p).
The mean of the random graph is:

$$\mu = p \times E \tag{2.1}$$

and it's standard deviation is:

$$\sigma = \sqrt{p \times (1 - p) \times E} \tag{2.2}$$

The probability mass function i.e. the probability that k number of edges are skipped before selecting next edge is

$$f(k) = (1 - p)^k \times p \tag{2.3}$$

The respective distribution function i.e. the probability that any number of edges from 0 to k is skipped is

$$F(k) = \sum_{i=0}^{k} f(i) \quad = 1 - (1-p)^{k+1} \tag{2.4}$$

Let α be the uniform random number in (0,1]. Then, the probability that α falls in interval (F(k-1), F(k)] is exactly F(k) - F(k - 1) = f(k). In other words, the probability that k is the smallest positive integer such that $\alpha \leq F(k) is f(k)$. To skip the k edges $F(k) \geq \alpha$. Hence

$$F(k-1) < \alpha \leq F(k) \tag{2.5}$$

$$1 - (1-p)^k < \alpha \leq 1 - (1-p)^{k+1} \tag{2.6}$$

Let $1 - \alpha = \psi$ and $\psi \in [0, 1)$

$$(1-p)^{k+1} \leq \psi < (1-p)^k \tag{2.7}$$

From eq.[2.7], k can be defined as

$$k = max(0, \lceil \log_{1-p} \psi \rceil) \tag{2.8}$$

2.1.3 ZER

In this algorithm instead of checking for probability of every edge we just calculate the skip value i.e. the expected number of edges that can be skipped to include next edge. The skip value is calculated by using the equation [2.8].

Algorithm 3 ZER algorithm

```
1: Input:       E : Possible Number of Edges
2:              p : Edge occurrence probability
3: Result:      Random Graph G(n,p)
4:
5: Initialization;
6: $G = \phi$;
7: $i = -1$;
8: while $i < E$ do
9:     Generate uniform random number $\psi \in [0, 1)$;
10:    Compute the skip value $k = max(0, \lceil \log_{1-p} \psi \rceil - 1)$;
11:    $i = i + k + 1$;
12:    $G \leftarrow i$;
13: end while
```

2.1.4 PreLogZER

When n is large then the ZER algorithm suffers from logarithm computation overhead. The ZER algorithm has overlapping sub structure property as it computes the logarithm values repeatedly. So to improve the ZER, we pre-calculate and store the logarithm values. The algorithm is as follows.

Algorithm 4 PreLogZER algorithm

```
Input:          E : Maximum Number of Edges
                p : Probability of en edge occurrence
Result:         Random Graph G(n,p)

Initialization;
$G = \phi$;
$i = -1$;
$c = \log(1-p)$;
for $i = 1$ to $MAXRANDOM$ do $LogArray[i] = \log(i/MAXRANDOM)$
end for
while $i < E$ do
    Generate uniform random number $\psi \in [0, MAXRANDOM)$;
    Compute the skip value $k = max(0, \lceil \frac{\log \psi}{c} \rceil - 1)$;
    $i = i + k + 1$;
    $G \leftarrow i$;
end while
```

2.1.5 PreZER

In the previous algorithm we have tried to reduce the logarithm computation overhead by precomputing all the logarithm values we may need. The efficiency will be further improved if we can avoid the logarithm computation altogether. We use the cumulative distribution function to devise the skip value.
Initially we precompute the m number of cumulative probabilities and store in memory for further reference. Now we generate a uniform random number $\gamma \in$ [0,1) and compare with cumulative probabilities one by one. That is we compare the γ value with cumulative probability F(i) for i = 0 to m. If the random value γ is lesser than any of the cumulative probability $F(i)$ then we set the skip value as i such that i will be the smallest value that satisfies the condition. If all the precomputed cumulative probabilities are lesser than the random value γ then we compute the skip value by using the logarithm computation used in the Algorithm 3 i.e. ZER

2.2 Parallel Algorithms

In this section we introduce the parallel algorithms for the Erdos Renyi sequential algorithms. We implement the parallel algorithms PER, PZER and PPreZER of the sequential algorithms ER, ZER and PreZER respectively. We have implemented the parallel algorithms using CUDA (Compute Unified Device Architecture) C++ programming platform for performing parallel computations on GPU (Graphics Processing Units). Each parallel algorithm is comprised of sequential code and parallel code, that is host code and device code. The host code is executed on the CPU and the device code is executed on GPU (Graphics Processing Unit). We use the latest CuRAND library framework for generating the random numbers on GPU. CuRAND library provides methods to generate the pseudo random numbers on both the host(i.e. CPU) and device (i.e. GPU). We setup the seeds for CuRAND sates for generating the random numbers on GPU. In our implementation we use 65,536 parallel threads, each thread executes the parallel code independently on it's own. Each thread will write the result to a shared memory independently.

Algorithm 5 PreZER algorithm

```
Input:          E : Maximum Number of Edges
                p : Probability of en edge occurrence
                m : Number of Pre-Computations
Result:         Random Graph G(n,p)

Initialization;
$G = \phi$;
$i = -1$;
for $i = 0$ $to$ $m$ do
    Compute cumulative probability F[i];
end for
while $i < E$ do
    Generate uniform random number $\alpha \in (0, 1)$;
    j=0;
    while $j \leq m$ do
        if $F[j] > \alpha$ then
            Set the skip value k = j;
            break;
        end if
        $j = j + 1$;
    end while
    if $j = m + 1$ then
        Compute the skip value $k = max(0, \lceil \log_{1-p}(1 - \alpha) \rceil - 1)$;
    end if
    $i = i + k + 1$;
    $G \leftarrow i$;
end while
```

2.2.1 PER

The edges set is divided into multiple edge blocks based on total number of threads. Each block will be processed concurrently. Each thread independently checks whether to include the edge or not by using the probability p and assigns either 1 (means edge included in graph) or 0(means edge is not included in graph) in it's corresponding index location of *deviceArray*. The *deviceArray* is a data structure which collects the result of each thread running concurrently. Finally the *deviceArray*'s of all the blocks are combined to get the random graph. For example consider that we have 1024 edges and 256 number of parallel threads, then the edge set is divided into 4 blocks of 256 edges each. The edges in each block processed concurrently. So we need 4 iterations to process all the edges. The results of all 4 iterations are combined to get the final random graph.

Algorithm 6 PER algorithm

```
Input:     E : Number of Edges
           p : Edge occurrence probability
Result:    Random Graph G(n,p)

Initialization;
$G = \phi$;
$b = Number\ of\ Thread\ Blocks$
$t = Number\ of\ Threads\ per\ Block$
$B = b \times t$;
$deviceArray[B] = \phi$
$iterations = \lceil \frac{E}{B} \rceil$
In parallel: Set up seeds for CuRAND States for generating random numbers on GPU.
for $i = 0$ to $iterations - 1$ do
    In Parallel: All the threads run the following code in parallel.
        $index = threadIndex + blockIndex \times t$
        Generate uniform random number $\alpha \in [0, 1)$;
        if $\alpha < p$ then
            $deviceArray[index] = 1$;
        else
            $deviceArray[index] = 0$;
        end if
    End Parallel Code
    for $i = 0$ to $B - 1$ do
        $G \leftarrow deviceArray[i]$;
    end for
end for
```

2.2.2 PZER

Each thread processing unit must know its starting edge index for generating the absolute edge index using the skip value. The starting edge index of a thread processing unit can not be determined unless all the preceding thread processing units finished the processing. So to solve this problem we separate the skip value generation from edge index computation. We generate the skip values in parallel and we use those skip values in sequential code to find the edge index and include the edge in the Graph.

Algorithm 7 PZER algorithm

```
1: Input:      E : Number of Edges
2:             p : Edge occurrence probability
3: Result:     Random Graph G(n,p)
4:
5: Initialization;
6: $G = \phi$;
7: $b = Number\ of\ Thread\ Blocks$
8: $t = Number\ of\ Threads\ per\ Block$
9: $B = b \times t$;
10: $deviceArray[B] = \phi$
11: $E = Maximum\ number\ of\ edges$
12: $LastEdge = -1$;
13: In parallel: Set up seeds for Curand States for generating random numbers on GPU.
14: while $LastEdge < E$ do
15:     In Parallel: All the threads run the following code in parallel.
16:         $index = threadIndex + blockIndex \times t$
17:         Generate uniform random number $\alpha \in [0, 1)$;
18:         $Skipvalue = max(0, \lceil \log_{1-p} \alpha \rceil - 1)$;
19:         $deviceArray[index] = Skipvalue$;
20:     End Parallel Code
21:     for $i = 0\ to\ B - 1$ do
22:         $LastEdge = LastEdge + deviceArray[i] + 1$;
23:         if $LastEdge \geq E$ then
24:             Break;
25:         end if
26:         $G \leftarrow LastEdge$;
27:     end for
28: end while
```

2.2.3 PPreZER

The PPreZER shares the structure of the algorithm PZER. The precomputation of m cumulative probability values can be done prior to the parallel processing code. We use the precomputed cumulative probabilities in the parallel code to find the skip value.

Algorithm 8 PPreZER algorithm

```
Input:      E : Number of Edges
            p : Edge occurrence probability
            m : Number of pre-computations
Result:     Random Graph G(n,p)

Intialization;
$G = \phi$;
$b = Number\ of\ Thread\ Blocks$
$t = Number\ of\ Threads\ per\ Block$
$B = b \times t$;
$deviceArray[B] = \phi$
$LastEdge = -1$;
for $i = 0\ to\ m$ do
    Compute cumulative probability F[i];
end for
In parallel: Set up seeds for Curand States for generating random numbers on GPU.
while $LastEdge < E$ do
    In Parallel: All the threads execute the following code in parallel.
        $index = threadIndex + blockIndex \times t$
        Generate uniform random number $\psi \in [0, 1)$;
        while $i < m$ do
            if $F[i] > \psi$ then
                $Skipvalue = i$;
                Break;
            end if
            $i = i + 1$;
        end while
        if $i = m + 1$ then
            $Skipvalue = max(0, \lceil \log_{1-p}(1 - \alpha) \rceil - 1)$;
        end if
        $deviceArray[index] = Skipvalue$;
    End Parallel Code
    for $i = 0\ to\ B - 1$ do
        $LastEdge = LastEdge + deviceArray[i] + 1$;
        if $LastEdge \geq E$ then
            Break;
        end if
        $G \leftarrow LastEdge$;
    end for
end while
```

Chapter 3

Distributing the Erdos Renyi Random Graph Generation

When ever the random network to be generated is massively large graph then it's better to perform the random graph generation computations in distributed manner. As the graph size (i.e. number of nodes) increases the single system won't be sufficient enough for holding the large data and performing computations. So we propose a novel distributed algorithm for baseline ER algorithm.
A distributed system is a collection of multiple hosts that are connected through a network to perform a portion of the computation task. A distributed algorithm runs on independent processors which are connected through a network. The distributed algorithms are subset of parallel algorithms. We implemented a novel distributed algorithm for Erdos Renyi random graph generation model. The Hadoop Map Reduce[19] programming framework is used for developing the distributed algorithm of ER model.

3.1 Hadoop Map Reduce

Apache Hadoop is an open source framework for distributed processing using a simple programming model. Hadoop consists of two core components namely Hadoop Distributed File System (HDFS) and the Map Reduce programming software framework.

- **HDFS:** Hadoop Distributed File System is a scalable high performance distributed file system. It is responsible for storing the data on cluster. HDFS stores the data as blocks or chunks of size 64 MB. HDFS stores multiple copies of the data blocks.
- **Map Reduce:** Hadoop Map Reduce is a java based programming framework for developing distributed programs which runs in parallel on large clusters.

Hadoop Distributed File System (HDFS) has a master slave architecture. Name Node works as master node and Data Nodes works as slave nodes. The Map Reduce framework is also a master slave architecture, in which JobTacker is the master and TaskTracker is the slave.

- **NameNode:** NameNode is a master node and it manages the DataNodes. It take care of file system name space. It replicates the data blocks on multiple nodes

- **DataNode:** It works as a slave node for NameNode. It stores the data blocks in local file system. It also stores the meta data for each block. It sends the data blocks to other nodes.

- **JobTracker:** JobTracker works as a master node in Map Reduce framework. JobTracker takes requests from clients and assigns TaskTracker with the tasks. JobTracker locates the DataNode where data is locally present. If that is not possible it tries to assign the tasks to TAskTracker within the same rack.

- **TaskTracker:** The TaskTracker works as a slave node for JobTracker. It executes the tasks sent by the JobTracker and reports the status.

The Hadoop distributed program is written using Mapper and Reducer methods. The Mapper code reads the input files as key value pairs and outputs set of intermediate key value pairs. Reducer takes input from the different Mappers as key value pairs. It processes the data and produces intermediate key value pairs. In this way one can write multiple mappers and multiple reducers for achieving distributed programming.

3.2 Distributed ER Algorithm DistER

In this section we propose a novel distributed algorithm DistER for the Erdos Renyi random graph generation model. It consists of two methods DistributedERMapper and DistributedERReducer.

The DistributedERMapper will take a input file which contains number of nodes as number of lines. That is the file contains empty or dummy lines with number of lines equals to number nodes of the random graph to be generated. The DistributedERMapper process the input file and generates intermediate $< key, value >$ pairs for each word. It uses the line number as the key and the word as the value.

The DistributedERReducer will receive the intermediate key value pairs generated by the DistributedERMapper. All the key value pairs which has the same key value will go into same reducer. In the DistributedERReducer we generate the neighborhood list for each node independently. Here the key i.e. the $lineNo$ works as the node for which the neighborhood list need to be generated. As we are considering the random graph with directed edges and self loops, each node can independently choose the neighborhood list using probability. The DistributedERReducer generates a uniform random number $\beta \in (0, 1]$ and it checks with probability p, if $\beta < p$ then it includes the edge i.e it sets the value 1 else it sets the value 0. It will check for all n possible edges and collects the predicates, where n is number of nodes. Finally the DistributedERReducer outputs the $< Null, result >$ key value pair where $result$ is the neighborhood list for that specific node $n = lineNo$. All the reducers will write the neighborhood list to the file that forms the random graph.

The Algorithm 9 which is given below shows proposed Distributed ER algorithm DistER.

Algorithm 9 Distributed ER Algorithm DistER

```
Input:      E : Number of Edges
            p : Edge occurrence Probability
Result:     Random Graph G(n,p)

procedure DistributedERMapper(String $key$, String $value$)
    for each word $w$ in $value$ do
        EmitIntermediate($lineNo, w$)
    end for
end procedure

procedure DistributedERReducer(String $key$, Iterator $value$)
    for $i = 1$ to $n$ do
        Generate uniform random number $\beta \in [0, 1)$
        if $\beta < p$ then
            $result \leftarrow result + 1$
        else
            $result \leftarrow result + 0$
        end if
        EmitFinal($NULL, result$)
    end for
end procedure
```

Chapter 4

Parallelizing Watts and Strogatz Model

Most large scale networks exhibit small world property. Examples include Internet, Neurons, Social Networks etc. In small world networks most nodes are not neighbors of one another but most nodes can be reached from every other by small number of hops or steps. Real world networks tend to have high clustering. Watts and Strogatz generates random graph with small world properties i.e. short average path lengths and high clustering. In small world network typical distance between two randomly chosen nodes is $L = \log N$.

Watts and Strogatz algorithm initially constructs a regular ring lattice with mean degree and then progressively rewires it one edge after the other until we obtain a random graph. During this process the farther the node being connected to the more the graph diameter is reduced.

Given the number of nodes n, mean degree k and rewiring probability p, the Watts Strogatz model generates the undirected random graph WS(n,k,p) in following way.

- Construct a regular ring lattice with each node connecting to k neighbors, that is $k/2$ on each side of the node. If $(0, 1, 2, ..., n-1)$ are nodes then an edge (i, j) exists if and only if $0 < |i - j| \bmod (n - \frac{k}{2}) \leq \frac{k}{2}$

- For every node $i = (0, 1, 2, ..., n-1)$ take every edge (i, j) and rewire it with probability p. Rewiring is done by replacing the edge (i, j) with edge (i, r) where r is chosen uniformly random from $(0, 1, 2, ..., n-1)$ and avoiding self loops $r \neq j$.

For example, constructing a regular ring lattice for number of nodes $n = 9$ and the mean degree

$k = 4$ is given below as an adjacency matrix representation.

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

4.1 Sequential Watts Strogatz Algorithm

Sequential Watts Strogatz algorithm consists of two procedures namely RingLattice and Rewire. The RingLattice procedure takes graph G and mean degree k as input. G is a adjacency matrix representation of the random graph. Watts Strogatz algorithm produces random graph with undirected edges and without self loops.

In the procedure RingLattice we include the $k/2$ edges on left side and the $k/2$ edges on the right side of the each node in the adjacency matrix representation. As we are using the adjacency matrix representation, if the current node is i and mean degree is k then we include the right neighboring edges (i, j_R) where $j_R = i+1, i+2, ..., i+\frac{k}{2}$ and left neighboring edges (i, j_L) where $j_L = i-1, i-2, ..., i-\frac{k}{2}$. While adding the edges if right neighbor node j_R value reaches maximum number of nodes n then we reset the value to $j_R = 0$. If the left neighbor node value j_L goes beyond the zero value then we reset the value $j_L = n-1$.

After adding k neighbors for each node that is after forming the regular ring lattice with the mean degree k then we call the procedure Rewire. The Rewire procedure takes each edge that is included earlier in the RingLattice procedure and rewires it with probability p. For each edge, we generate a uniform random number $\gamma \in (0, 1]$ and check with probability p i.e. if $\gamma < p$ the we delete that edge (i, j) and we reconnect it to a random node $randNode \in [0, n-1]$ which is generated uniformly random. That is we replace the edge (i, j) with $(i, randNode)$ if it satisfies the condition else the edge (i, j) remains included. The Algorithm 10 shows the sequential Watts Strogatz algorithm.

4.2 Parallel Watts Strogatz Algorithm

In this section we propose a novel parallel algorithm for sequential Watts Strogatz algorithm. Initially it calls the RingLattice procedure and forms the ring lattice.As the Watts Strogatz algorithm generates undirected graph we can not perform including or deleting the edges parallel manner, because the deletion or inclusion of an edge (i, j) is dependent on (j, i) in adjacency matrix representation of the random graph G. So we separate the edge inclusion or deletion from the rewiring phase. So in the parallel code we generate the uniform random number $\gamma \in [0, 1)$ and check with rewiring probability p. That is if $\gamma < p$ then we generate a uniform random node $randNode \in [0, n-1]$ and

Algorithm 10 Watts Strogatz Algorithm

```
Input:      n : Maximum Number of Nodes
            p : Rewiring Probability
            k : Mean Degree
Result:     Random Graph G(n,k,p)

procedure WattsStrogatz(n,k,p)

    Initialization;
    $G = \phi$;
    Call procedure $RingLattice(G, k)$;
    Call procedure $Rewire(G, k, p)$;
end procedure

procedure RingLattice
    for $i = 0$ to $n - 1$ do
        $j_L = i$;
        $j_R = i$;
        for ( do $a = 1$ to $k/2$ )
            $j_R = j_L + 1$;
            if $j_R \geq n$ then
                $j_R = 0$;
            end if
            $G \leftarrow (i, j_R)$
            $j_L = j_L - 1$;
            if $j_L < 0$ then
                $j = n - 1$;
            end if
            $G \leftarrow (i, j_L)$
        end for
    end for
end procedure

procedure ReWire
    for $i = 0$ to $n - 1$ do
        $j_L = i$;
        $j_R = i$;
        for ( do $a = 1$ to $k/2$ )
            $j_R = j_R + 1$;
            if $j_R \geq n$ then
                $j_R = 0$;
            end if
            Generate uniform random number $\gamma \in [0, 1)$
            if $\gamma < p$ then
                Delete the Edge $(i, j_R)$;
                Generate a uniform random integer $randNode \in [0, n - 1]$ and $randNode \neq j_R$
                $G \leftarrow (i, randNode)$
            end if
            $j_L = j_L - 1$;
            if $j_L < 0$ then
                $j = n - 1$;
            end if
            Generate uniform random number $\beta \in [0, 1)$
            if $\beta < p$ then
                Delete the Edge $(i, j_L)$;
                Generate a uniform random integer $randNode \in [0, n - 1]$ and $randNode \neq j_L$
                $G \leftarrow (i, randNode)$
            end if
        end for
    end for
end procedure
```

store the randNode in deviceArray. If the condition is not satisfied the edge will have to remain as it is so to indicate this we store -1 in deviceArray. Now in the sequential code we check the deviceArray, if it contains other than -1 then the current edge will be replaced with the value in the deviceArray otherwise the edge remains same. The Algorithm 11 is the novel parallel algorithm for watts and strogatz.

Algorithm 11 Parallel Watts Strogatz Algorithm

```
Input:        n : Maximum Number of Nodes
              p : Rewiring Probability
              k : Mean Degree
Result:       Random Graph G(n,k,p)

Intialization;
$G = \phi$;
$b = Number\ of\ Thread\ Blocks$
$t = Number\ of\ Threads\ per\ Block$
$B = b \times t$;
$LastEdge = 0$;
$deviceArray[B] = \phi$
Call procedure RINGLATTICE(G,k);
In parallel: Set up seeds for Curand States for generating random numbers on GPU.
while $LastEdge < E$ do
    In Parallel: All the threads execute the following code in parallel.
        $index = threadIndex + blockIndex \times t$
        Generate uniform random number $\alpha \in [0, 1)$;
        if $\alpha < p$ then
            Generate uniform random number $RAND \in [0, n-1]$;
            $deviceArray[index] = RAND$;
        else
            $deviceArray[index] = -1$;
        end if
    End Parallel Code
    for $i = 0\ to\ \frac{B}{k} - 1$ do
        for $j = 0\ tok - 1$ do
            $ind = i \times k + j$
            if $deviceArray[ind] \neq -1$ then
                Delete the old Edge
                $iNode = LastEdge + 1$
                $G \leftarrow (iNode, deviceArray[index])$
            end if
        end for
    end for
    $LastEdge += \frac{B}{k}$
end while

procedure RINGLATTICE
    for $i = 0$ to $n - 1$ do
        $j_L = i$;
        $j_R = i$;
        for ( do $a = 1\ to\ k/2$ )
            $j_R = j_R + 1$;
            if $j_R \geq n$ then
                $j_R = 0$;
            end if
            $G \leftarrow (i, j_R)$
            $j_L = j_L - 1$;
            if $j_L < 0$ then
                $j = n - 1$;
            end if
            $G \leftarrow (i, j_L)$
        end for
    end for
end procedure
```

Chapter 5

Experimental Setup and Results

5.1 Experimental Setup

All the sequential algorithms ER,ZER,PreZER,WattsStrogatz were implemented in GNU C++ with C++ 2011 standard.Parallel algorithms were implemented using CUDA (Compute Unified Device Architecture) C++ programming platform for performing parallel computations on GPU (Graphics Processing Units). Each parallel algorithm is comprised of sequential code and parallel code, that is host code and device code. The host code is executed on the CPU and the device code is executed on GPU (Graphics Processing Unit). In our parallel algorithm implementation we used 65,536 parallel threads for each iteration, each thread executes the parallel code independently on it's own. All the programs were executed on a GPU server with Ubuntu 14.04 LTS operating system and Intel Xeon CPU 2.70GHz machine with nVidia Tesla K20X GPGPU card.
The default setting for ER algorithms are 10000 nodes and m value is set to 300.For Watts Strogatz algorithms the default values are probability p=0.5 and number of nodes is 20000.

5.2 Erdos Renyi Model

5.2.1 Speedup assessment

on average the algorithms ZER, PreZER, PER, PZER and PPreZER have a speedup factor of 5,4,13, 23 and 26. If the probability $p \leq 0.5$ then the algorithms ZER, PreZER, PER, PZER and PPreZER have a speedup factor of 9,8,13,33 and 36. All the algorithms proposed in this thesis are made space efficient by representing an edge occurrence with just 1 bit which can be scalable to large graphs. That is if we have 10000 nodes, the maximum number of possible edges is 100000000 for directed graph with self loops. So as I'm representing edge occurrence with 1-bit the space needed is $100000000 bits = 12.5 MegaBytes$.

5.2.2 Results

We initially compare all the six algorithms ER,ZER,PreZER, PER, PZER and PPreZER with varying the probability. The Figure 5.1 shows the comparison among all six algorithms with probability.

From the figure we can observe that the for smaller probability values i.e. $p \leq 0.5$ the algorithms ZER, PreZER, PZER and PPreZER becomes very fast because the smaller probability values offers more opportunities to skip the edges that is the skip values generated will be high. The variation of probability values will not have any effect on ER and PER algorithms because the ER and PER algorithms checks all the edges irrespective of the probability value. The parallel algorithms PER, PZER and PPreZER performance is almost equal. The ZER will perform better upto $p = 0.5$ after that the performance is decreased because of logarithm computation overhead.

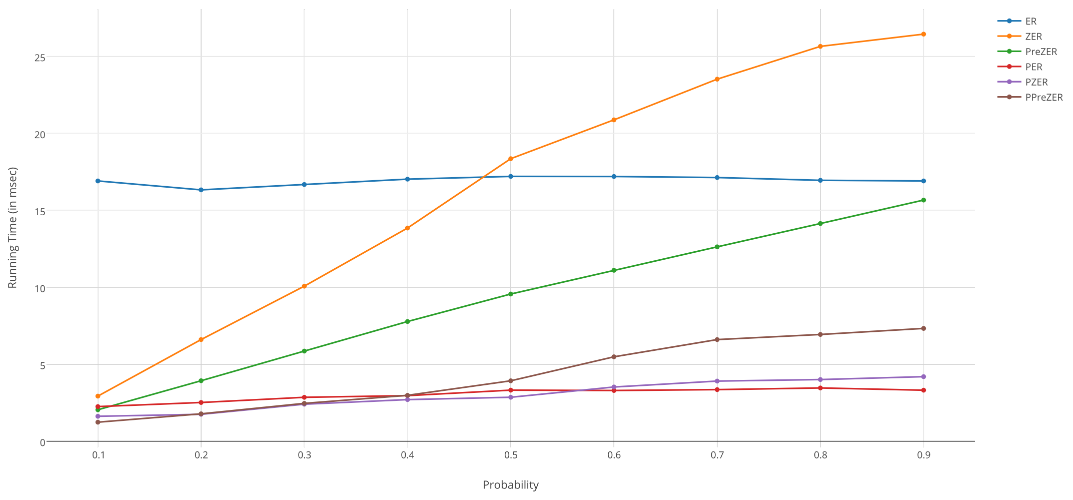


Figure 5.1: Running Times of algorithms with varying probability

Now we check how the parallel algorithms perform as the size of the graph increases. Figures 5.2 , 5.3 and 5.4 shows the performance of the algorithms for three different probability values 0.1,0.01 and 0.001 with varying the graph size i.e. increasing the number of nodes. The parallel algorithms perform a lot better than the sequential algorithms as the graph size increases. For low probability values running times of both the PZER and PreZER algorithms are almost same as there is high probability to skip the edges. For p=0.001, the skip values generated will be very high so the PreZER performs better than parallel algorithms upto 40k nodes and ZER performs better upto 60k nodes. This is because for small graph sizes the parallelization steps become overhead compare to ZER and PreZER.

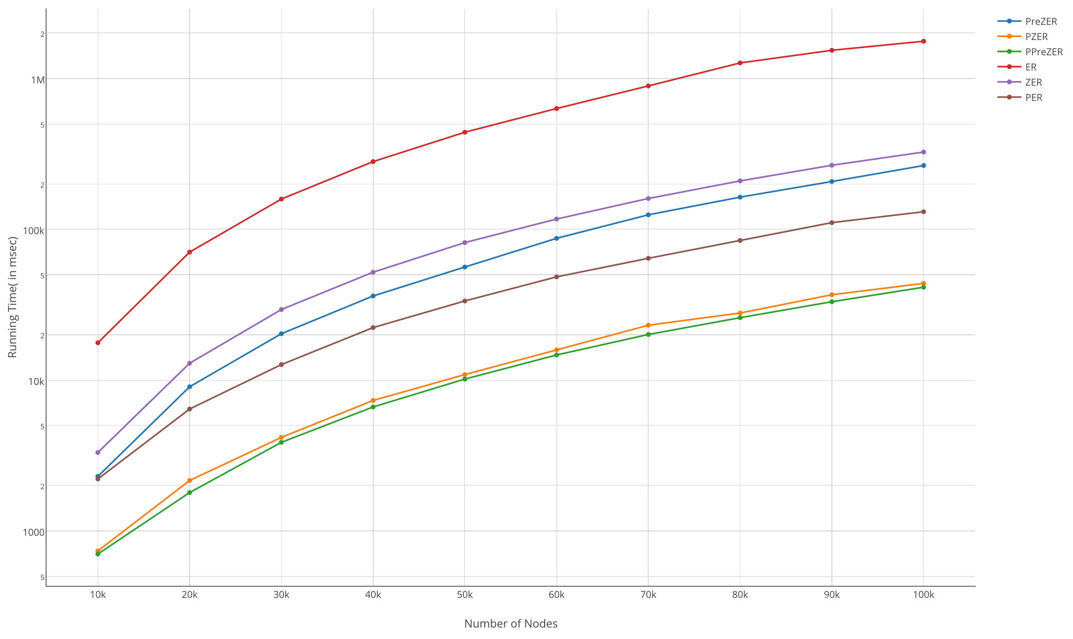


Figure 5.2: Running Times for varying Graph size with p=0.1

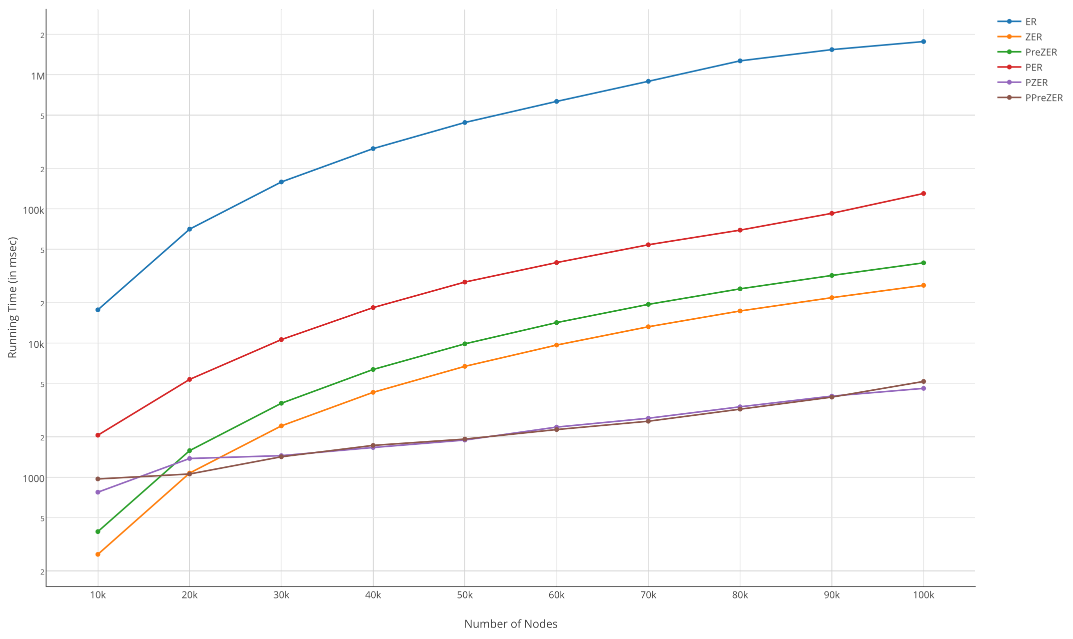


Figure 5.3: Running Times for varying Graph size with p=0.01

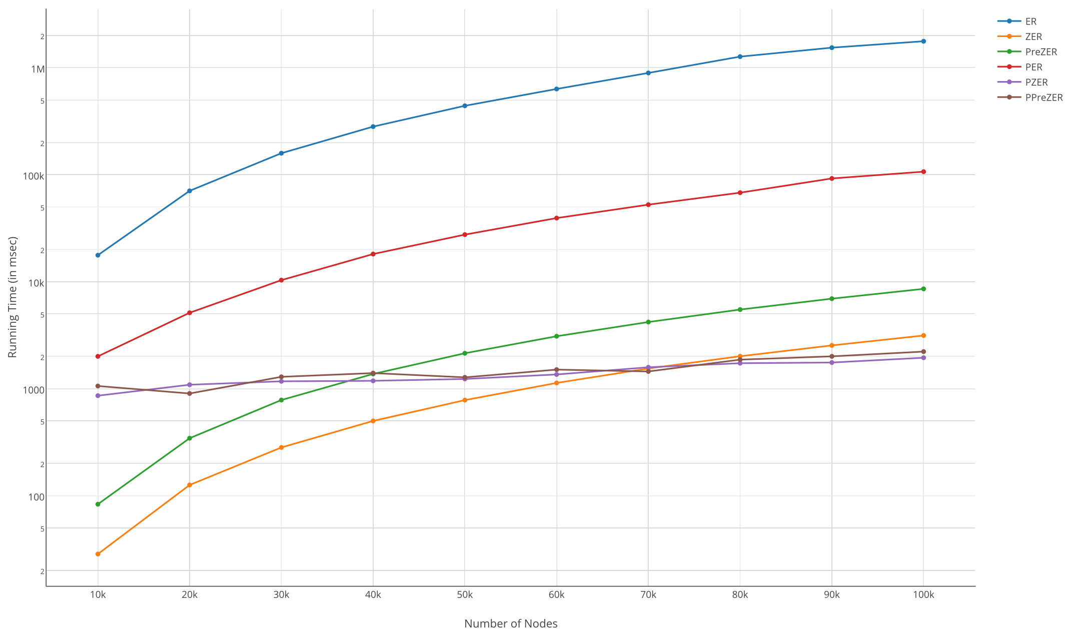


Figure 5.4: Running Times for varying Graph size with p=0.001

5.3 Watts Strogatz Model

The average speedup factor for the Parallel Watts Strogatz (PWattsStrogatz) over the sequential Watts Strogatz algorithm is 3. In watts strogatz algorithm the parallelization process is limited when compare to ER model because of undirected graph. In parallel watts strogatz algorithm we separate the edge inclusion and deletion from the parallel code and we do inclusions and deletions of edges in sequential code. This limits parallelization step. We initially compare both the algorithms WattsStrogatz and PWattsStrogatz by varying the probability with k=512 and 20k nodes. The figure 5.5 shows the behaviour of WattsStrogatz and PWattsStrogatz with the variation of probability. From the figure we can observe that the running time of both the algorithm is just slightly varies with the probability. This is because the inclusion and deletion of edges as the probability increases more number of edges will be rewired.

From the Figures 5.6, 5.7, 5.8, 5.9 and 5.10 we can observe that as the graph size i.e. number of nodes increases the parallel watts strogatz algorithm (PWattsStrogatz) is consistently faster than the sequential watts strogatz algorithm. The parallel watts strogatz algorithm beats the sequential watts strogatz algorithm by a avg factor of 3.

Figure 5.11 shows the behavior of both the algorithms WattsStrogatz and PWattsStrogatz by varying k value with fixed probability 0.5 and 20k nodes. From figure we can observe that as the k value increases the running times of both the algorithms increases. Even though the k value increases the speedup factor of parallel watts strogatz algorithm remains same.

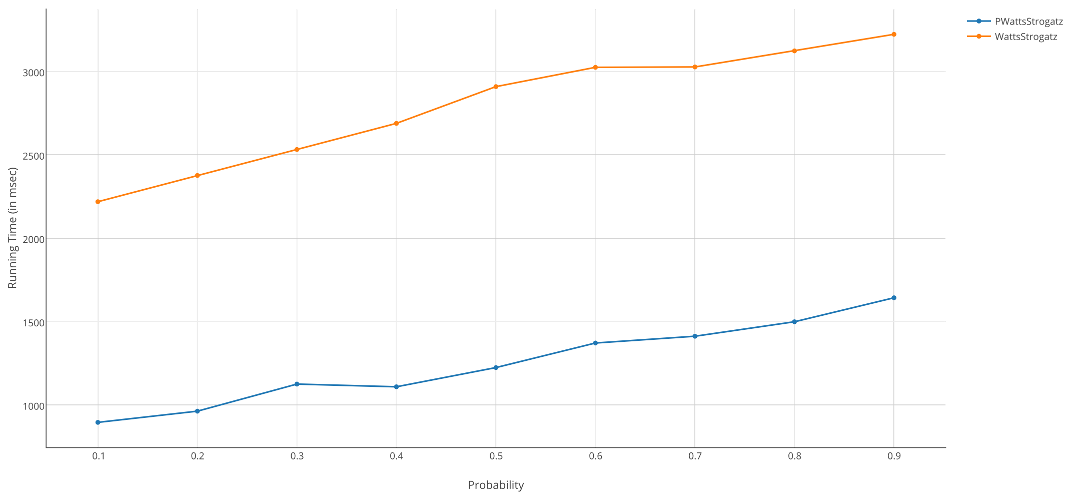


Figure 5.5: Running Times for varying Probability with k=512 and 20k nodes

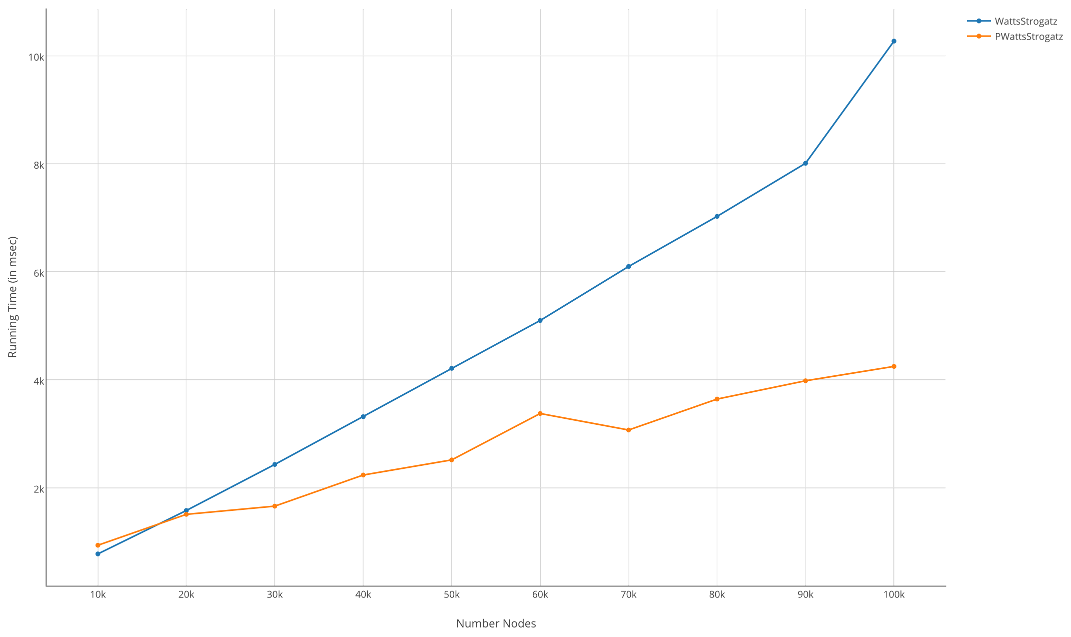


Figure 5.6: Running Times for varying Graph size with k=256

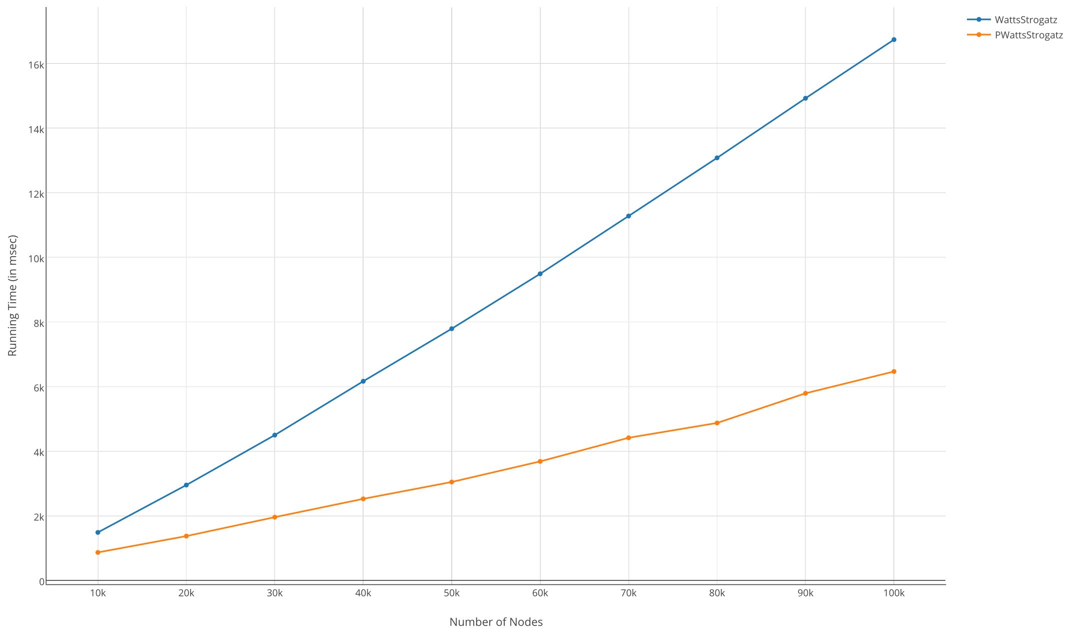


Figure 5.7: Running Times for varying Graph size with k=512

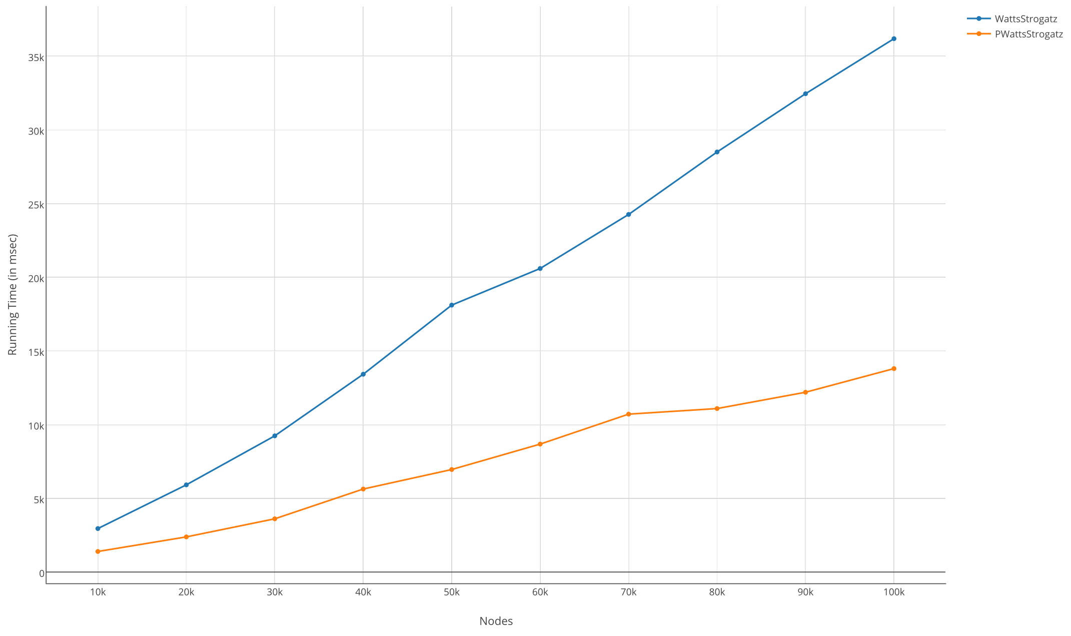


Figure 5.8: Running Times for varying Graph size with k=1024

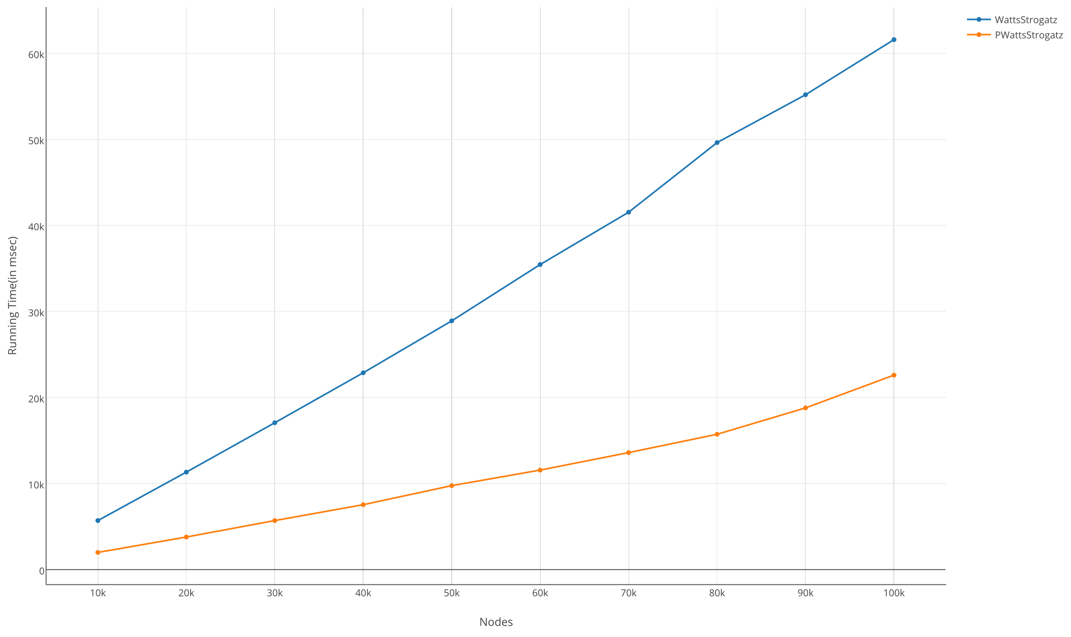


Figure 5.9: Running Times for varying Graph size with p=2048

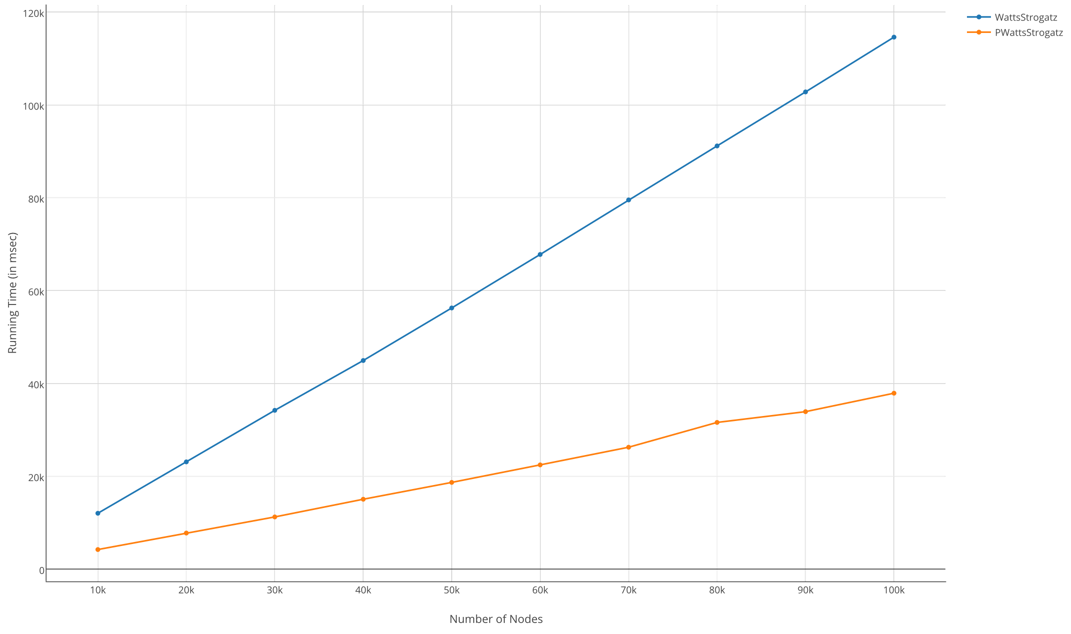


Figure 5.10: Running Times for varying Graph size with k=4096

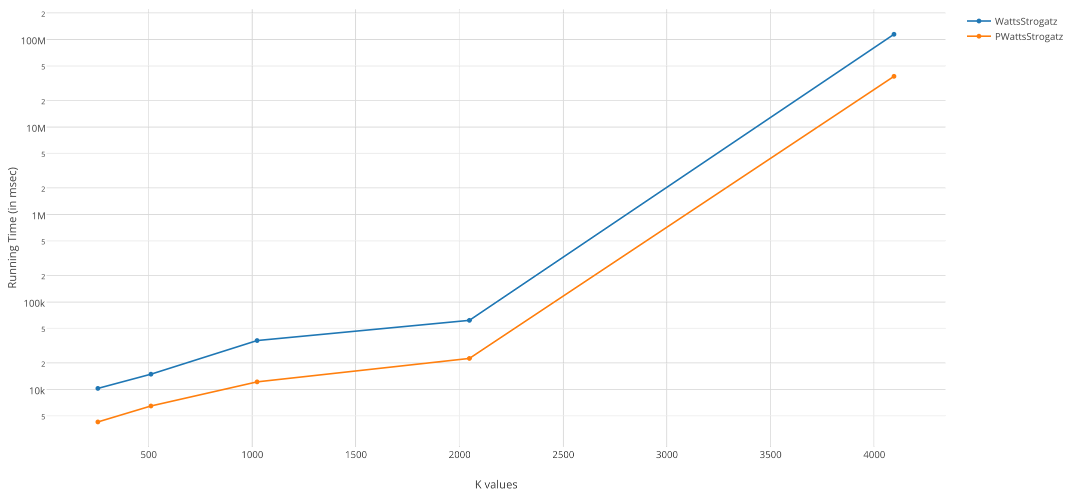


Figure 5.11: Running Times for varying k values

Chapter 6

Conclusion

Random Graphs evolved as a data representation and modeling tool for complex networks, but the previous research hasn't really focused on improving the efficiency in generating the random graphs.In this thesis work we have studied the the Erdos Renyi random graph model and we implemented the sequential and parallel algorithms given by the authors of the journal. We have further improved the performance of the parallel algorithms.

In this thesis work we proposed a novel distributed algorithm for the Erdos Renyi random graph generation model. We have studied the Watts Strogatz algorithm and implemented a novel parallel algorithm for watts strogatz algorithm which is 3 times faster than the sequential algorithm.

References

[1] S. Bressan, A. Cuzzocrea, P. Karras, X. Lu, and S. H. Nobari. An Effective and Efficient Parallel Approach for Random Graph Generation over GPUs. *J. Parallel Distrib. Comput.* 73, (2013) 303–316.

[2] M. E. J. Newman, D. J. Watts, and S. H. Strogatz. Random graph models of social networks. *Proceedings of the National Academy of Sciences of the United States of America* .

[3] S. Nobari, X. Lu, P. Karras, and S. Bressan. Fast Random Graph Generation. In Proceedings of the 14th International Conference on Extending Database Technology, EDBT/ICDT '11. ACM, New York, NY, USA, 2011 331–342.

[4] V. Batagelj and U. Brandes. Efficient generation of large random networks. *Phys. Rev. E* 71, (2005) 036,113.

[5] M. Krivelevich and B. Sudakov. Pseudo-random Graphs. In E. Gyri, G. Katona, L. Lovsz, and T. Fleiner, eds., More Sets, Graphs and Numbers, volume 15 of *Bolyai Society Mathematical Studies*, 199–262. Springer Berlin Heidelberg, 2006.

[6] E. N. Gilbert. Random Graphs. *Ann. Math. Statist.* 30, (1959) 1141–1144.

[7] P. Erdos and A. Renyi. On the Evolution of Random Graphs. In PUBLICATION OF THE MATHEMATICAL INSTITUTE OF THE HUNGARIAN ACADEMY OF SCIENCES. 1960 17–61.

[8] Y. Xia, C. Tse, F. Lau, W. M. Tam, and X. Shan. Traffic congestion analysis in complex networks. In Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium on. 2006 4 pp.–.

[9] Z. Saul and V. Filkov. Methods for Random Modularization of Biological Networks. In BioInformatics and BioEngineering, 2006. BIBE 2006. Sixth IEEE Symposium on. 2006 285–288.

[10] J. H. Fowler, C. T. Dawes, and N. A. Christakis. Model of genetic variation in human social networks. *Proceedings of the National Academy of Sciences* 106, (2009) 1720–1724.

[11] D. McDonald, L. Waterbury, R. Knight, and M. Betterton. Activating and inhibiting connections in biological network dynamics. *Biology Direct* 3, (2008) 49.

[12] M. Dyer and A. Frieze. Randomly colouring graphs with lower bounds on girth and maximum degree. In Foundations of Computer Science, 2001. Proceedings. 42nd IEEE Symposium on. 2001 579–587.

[13] K. Oikonomou and I. Stavrakakis. Performance Analysis of Probabilistic Flooding Using Random Graphs. In World of Wireless, Mobile and Multimedia Networks, 2007. WoWMoM 2007. IEEE International Symposium on a. 2007 1–6.

[14] A. Ganesh, L. Massoulie, and D. Towsley. The effect of network topology on the spread of epidemics. In INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE, volume 2. 2005 1455–1466 vol. 2.

[15] J. Diaz, J. Petit, and M. Serna. A random graph model for optical networks of sensors. *Mobile Computing, IEEE Transactions on* 2, (2003) 186–196.

[16] nVidia. CUDA Zone: Toolkit & SDK. *https://developer.nvidia.com/cuda-toolkit* .

[17] D. J. Watts and S. H. Strogatz. Collective dynamics of /'small-world/' networks. *Nature* 393, (1998) 440–442.

[18] X. Fang and J. Zhan. Task-Oriented Social Ego Network Generation via Dynamic Collaborator Selection. In Privacy, Security, Risk and Trust (PASSAT), 2012 International Conference on and 2012 International Confernece on Social Computing (SocialCom). 2012 41–50.

[19] Apache. Hadoop Map Reduce. *https://hadoop.apache.org/* .