M.Tech (Computer Science) Dissertation Series

GPU Accelerated Analysis and Simulation of Large Scale Networks

A dissertation submitted towards partial fulfillment of the requirements for the award of M.Tech(Computer Science) degree of Indian Statistical Institute

by

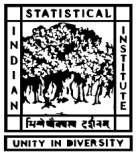
Chirag Gupta

Roll No. CS1317

under the supervision of

Prof. Nabanita Das

Advanced Computing and Microelectronics Unit



Indian Statistical Institute

203, Barrackpore Trunk Road

Kolkata 700108

CERTIFICATE

This is to certify that the thesis titled **GPU Accelerated Analysis and Simulation of Large Scale Networks** submitted by **Chirag Gupta** towards partial fulfillment for the award of the degree of Master of Technology is a bonafide record of work carried out by him under my supervision. The thesis has fulfilled all the requirements as per the regulations of this Institute and, in my opinion, has reached the standard needed for submission. The results embodied in this thesis have not been submitted to any other university for the award of any degree or diploma.

(Prof. Nabanita Das)
Advanced Computing and Microelectronics Unit
Indian Statistical Institute, Kolkata

M.Tech (Computer Science) Dissertation Series

GPU Accelerated Analysis and Simulation of Large Scale Networks

A dissertation submitted towards partial fulfillment of the requirements for the award of M.Tech(Computer Science) degree of Indian Statistical Institute

by

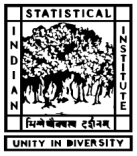
Chirag Gupta

Roll No. CS1317

under the supervision of

Prof. Nabanita Das

Advanced Computing and Microelectronics Unit



Indian Statistical Institute

203, Barrackpore Trunk Road

Kolkata 700108

ACKNOWLEDGEMENT

First of all, I wish to express my sincere gratitude to **Prof. Nabanita Das, Advanced Computing and Microelectronics Unit, ISI Kolkata** for suggesting me this problem in the first place and her valuable guidance and inspiration throughout my work.

I am very grateful to all my classmates and Dibakar Saha, ACMU for their support and motivation to complete this dissertation.

(Chirag Gupta)
M.Tech(CS)
Roll No. CS1317
Indian Statistical Institute
Kolkata, India

ABSTRACT

In the world of digital interaction, online social networks have become an inevitable part of our daily life. It becomes a strong and reliable medium to share individual views and other media to the rest of the world. In this report, we attempt to analyze, why some of the happenings in the world shared on the social network covers entire network and becomes popular or trending, while the others just die out over time or remains locally popular? Simulating or analyzing these in live environment poses big challenges due to their large size of underlying network and fast paced data generation. It is computationally intensive for CPU's to manage such massive operations which is out of question as far as real-time results are desired. An efficient competitive diffusion scheme is proposed here to optimize for parallel execution in CUDA architecture. Extensive simulation study shows the viral and non-viral diffusion patterns and their associations with underlying graph structure.

Keywords: Graphics Processing Unit [GPU], Heterogeneous Computing, Social networks, Large-Scale Networks, Parallel Computing, CUDA Architecture

Contents

Chapter 1

Introduction

In recent years, with the advent of information technology for human interactions, our interaction with people, machine e.g. appliances, sensors and devices have increased to a large extent. If we represent these interactions in the form of a network, due to various types of interacting bodies and their interactions, we end up in a network with millions of nodes and billions of edges, which we call as large scale networks. Social network, Phone call network etc. are perfect examples of such large scale networks. Recent years have witnessed the explosive growth of online social platforms, social networks like twitter, facebook have reached tens of millions of users with billion activities per day. As the interaction data is available online and accessible to us, many researchers have shown their interest in studying how interactions among people can be categorized? How a topic or a information diffuses inside these networks? Studying these phenomena are typically computationally intensive while large size of the network makes this analysis even harder. Most of these behaviors related to network graphs are temporal, thus a fast computing model is desired for such analysis.

1.1 Large scale network analysis

It is evident that now a days, most of the human interactions happen via information technology. A network formed with these interactions is massively large and spread across geographical areas. Scientists and researchers has been working for a long time to solve different problems associated with these large graphs. Most of these are generic graph problems but have become practically infeasible because of the associated graph size. Thus, An efficient solution to these problems on large graphs, in terms of memory and execution time is awaited from a long time. Parallel algorithms for graph problems and their implementations on different parallel frameworks come to a rescue in this situation. In this section, we will discuss some of graph problems and their efficient scalable solutions using multi-core parallel architectures.

Here, we first focus on the classical graph traversal problem i.e. Breadth First Search (BFS). For a given graph $G(V, E)$, BFS algorithm is already of linear complexity *i.e.* $O(V + E)$ but it still takes significant amount of time on a graph with hundreds of millions or billions of nodes, and associated edges. To achieve real-time results researchers

have now looking forward towards multi-core or parallel architectures. Bader et. al.[1] proposed a parallel algorithm for efficient execution of BFS on a special purpose Cray MTA-2 parallel computer. This algorithm forces a synchronization at each level in the graph and performs BFS on graph nodes in parallel. In another work on parallel BFS algorithms by Harish et al. [2], authors propose a scheme for general purpose CUDA architecture to solve the same problem. CUDA architecture based hardware are easily available and cost is much less than special purpose parallel computers like Cray MTA-2.

Another popular graph problem of finding Shortest Path is approached by Harish [2]. In this contribution, authors propose a parallel algorithm with efficient implementation scheme on general purpose CUDA framework. Author has proposed parallel algorithm for both Single Source Shortest Path (SSSP) problem and All Pair Shortest Path (APSP) problem. A modified version of Floyd Warshall algorithm is proposed to solve aforementioned problems in parallel.

Another challenging problem is Community Detection in large graphs. In a large network, study of these communities explains a lot of behavioral aspects of the underlying graph and may help in identifying the interaction patterns and predicting the future too. Community detection has been a popular research topic since long time. In a seminal work (2007) by Raghavan et al.[3] had shown a near linear time algorithm to detect community structures in large scale networks and developed an algorithm based on label propagation, which cleverly detects the communities in network graph.

A graph is initialized with the labels randomly and these labels are allowed to propagate via neighbors. Intuition to this technique is as follows- As the labels propagate, densely connected components of graph quickly reach a consensus on a unique label. Thus, many consensus are created throughout the network, they continue to expand outwards until it is possible to do so. At the end of the propagation process, nodes having the same label are grouped together as one community. Although this provides a scalable solution for large graphs, but real-time solution is still on its way. Recently, in 2014, Chavarría-Miranda et al. [4] propose a scalable graph community detection algorithm on Tilera-360 Many-core Architecture.

In the Graph analysis, the measurement of betweenness centrality measure has always been an intriguing problem due to its variety of application i.e. structural analysis of knowledge networks, power grid contingency analysis, quantifying importance in social networks, analysis of decision/action network and also for finding the best store locations in cities, there is a quest of better techniques and algorithms for its computation. Betweenness centrality measure is known to be an expensive kernel in graph mining. There are two ways of GPU implementation for betweenness centrality computation. Conventional algorithm follows node-based parallelism which takes less memory and performs well for almost regular graphs having less degree variance. However, Scale free networks which follow power law distribution (best available approximation of current social networks) have higher degree variance. An edge-based approach on GPU improves GPU throughput with better load balancing for scale free networks. Sariyüce et al. [5] propose a scheme for efficient implementation on GPUs which reduces memory usage and creates a virtualization of high degree vertices to solve the imbalance problem in vertex-based approach and keep the memory usage lower than edge-based approach.

1.2 Information diffusion process

In the world of digital interaction, online social network becomes an inevitable part of our daily life. It has become a strong and reliable medium to share individual views and other media to the rest of the world. Some of the events happened in the world and shared on the social network and cover entire network and become popular or trending, while others just die out over time or remain locally popular only.

Online social networks allow hundreds of millions of Internet users worldwide to produce and consume content. They provide access to a vast source of information on an unprecedented scale. Online social networks play a major role in the diffusion of information by increasing the spread of novel information and diverse viewpoints [6]. Authors proved social networks to be very powerful in many situations, like Facebook during the 2010 Arab spring [7] or Twitter during the 2008 U.S. presidential elections[8]. Given the impact of online social networks on society, the recent focus is on extracting valuable information from this huge amount of data. Events, issues, interests etc. happen and evolve very quickly in social networks and their capture, understanding, visualization, and prediction are becoming critical expectations from both end users and researchers.

Most of earlier works try to model the information flow in networks that plays a crucial role in management of misinformation propagation too. A recent study by Fang et al.[9] shows spread of the Ebola virus news on twitter. It started from Guinea, Liberia and Sierra Leone, which become viral and sustained in the network for sufficient period of time. Although, Ebola is not a new disease, this time its affect in west Africa was severe and became a hit news throughout the world. Their work explains several dynamics regarding this particular information flow on twitter. Mark Twain is credited with the aphorism that a lie can travel half-way around the world while the truth is putting on its shoes. Thus misinformation study too is a crucial issue in social network scenario.

Study of social networks is motivated by the fact that understanding the dynamics of these networks may help in better prediction of following events (e.g. analyzing revolutionary waves), solving issues (e.g. preventing terrorist attacks, anticipating natural hazards, epidemic analysis), optimizing business performance (e.g. optimizing social marketing campaigns) etc. In recent years, researchers have proposed a variety of techniques and models to capture information diffusion in online social networks[10], analyze it, extract knowledge from it and predict it. Information diffusion is a vast research domain and has attracted research interests from many fields, such as physics, biology (i.e. study of spread of disease among a population) etc.

We mainly focus here on the following cases of information diffusion in online social networks, that raises the following questions :

- To find the most diffused information or topic which signifies popularity of the topic.
- To find the cause of diffusion in the network and its lifetime in the future.
- To find the members in the network who play an important role in diffusion.

Fig. 1.1 shows the diffusion process of a topic in social network. At *Time=0* few nodes from social network choose to talk about one topic(a.k.a information). We call these nodes

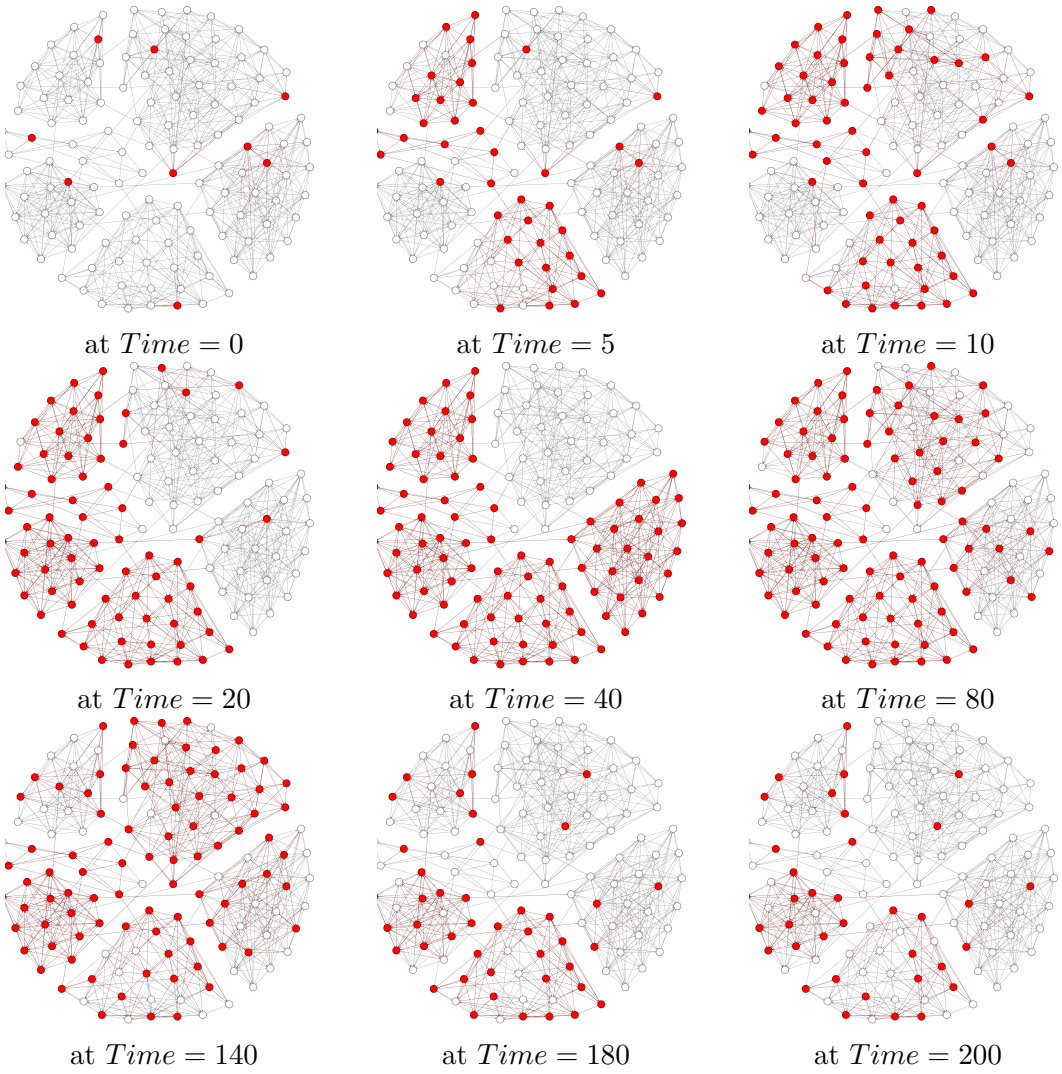


Figure 1.1: Information diffusion of a topic over time

as seed nodes. Before this event, there were no instance of this topic in the network. Each image above shows the status of the topic instances in social network at different time instants. Due to seed nodes activities, this information reaches to their neighbors and they may choose to talk about this topic too. Now social network have more number of instances of same topic in it. Similarly, over the time this process goes on and more number of nodes get introduced to this topic.

Here, the interesting fact is that if this case happens with every topic, all topic must have reached to most of the nodes in the network, But this is not a real scenario because there is an inherent competition among the topics present in the network. Propagation of a topic depends on user's choice, if he chooses to talk on topic A, A is forwarded, with no propagation of the rest of the topics at that time instant. This behavior depicts a competition in topic selection by the user. It may depend on user's behavioral, cultural, geographical and many other types of interests. Thus, not all the topics span across the network, some may remain prevalent in the network and become popular, while others die quickly. This is a temporal phenomenon, a topic will not be prevalent in the network for all the time, after some time, users of the network loose interest in this topic or some

other may defeat this topic in competition in later stage of time and topic dies out.

Various theoretical models are proposed so far to predict the behavior of social networks. For validation extensive simulation studies are carried out on high performance computers or computing clusters. This infrastructure is very expensive and also not energy-efficient. In our study, since in social networks, nodes are, in general, independent in their activities over time, there is a scope of node-level parallelism. Therefore, Parallel computing platform CUDA can serve as a good framework. NVIDIA GPU's support CUDA frameworks and provide multi-core execution platform for high throughput applications. It is capable of running million of threads concurrently and provides adequate memory to accommodate sufficient large graphs, however smart representation of graph and memory utilization is required to manage them efficiently and to achieve significant speed up. The challenges are mainly due to interdependency among threads and data storage in global or local memories. However, CUDA framework based hardware are much cheaper and affordable than conventional high performance parallel machines and allows horizontal scalability as the data size grows. In this work, we propose different models for information diffusion in social networks and explore how those can be implemented on CUDA architecture for extensive simulation study on real networks to achieve speed up that may help us to predict the network behavior in real time.

Chapter 2

CUDA Architecture

2.1 Overview

`CUDA`, which stands for `Compute Unified Device Architecture`, is a parallel computing platform and `application programming interface (API)` model created by NVIDIA. It allows software developers to use a CUDA-enabled `graphics processing unit (GPU)` for general purpose processing an approach known as GP-GPU. The CUDA platform is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements.

The CUDA platform (As shown in fig. 2.1) is designed to work with programming languages such as C, C++ and Fortran. This accessibility makes it easier for specialists in parallel programming to utilize GPU resources, as opposed to previous API solutions like Direct3D and OpenGL, which required advanced skills in graphics programming.

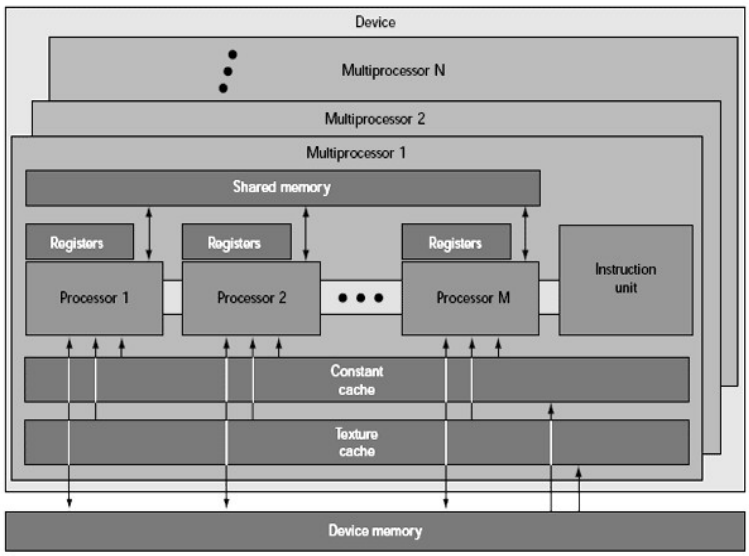


Figure 2.1: CUDA Architecture

CUDA provides both a low level API and a higher level API. The initial CUDA SDK was made public on 15 February 2007, for Microsoft Windows and Linux. Mac OS X support was later added in version 2.0, which supersedes the beta released February 14, 2008. CUDA works with all NVIDIA GPUs from the G8x series onwards, including GeForce, Quadro and the Tesla line. CUDA is compatible with most standard operating systems.

2.1.1 NVIDIA GPU

NVIDIA developed CUDA platform to increase graphic intensive gaming performance. Later on, it becomes very populate among high-performance computing enthusiasts. Currently only NVIDIA GPU's support CUDA architecture. It comes in several series while GeForce is meant for gaming purposes, Quadro and tesla series are meant for computing purposes.

For this dissertation purposes NVIDIA Tesla C2075 graphic card is used. It is based on the next-generation CUDA architecture codenamed `Fermi`. It consists of 14 parallel compute units each sporting 32 CUDA cores clocked at 1.15Ghz, 448 cores in total. C2075 model have 6GB memory clocked at 1.5Ghz in total, which is sufficient for several applications. Tesla series provide wide memory interface of 368-bit with high transfer rate. A typical `Fermi` architecture based core is shown in the fig. 2.2.

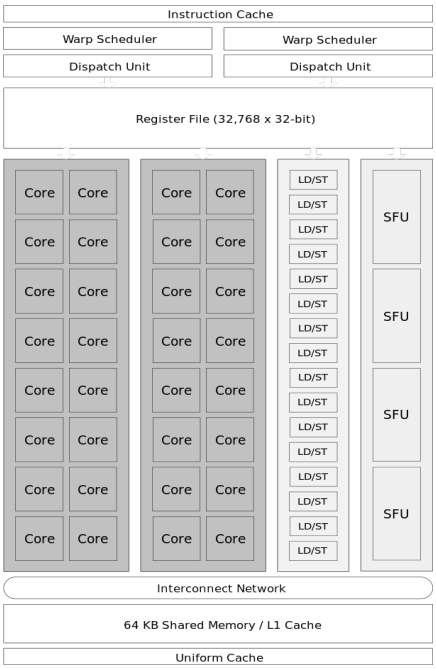


Figure 2.2: Fermi Architecture based GPU

CUDA has several advantages over traditional general-purpose computation on GPUs (GPGPU) using graphics APIs:

1. `Scattered reads` - code can read from arbitrary addresses in memory
2. `Unified virtual memory` (CUDA 4.0 and above)

3. `Unified memory` (CUDA 6.0 and above)
4. `Shared memory` - CUDA exposes a fast shared memory region that can be shared amongst threads. This can be used as a user-managed cache, enabling higher bandwidth than is possible using texture lookups.
5. `Data transfer` - Faster downloads and readbacks to and from the GPU
6. `Operations support` - Full support for integer and bitwise operations, including integer texture lookups

2.1.2 Architectural difference between CPU and GPU

Since 2006 GPUs have led the floating point performance. While performance of microprocessor is not so good in floating point, GPUs have continued to improve relentlessly. As per 2006, floating point calculation throughput between many-core GPUs and multi-core CPU is near about 10. Which is quite significant in the application where time constraint is tight. Recent development in GPU architecture even improved the performance to a new level, which led to lot of compute intensive software to execute their expensive codes in GPU to get speedup.

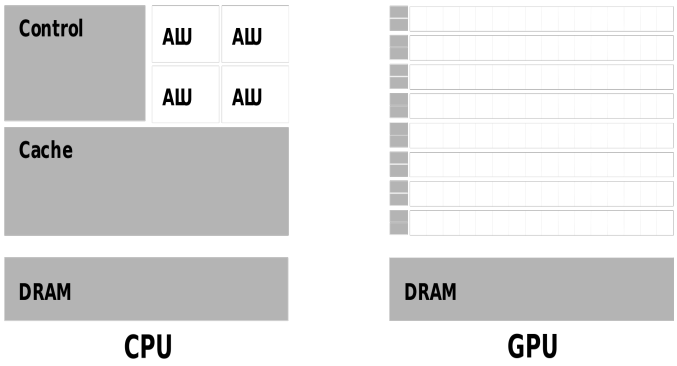


Figure 2.3: Difference between CPU and GPU architecture

This large difference in performance of a CPU and GPU comes because of their fundamental design philosophies. CPU are general purpose while GPU are optimized for numeric and floating point operations. Execution in CPU is sequential and it tries to perform operation as fast as it can, while GPU follows parallel programming paradigm, However, it does not contain a powerful core, but it create separate thread for each data, which in tern gives a higher throughput, hence speedup. Smart applications are hybrid in nature, it utilizes CPU for regular operations, while involves GPU for parallel extensive code execution. CUDA platform provides a seamless bridge in between GPU and CPU. It provides several pre-built function to support memory transfer, its initialization, launch and managing parameters, which we will discuss later in coming chapters.

2.2 CUDA threads

Fine-grained, data-parallel threads are the fundamental means of parallel execution in CUDA. These threads are invoked by a special type of function named `Kernel` function. Launching a CUDA kernel creates a grid of threads that all execute the kernel function. That is, the kernel function specifies the statements that are executed by each individual thread created when the kernel is launched at run-time.

2.2.1 Thread organization

Since all threads in a grid execute the same kernel function, they rely on unique coordinates to distinguish themselves from each other and to identify the appropriate portion of the data to process. These threads are organized into a two-level hierarchy using unique coordinates, called *blockId* and *threadId*, assigned to them by the CUDA runtime system. The *blockId* and *threadId* appear as built-in variables that are initialized by the run-time system and can be accessed within the kernel functions. When a thread executes the kernel function, references to the blockId and *threadId* variables return the appropriate values that form coordinates of the thread. At the top level of the hierarchy, a grid is organized as a two dimensional array of blocks. The number of blocks in each dimension is specified by the first special parameter given at the kernel launch.

In CUDA, there are special parameters that specify the number of blocks in each dimension as a struct variable *gridDim*, with *gridDim.x* specifying the number of blocks in the x dimension and *gridDim.y* the y dimension. The values of *gridDim.x* and *gridDim.y* can be anywhere between 1 and $65,536$. The values of *gridDim.x* and *gridDim.y* can be supplied by run-time variables at kernel launch time. Once a kernel is launched, its dimensions cannot change in the current CUDA run-time implementation. All threads in a block share the same *blockId* values. The *blockId.x* value ranges between 0 and *gridDim.x*-1 and the *blockId.y* value between 0 and *gridDim.y*-1.

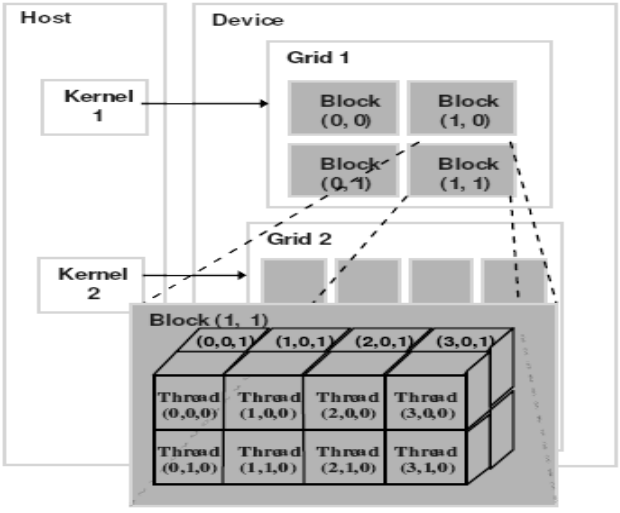


Figure 2.4: Thread Organization inside kernel (Thread representation in Grids and Blocks)

A grid consists of several blocks. For example a grid launched at kernel with *gridDim.x* = 2 and *gridDim.y* = 2, actually contains four blocks. These blocks are organized into a 2×2 array. Each block in the array is labeled with (*blockId.x*, *blockId.y*). For example, Block$(1, 0)$ has its *blockId.x*= 1 and *blockId.y*= 0 As shown in fig. 2.4.

Inside the block, at the bottom level of CUDA architecture hierarchy, threads resides. all blocks of a grid are organized into a three-dimensional array of threads. As grid and blocks, individual threads are identified by a special struct variable *threadId*. Each *threadId* consists of three components: the x coordinate *threadId.x*, the y coordinate *threadId.y*, and the z coordinate *threadId.z*. The number of threads in each dimension of a block is specified by the second special parameter given at the kernel launch as *blockDim* insider kernel code.

A typical kernel launch is as follows:

```
dim3 dimBlock(16, 16, 1);
dim3 dimGrid(100, 1, 1);
KernelFunction<<<dimGrid, dimBlock>>>(...);
```

It will instantiate a 1D grid of 100 blocks where each block has 16×16 threads. A total of 25600 threads will be instantiated together.

2.2.2 Transparent scalability

CUDA allows threads in the same block to coordinate their activities using a barrier synchronization function syncthreads(). When a kernel function calls syncthreads(), all threads in a block will be held at the calling location until everyone else in the block reaches the location. This ensures that all threads in a block have completed a phase of their execution of the kernel before they all move on to the next phase.

The ability of synchronizing with each other also imposes execution constraints on threads within a block. These threads should execute in close time proximity with each other to avoid excessively long waiting times. CUDA run-time systems satisfy this constraint by assigning execution resources to all threads in a block as a unit. That is, when a thread is of a block is assigned to an execution resource, all other threads in the same block are also assigned to the same resource. This ensures the time proximity of all threads in a block and prevents excessive waiting time during barrier synchronization.

This leads us to a major trade-off in the design of CUDA barrier synchronization. By not allowing threads in different blocks to perform barrier synchronization with each other, CUDA run-time system does not need to deal with any constraint while executing different blocks. That is, blocks can execute in any order relative to each other since none of them need to wait for each other. This flexibility enables scalable implementations. Lack of synchronization across blocks allows block to execute in any order, which leads to scalability as the block which have all resources can execute without worrying about pre-specified order.

2.2.3 Thread scheduling

CUDA threads scheduling is done by CUDA thread scheduler, which depends on device design architecture i.e. Fermi, Keplar. Programmer setup the kernel parameters and accordingly threads are spawned in device. Each block moves into one SM (Streaming Multi-processor). While assigning blocks thread scheduler tries to maintain load balance by enforcing strict round-robin order with modulo smid. Blocks make warps of assigned threads up to 32 thread each. The shape of a grid (1-D or 2-D) influences the order in which thread blocks are picked. For 1-D grids, thread blocks are picked in increasing order of thread block ID. For 2-D grids, thread blocks are picked in a pattern resembling a space-filling Hilbert curve aimed perhaps at preserving 2-D locality.

Once each thread is assigned to individual CUDA core, All threads start execute kernel code. Thread identified themselves by their Ids, which is calculated using block dimension and grid dimension specified in kernel launch.

ThreadId calculation depends on kernel launch configuration and can be calculated as follows:

1. `1-D grid of 1-D blocks:`

   int threadId = blockIdx.x *blockDim.x + threadIdx.x

2. `1-D grid of 2-D blocks:`

   int threadId = blockIdx.x * blockDim.x * blockDim.y + threadIdx.y * blockDim.x + threadIdx.x;

3. `2-D grid of 1-D blocks:`

   int blockId = blockIdx.y * gridDim.x + blockIdx.x;
   int threadId = blockId * blockDim.x + threadIdx.x;

4. `2-D grid of 2-D blocks:`

   int blockId = blockIdx.x + blockIdx.y * gridDim.x;
   int threadId = blockId * (blockDim.x * blockDim.y) + (threadIdx.y * blockDim.x) + threadIdx.x;.

3-D grids have became available with CUDA 4.0 onwards, which utilizes third dimension z in blockDim, blockIdx and threadIdx data structures.

2.3 CUDA memory model

a CUDA kernel function which can be invoked by a massive number of threads. The data to be processed by these threads are first transferred from the host memory to the device global memory. The threads then access their portion of the data from the global memory using block and thread IDs. We have also learned the more details of the assignment and scheduling of threads for execution. Although this is a very good start, these simple CUDA kernels will likely achieve only a small fraction of the potential speed

of the underlying hardware. This is due to the fact that global memory, which is typically implemented with Dynamic Random Access Memory (DRAM), tends to have long access latencies (hundreds of clock cycles) and limited access bandwidth. While having many threads available for execution can theoretically tolerate long memory access latencies, one can easily run into a situation where traffic congestion in the global memory access paths prevents all but very few threads from making progress, thus rendering multiple Streaming Multiprocessors idle. In order to circumvent such congestion, CUDA provides a plethora of additional types of memories that can filter out a majority of data requests to the global memory.

2.3.1 Memory classification

Like CPU, Graphics Processing Unit (GPU) also maintain hierarchy to provide better fast and reliable memory management. Total memory of GPU is classified in 5 classes:

1. `Registers`: These are the fastest memory available in GPU. *Register Memory* is only accessible to threads. Number of the registers are limited and it depends on GPU series. A variable stored in register memory is only available during thread's life-time, as threads finished its work in kernel, value of the register goes off.

2. `Shared Memory`: Since register memory is very less, to accommodate more data, a *Shared Memory* is available in GPU. Shared memory is again a fast access memory. It is only accessible to one block. A specified block of memory is available to each CUDA block, which depends on the GPU series. All the threads in a particular block can read and write the data in this memory while this data won't be accessible to other blocks. This memory have parallel access, so multiple threads can access the memory concurrently. A variable defined using `__shared__` will automatically get stored in this memory if enough space available.

3. `Global Memory`: To support large amount of data and access to all threads executing in a kernel, *Global Memory* is provided in the GPU. This memory is typically implemented using DRAM technology. Access to this memory have higher latency than aforementioned memory types. It plays a crucial role in CUDA programming, as different block in a grid have access to this memory. Depending on the graphics card series, it may vary from 256MB to 12GB. Tesla C2075 sports a total memory of 6GB. Global memory have a finite access bandwidth, in high load condition it may face traffic congestion problem in memory access. Host CPU can access global memory and can perform read/write using CUDA APIs, while registers and shared memory is inaccessible to Host CPU.

4. `Constant Memory`: This is a special type of memory, which is read-only. It is particularly used to store constant used in the CUDA programs and other lookups mechanisms. This memory is typically faster and provides high bandwidth, if multiple threads access to the same location. This memory is again accessible from Host CPU, which can write content before kernel launch. A variable defined by `__constant__` will be automatically stored in this memory.

5. `Texture Memory`: This is again a special type of memory used for texture. It is mostly used particularly in graphics application and games.

A typical organization of memory in GPU is shown in the fig. 2.5.

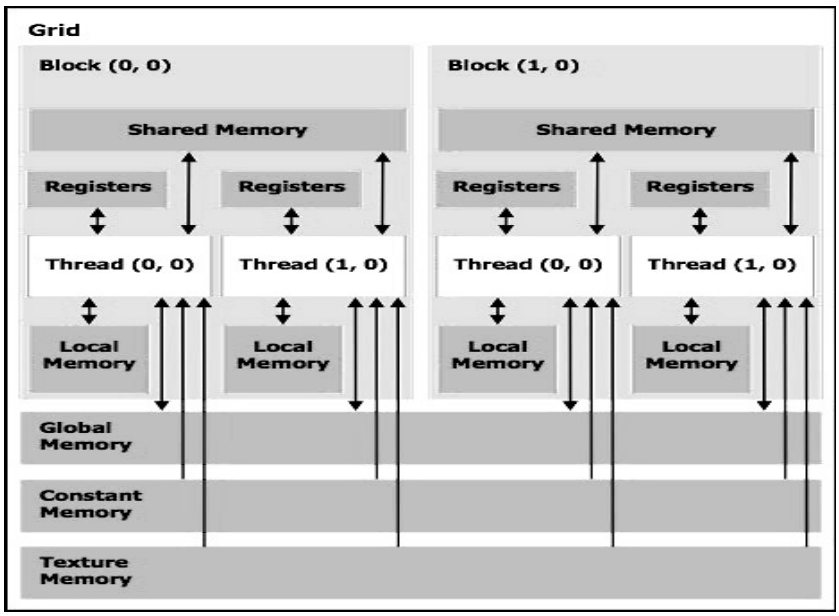


Figure 2.5: GPU Memory Hierarchy and Access Control

2.4 CUDA programming model

NVIDIA GPUs are programmed as a sequence of kernels. CUDA offers a data parallel programming model that is supported on NVIDIA GPUs. In this model, the host program launches a sequence of kernels. Threads are grouped into blocks, and blocks are grouped into a grid. Each thread has a unique local index in its block, and each block has a unique index in the grid. Kernels can use these indices to compute array subscripts, for instance.

2.4.1 Data parallelism

CUDA program are multi-threaded programming model which aims for Data parallelism. In lot of applications there are subsections of the code in which one particular instruction of block need to be executed lot of time and all of them are independent to each other, even the order of execution of these instruction doesn't matters. Such sections can be parallelized and CUDA framework can provide facility to execute all of them in parallel. For example, Let P be the resultant matrix obtained after multiplication of two matrices M and N. Each element of P is generated by performing a product between corresponding row and column, which is independent of all other elements. Therefore, computation of each element can be done in an individual thread.

2.4.2 CUDA program structure

A CUDA program is structured in two parts.

`Host Program`: *Host Program* is the driver program which execute in Host CPU. All the interaction to GPU must be done through this program. The CUDA platform is accessible to software developers through CUDA-accelerated libraries, compiler directives

such as OpenACC, and extensions to industry-standard programming languages including C, C++ and Fortran. C/C++ programmers use CUDA C/C++, compiled with *nvcc* NVIDIA's LLVM-based C/C++ compiler.[4] Fortran programmers can use *CUDA Fortran*, compiled with the PGI CUDA Fortran compiler from The Portland Group. Third party wrappers are also available for Python, Perl, Fortran, Java, Ruby, Lua, Haskell, R, MATLAB and native support in Mathematica.

Host program allocates memory for program execution in GPU and does the all memory transfer to and from GPU. CUDA provides all pre-built functions for support these seamlessly. Host program invokes kernel with a suitable configuration and appropriate function parameters.

`Device Program`: *Device Program* (also known as *Kernel*) is the core part in CUDA programming, as this is the only part which get executed in GPU environment. As per the configuration supplied during kernel launch, it spawns appropriate number of threads. Each thread is identifiable using its Id, which is calculated in kernel as mentioned in earlier sections. Once the Id is calculated, each threads, knows its block and the grid, CUDA framework assign one block to one *SM (Streaming Multi-processor)* and schedule the blocks accordingly. Execution of these threads are independent and can be synchronized at block level using *__syncthreads()* command in the code. It stops the execution of the current threads until all threads in that block execute the code above it.

2.4.3 Memory device and data transfer

In CUDA architecture, Host and Device both have their separate memory space. In order to execute a kernel on the device, Host program need to allocate and initialize memory in device memory space. Once the memory is initialized, kernel is launched from host which will access allocated memory, does the computation and store the results in specified memory location in GPU memory space. Once kernel execution is completed, Host reads the results from GPU memory space and does the rest of the computing there on.

The CUDA memory model is supported by set of API fucntion that can be called by host program. Some of memory related functions are as follows-

1. `cudaMalloc`*(void **devPtr, size_t size)* : allocates memory chunk of specified size in device memory space with devPtr as a reference.
2. `cudaFree`*(void *devPtr)*: frees memory allocated with the reference devPtr in device memory space.
3. `cudaMemcpy`*(void *dst, const void *src, size_t count, enum cudaMemcpyKind kind)* : it copies data between host memory to device memory. Direction of transfer is specified by the last parameter of this function. It takes two values *cudaMemcpyHostToDevice* and *cudaMemcpyDeviceToHost*. First one moves data from source in host memory to destination in device memory, while the second one moves data from source in device memory to destination in host memory.
4. `cudaMemset`*(void *devPtr, int value, size_t count)*: initializes the memory assigned at devPtr with the value specified.

Chapter 3

Analysis and Simulation of Large Scale Networks

Online Social networks have recently become an effective and innovative channel for spreading information among hundreds of millions of end-users.

3.1 Problem statement

This section explores one key question: *For a given social network graph, how does a piece of information travels over time and how it becomes so popular in network a.k.a* `viral`? It is crucial to understand the process which makes a certain topic viral while others do not as it could have wide ranging economic, social and political consequences. We have considered social network as directed, unweighted and connected graph. Interpretation of this graph is same as follower-followee relationship in twitter network. It contains nodes for all the users while there are edges from a followee to all its immediate followers. Once a node spreads any information, it will be passed on to its 1-hop followers.

Considering the above setting of information flow, For a given simple unweighted network graph $G(V, E)$, we attempt to study the information diffusion pattern and model it mathematically to represent this behavior that can run efficiently on general purpose parallel framework like CUDA to ensure faster execution. Here, the graph is considered to be static in nature, in which changes in nodes or edges are not allowed.

3.2 Prior works

Online social networks have become an effective channel for spreading people's words and playing a crucial role in influencing hundreds of millions of its users. In recent years, some of study on information diffusion in social network has been done, while very little work[12] has been done on this study using *Graphics Processing Unit(GPU)* in order to increase performance.

We have studied that in most of the earlier works, there are three different approaches

that has been followed for modeling diffusion problems. These are Diffusive Stochastic model, Independent Cascade and Linear Threshold model and Infectious disease model.

Firstly a, Diffusive Stochastic model is presented by S. Rajyalaxmi et al.[13] It is a competitive model where a list of topics are stored in a *global list* contains in which new topics are being added by following a Poisson process. Each node maintains its own local list. Whenever, a node is active, it looks for its neighbors and global list and chooses one of topic probabilistically. This topic is added to its local list and process continues. Topics in user list are associated with weights which are decaying exponentially to simulate the real world behavior. This model is more biased towards the new topics in the network. Older topics may retain or grow in strength in local neighborhoods only if those neighborhoods see a growth in instances being created in the neighborhood, capturing the notion that users in a social network are often peer pressured into responding to a particular topic if their neighbors are interested in it.

Another efficient approach is presented by Taxidou et al.[14] which utilizes Independent information cascades formed by different topics in the network. Independent cascade model represents information diffusion using influence paths that express the relationship of influence of one per person due to other. The set of influence paths form a social graph that shares a common root as the user who initiated the topic in the social network. This set is called as *information cascade*[15]. In cascade, social network users are represented as nodes which are influenced by the root or another user. Edges of the cascade represent edges of the social graph over which influence actually spread. In general, influencer is the node who exposed the information to its neighbors and influence on them in such a way that they forward this information recursively. In order to analyze two metrics were introduced *Connectivity-Rate (CR)* and *Root-Fragment-Rate (RFR)*. These metrics are related to connectivity between information cascades. The *Connectivity-Rate* assesses whether there is a connection between two users (nodes) in the cascade. It returns the percentage of users that have at least one connection and are thus influenced by the another user. The *Root-Fragment-Rate* assess whether there is a path to the root user from every other user. It returns the percentage of nodes that are connected with the root directly or via an influence path over multiple users. CR provides a very basic and loose indicator whereas RFR utilizes a very strict notion. These two metrics together provide a sensible bounds on other vital metrics in information propagation.

Next, Xu et al.[16] models this problem as infectious disease model or epidemic model. Disease model is widely adopted in studying information diffusion because it shows a deep relation between information spread and epidemics. Mostly SIS and SIRS models are used to interpret information diffusion behavior among social network. In SIS model, users are divided in two groups, *Susceptible* and *Infected*. Initially, person X does not know the information (*susceptible*), after an exposure to the information from root node (user who initiated the topic) or by another users (typically neighbors), X gets to know the information (*infected*) with probability p(*infectiousness*). In SIRS model, *susceptible* user once exposed to information become *infected*, but after some time user forget that information (*Recovered*) and become *susceptible* again.

3.3 Our work

Due to popularity of Social Network, millions of user are connected to each other via these and share their views and other media on daily basis. Therefore, a large number of interactions are being made due to these activities. Modeling these activities in a legacy computing environment (*sequential*) is not feasible, as it becomes slow when the number of users and interactions increase. While parallel and distributed systems provide a reasonable solution for such problems with significant speedup in execution time and scalability for large number of users.

3.3.1 Graph generation

Here, we have modeled a social network graph as a small-world network [18][19]. We have generated this network using Watts and Strogatz model. This model is a random graph generation model which produces graphs with small-world properties, including short average path lengths and high clustering. Most of the social networks and other interaction networks exhibit small-world properties. The phrase *six degrees of separation* is often used as a synonym for the idea of the *small world* phenomenon which is coined by Frigyes Karinthy in 1929 and popularized by a 1990 play written by John Guare. Six degrees of separation is the theory that everyone and everything is six or fewer steps away, by way of introduction, from any other person in the world, so that a chain of "a friend of a friend" statements can be made to connect any two people in a maximum of six steps.

Algorithm for generating graph using Watts and Strogatz Model:

> Given the desired number of nodes N, the mean degree K(assumed to be an even integer), and a special parameter β, satisfying $0 \leq \beta \leq 1$ and $N \gg K \gg \ln(N) \gg 1$, the model constructs an graph with N nodes and $\frac{NK}{2}$ edges in the following way:
>
> 1. Construct a regular ring lattice, a graph with N nodes each connected to K neighbors, $K/2$ on each side. That is, if the nodes are labeled $n_0, \ldots, n_{N-1}$, there is an edge (n_i, n_j) if and only if $0 \leq |i-j| mod(n - \frac{K}{2}) \leq \frac{K}{2}$.
> 2. For every node $n_i = n_0, \ldots, n_{N-1}$ take every edge (n_i, n_j) with $i \leq j$, and rewire it with probability β. Rewiring is done by replacing (n_i, n_j) with (n_i, n_k) where k is chosen with uniform probability from all possible values that avoid self-loops ($k \neq i$) and link duplication (there is no edge $(n_i, n_{k'})$ with $k' = k$ at this point in the algorithm).

3.3.2 Diffusion model

In order to model the flow of information in social network, we have extended the model presented by S. Rajyalaxmi et al.[13]. In this model we manage a global list for seed topics and each node manages its own local list which contains the topics it is aware of in the network. New topic are introduced in the global list using poisson arrival. Each topic is associated with a weight attached that shows the user's interest level in it. These

weights decrease exponentially over time. On each activity, node selects new topic either from global list or from its neighbors in probabilistic manner utilizing weights to calculate probability of each topic in its neighborhood among competing topics.

This model is a time-evolving stochastic model that captures some of the key features of the diffusion of topics in a social network. A stream of exogenous topics is presented to the network's users as *global list*. New topic in *global list* is added by poisson arrival process. Each exogenous topic has a weight that decays exponentially with the passage of time, reflecting the way in which interest in topics decreases as they get older.

Activity time for all nodes follows uniform distribution and is independent of the distribution of topic arrival in global list. In each iteration node generates a random number and decides to be active or stay inactive. If a node becomes active, it decides to talk on a topic from the global list or from one of the topics in the lists of its neighbors. This action creates an instance of that topic in the its *local list*. Entry in *local list* contains an weight associated with each topic, that decays exponentially with the passage of time, reflecting the way in which interest in topics decreases as they get older. The probability distribution by which the user decides which topic to speak on is determined by the weights of the topics in the global list and the weights of instances in its neighbors lists. Futhermore, we have proposed three different schemes to simulate these actions, which we will discuss later in section 3.3.4.

3.3.3 Diffusion model parameters

We started with the diffusive stochastic model as baseline model and done some modification in it. This modified model represent the information diffusion using the following parameters-

1. *Topic Interarrival time(λ_1)* : A new topic enters into network by appearing in *global list*. These new topics are being added to global list by a poisson point process with density λ_1.

2. *User Activity time(λ_2)* : Once active, user performs an action which involves creating a new instance of topic that has already appeared in the system. The activity time for all the users are independent of each other and the time of topic arrival in the global list.

3. *Global Weight parameters (A,α)* : In global list, each topic is associated with a weight which is initialized by A at its arrival time. Weight associated with the topic gives a notion of interest in the topic. When the topic becomes older in the network, its weight decreases. It is also observed that people mostly talk about recent happenings in the social network. Thus decreasing weight over time simulates the real situation. All the weights decay exponentially over time with a non-negative decay parameter α. Therefore, the weight of a topic in the global lost decays with time and for an topic that appeared at a time t, the weight at time $t' \geq t$ is $Ae^{-\alpha(t'-t)}$.

4. Local Weight Parameters (B,β) : Similar to global list, topics in the local list also loose interest over time. Therefore, the weights in the local list also decay over time with a positive decay parameter β. For a topic instance created at time t by user, the weight of the topic instance at time $t' \geq t$ is $Be^{-\beta(t'-t)}$, where B is the initial weight of the topic.

5. *Reset Weight C*: A topic t which is already introduced in the network gains interest of users if it is discussed again in the network. Therefore, we include a parameter *reset weight* to model the boost in interest for existing topic. When a topic is discussed, its weight is reset to C in the local list.

In our model, we assume that the activity time of each user is independent of each other. Therefore, activity of user (node) shows an inherent parallelism, which is exploited in this work. However, topic selection in each activity is interdependent and depends on the local lists of neighbors and global list weights. In our simulation, we have followed node level parallelism, where each user (node) is assigned a thread which functions in parallel independently. Since Graphic Card Unit(CUDA device) is limited in resource a efficient memory management is desirable.

3.3.4 Topic diffusion schemes

In proposed model, we have considered that an user (node) is influenced by its neighborhood and acts as per available information in its neighborhood. The time is discrete, and at each time slot, each node probabilistically decides to be active or not. An activity means to post a feed or tweet etc. that we will refer as copying a topic. To simulate this dependence and to understand the diffusion process more closely, we have proposed three models for topic selection, which are as follows:

1. **Opportunistic Scheme**: In this scheme, each node-i, $i \in V$, becomes active with a predetermined probability p and determines to choose a new topic. A new topic can be chosen either from global list or from its neighbors. Node i chooses one of the sources probabilistically. If a node decides to choose the topic from global list, new topic is added to its local list directly and topic's weight is set to *reset weight C*.
   If a node decides to choose new topic from its neighborhood, firstly it chooses a node j from its one hop neighbors. For our model, we have considered network as an unweighted graph, therefore, all its neighbors are equi-probable. Once it chooses a neighbor, it selects a topic from neighbor's local list probabilistically, where a topic with higher weight has the higher probability to be chosen than rest. This scheme depicts the behavior of user where all its neighbors have equal influence and only topics of a chosen neighbor compete each other not the topics or activities of other neighbors. This scheme is biased towards new topics, similar phenomenon is also observed in online social networks. Probability of choosing a new topic can be calculated as follows:

   Probability calculation: Let a node(user) i is active at time $t \geq t_0$ and decides to choose a topic from global list GL. Let GL contains m topics with non-zero weights at time t. New topic $K \in GL$ is chosen probabilistically from global list with a probability

   $$p_K(t) = \frac{w_K(t)}{\sum_{\forall i \in G} w_i(t)} \tag{3.1}$$

   where $w_k(t)$ is the weight of topic K at time t in the global list.
   Alternatively, node i may decide to choose a topic from its neighbors. Let

the node have $n(i)$ neighbors(followers). A neighbor j is chosen randomly with an equal probability of $\frac{1}{k}$ among all its neighbor. Let L_j be the local list of j with topics having non-zero weights. In this scheme, a topic K is chosen randomly from the local list of j with a probability

$$p_K(t) = \frac{w_K(t)}{\sum_{\forall i \in L_j} w_i(t)} \tag{3.2}$$

where $w_i t$ is the weight of topic i at time t in the local list L_j.

This model is simple and suitable to CUDA platform as very less memory interactions exist among different nodes.

2. **Competitive Scheme** : In this scheme, a node becomes active with a probability p and determines to choose a new topic K. Unlike, opportunistic scheme, it collates global list and local lists of all its neighbors together to get a list of topics LT with nonzero weights and aggregate the weights of available topics (here, addition is chosen as an aggregation function). Now, a new topic is chosen from it probabilistically, while topic with higher aggregate weight has higher probability than other competing topics. The probability of choosing a particular topic K from LT is given by:

$$p_K(t) = \frac{w_K(t)}{\sum_{\forall i \in LT} w_i(t)} \tag{3.3}$$

where $w_i t$ is the aggregated weight of topic i at time t in the list LT.

This scheme simulates the competitive behavior in social network where a user is more influenced if multiple number of neighbors are talking about the same topic rather than individual one. This scheme simulates the real world behavior more closely.

Probability calculation: Let global list G contains some topics with non-zero weights. Upon becoming active, a node i determines to choose a topic at time $t \geq t_0$. Node i have $n(i)$ neighbors(followers). Let N be the set of its neighbors, where $N = \{i_1, i_2, i_3, \ldots, i_k\}$. Each neighbor $j \in N$ manages its own local list L_j. According to this scheme, LT is the set of all topics available in the global list and in the local lists of N. In LT, weights related to a topic is added together, thus LT is a list of aggregated topic weights present in G and in the local lists of N. Let a topic K is chosen from LT. Then the probability of choosing this topic is can be found by eqn. (3.3).

While this scheme seems more realistic, it comes with a implementation cost. Its implementation in CUDA have performance issues as degree distribution of underlying graph is not uniform in social network which exhibits small world property. Variation in degrees of the nodes is not small, which increases the epoch(round) time to the execution time of the node with the highest degree. This unbalanced execution pattern reduce the time-performance of our simulation.

3. **Push-Gossip Scheme**: This scheme adopts best features from both of above schemes i.e. balanced execution from *opportunistic scheme* and competitive behavior from *competitive scheme*. In this scheme, we introduce a new data structure

named log which stores the history of activities happened in the past. *log* is a circular array based data structure, which starts overwriting its content from start once full. Each node maintains its own log of a fixed size determined at the start of simulation. Once a node talks about some topic, it pushes an update to all its neighbors and this update gets recorded in the *logs* of its neighbors.

In subsequent iterations, when a node becomes active again and determines to choose a new topic, it looks at its local data structure *log* and global list to choose a new topic K. In *log*, we allow to store duplicate instances of topic to capture the competitive topic behavior. If multiple nodes have talked about the same topic in neighborhood in the past, log will contain multiple instances of it.

> *Probability calculation:* Let global list G has some topics with non-zero weights. At time $t' \geq t_0$, a node M becomes active and determines to choose a topic. Node M manages a log L_M of a fixed size l. Whenever, a node is active, it collates global list and its log to form a list of topics. Let this list be $T = \{G \cup L_M\}$ and $freq_i$ represent the frequency of topic i in T. It chooses a topic from T with a probability proportional to its frequency in T.
>
> If a topic K is chosen by node M from T, probability of choosing this topic can be represented as:

$$p_K(t) = \frac{freq_K}{\sum_{\forall i \in T} freq_i} \tag{3.4}$$

In this scheme, Since all nodes have *log* of same size, all threads are balanced and execute in perfect synchronization. This scheme does not suffer from the performance bottleneck as discussed for competitive scheme.

Chapter 4

Implementation with CUDA

In the last chapter, we discussed our model and different topic selection schemes. For our implementation we have used general purpose parallel CUDA framework. In CUDA framework, all parallel blocks are implemented as kernel functions. For implementation, we have divided our entire program into several parallel blocks and wrote individual kernel functions for them. Kernel function *init_global_list* initializes global list configuration, while *init_local_list* initializes local lists of all nodes. Rest of the model is implemented in a kernel named *Simulation*, which creates one thread for each node, associates required data-structure with them. Hereafter, All threads collectively execute in parallel for a specified number of iterations. In this section, we provide details of the tweaks applied to improve efficiency of our model in CUDA programming framework.

4.1 Implementation challenges

4.1.1 Memory efficient graph storage

In General, graphs are stored either as adjacency matrix or adjacency list. Adjacency matrix stores graph in a $N \times N$ matrix and contains 1 whereas there is a direct link between nodes, where N is the number of nodes in the graph . It provides an $O(1)$ access when queried for *whether two node shares an edge or not*, but requires $O(N^2)$ memory. Due to space constraint, this scheme is not efficient for large graphs. Storing graph as adjacency list is a space-friendly option, as it stores edges in linked-list fashion for each node. This representation surely consumes less space but it is slow in access.

Since, in our model, each node requires its neighbor list only which is essentially an adjacency list corresponding to that node. Because of variable size of edge lists for each vertex, its GPU representation may not be efficient under the CUDA model. CUDA memory allocation and accesses are more suitable for arrays and static variable therefore, managing linked-list is not efficient. If we represent these linked-lists as an array of the size equal to largest degree of node in network at each node, we need a lot of memory because our degree distribution is not uniform.

To get rid from this problem, we represent the graph $G(V, E)$ in a compact adjacency

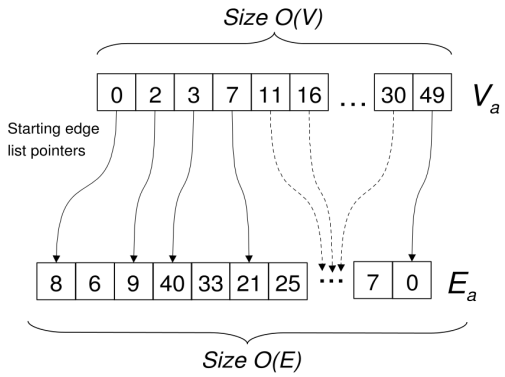


Figure 4.1: Compact Edge-List representation of Graph

list form where adjacency lists are packed into a single large array. Vertices of graph G are represented as an array V_a. Another array E_a of adjacency lists stores the edges with edges of vertex $i + 1$ immediately following the edges of vertex i for all i in V. Each entry in the vertex array V_a corresponds to the starting index of its adjacency list in the edge array E_a. Each entry of the edge array E_a refers to a vertex in vertex array V_a. An example of such representation is shown as in the Fig. 4.1

It takes $O(V + E)$ memory space, which is linear in input. This graph is stored in global memory of CUDA device. In CUDA framework, access to global memory is managed block wise. Therefore, once a block containing first neighbor of a node is accessed, its consequent memory locations contain other neighbors, which can be accessed by node sequentially.

4.1.2 Improved memory access

As global memory latency is much higher than shared memory latency, if global memory is accessed large number of time, it may slowdown the entire simulation performance. In a graph G, a node may have neighbors scattered across the graph, which may require multiple access to global memory in order to prepare a list for its neighbors. Here, we have proposed a new scheme to reduce the number of memory accesses. It is observed that nodes usually form communities in online social network. These communities may have formed because of their social, professional and behavioral interactions. It is also observed that most of the communications in social networks are of intra-community type. Very little percentage of communications go beyond the boundaries of these communities. Fig. 4.2 shows community structure in a subnetwork of facebook (325 nodes, 5028 edges), while Fig 4.3 shows community structure in a subnetwork of twitter (116 nodes, 728 edges) [Data source: SNAP[20]]. It can be observed in Fig. 1.1 that a topic spreads in a community first and then moves out to other communities using inter-community edges.

In CUDA, memory global memory accesses are done block wise. User actions in a social network are influenced primarily by its neighborhood. Thus, if a strong community edges are stored together in global memory, single memory block access may be sufficient to fetch all its neighbors and all other nodes in CUDA block may utilize this access read for their execution too. In our simulation, we have carried out this improvement on a graph as shown in fig 4.4, which have 127 nodes and 1223 edges having 6 communities in total.

Nodes are relabeled with new node ids based on their communities. In our simulation results, we have observed an improvement in execution time with new relabeling scheme.

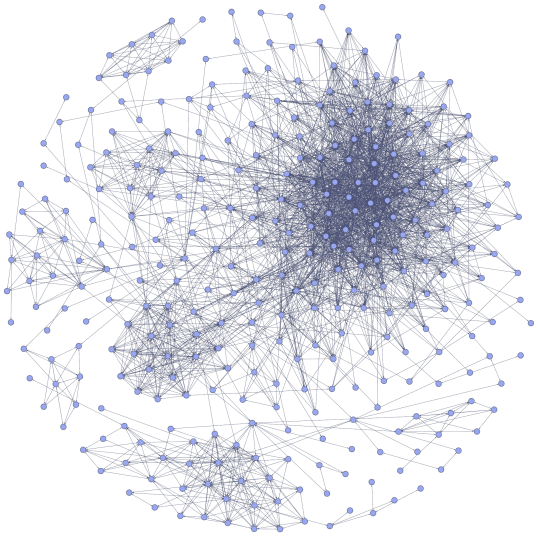


Figure 4.2: Facebook subnetwork

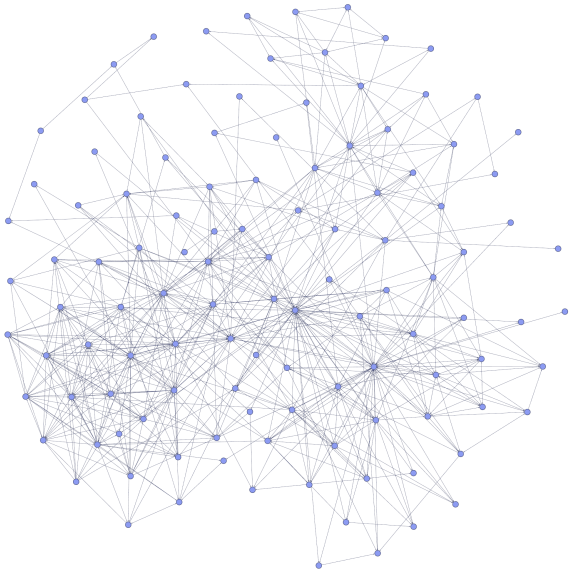


Figure 4.3: Twitter subnetwork

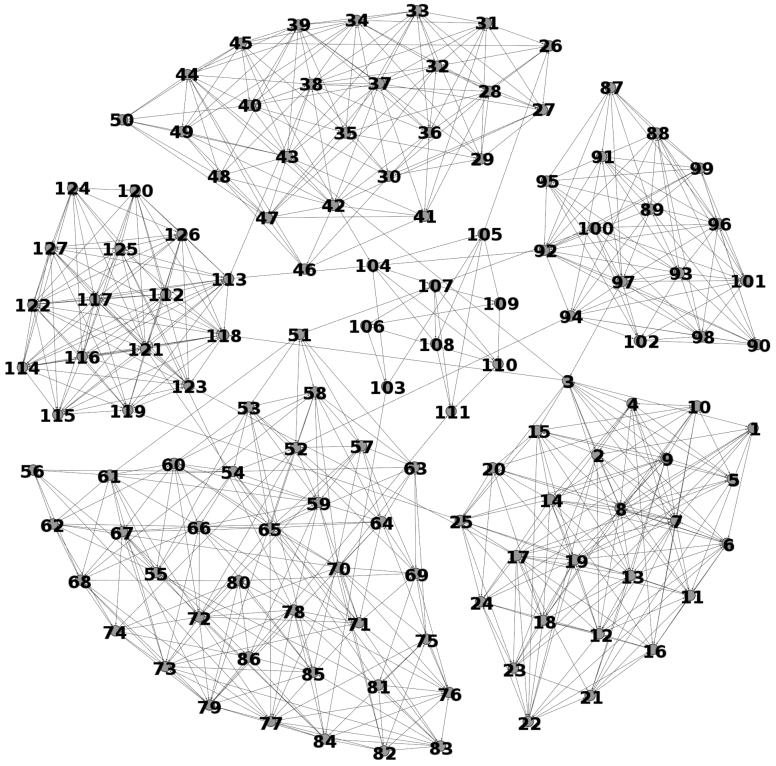


Figure 4.4: Improved Memory Access using Node Relabeling

4.2 Experimental Results

4.2.1 Hardware overview

To perform our simulations, we have carried it out using CUDA framework on NVIDIA Tesla $C2075$ graphic card (448 cores). While NVIDIA graphics card is acting as device, Intel Xeon CPU $E5-1650$ acts as a host in CUDA framework. Host CPU boasts 6-cores (hyper-threaded to 12-cores) where each core is clocked at 3.20GHz. It sports a total 12MB cache in total with a max memory bandwidth of 51.2 GB/s. All sequential simulations are carried out on this CPU alone while parallel simulations are carried out keeping Intel Xeon as host and NVIDIA Tesla as device.

4.2.2 Graph generation and model parameters

In order to simulate small-world behavior of social network, underlying graph is generated using Watts-Strogatz[19][18] model as described in section 3.3. This model is widely accepted to represent small-world network. In our simulation we have observed a significant speedup for the graph with more than 10000 nodes. For the test purposes, we ran our simulation till 100000 nodes. To compare execution time results among CPU execution and different schemes in GPU we have set our diffusion model parameters as follows:

Global weight parameter A and local weight parameter B, both are set to 1.0 to allow seed topic to have Maximum weight. Global decay parameter α is set to 0.1. With $\alpha = 0.1$, a topic can survive up to next 20-25 iterations if no activity related to this topic happens during this time frame. Similarly local decay parameter β is set in range from $0.05 - 0.1$ to control the diffusion process. Lowering β, increases life time of a topic, which allows more diffusion in the network. Reset parameter C is set in a range of $0.5 - 1.0$ to control the diffusion process. At each node, local list size is capped at 10. In push-gossip scheme, instead of local lists, we maintain logs at each node which stores history of up to 25 recent activities. As sated earlier, It is maintained as circular list to avoid overflow and allow recent activities to replace the older ones.

For simulation, *globallist* is stored in global memory of device. Since local lists and the global list are to be accessed by all nodes, these lists are also stored in global memory. Although all the updates in the local list are maintained by the node itself to which it is associated. All other operations related to model are implemented as device functions, which executes in GPU itself. In push-gossip scheme, logs are also maintained in global memory to provide global access to all nodes.

4.2.3 Simulation analysis

To study the information diffusion, we carried out our simulation on networks of different sizes and degree distribution. We observed their execution time for all three proposed schemes in GPU and compared them with sequential execution time in CPU while keeping all model parameters fixed.

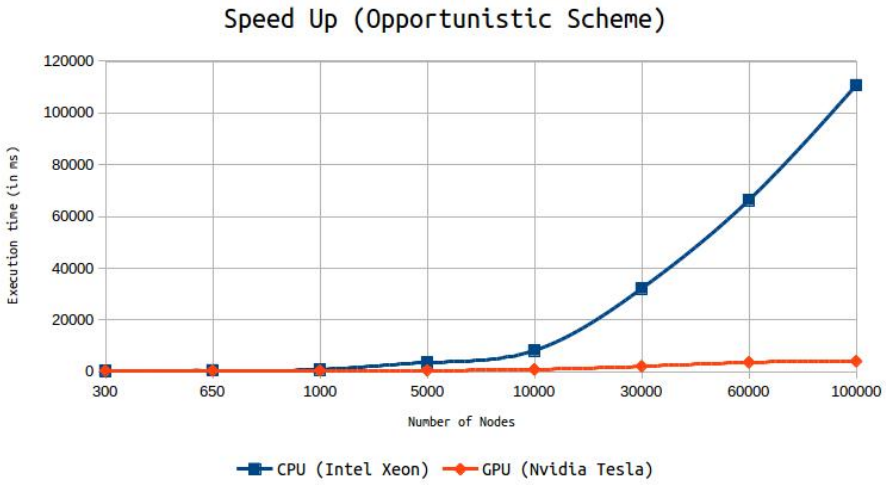


Figure 4.5: Speedup in Opportunistic Scheme

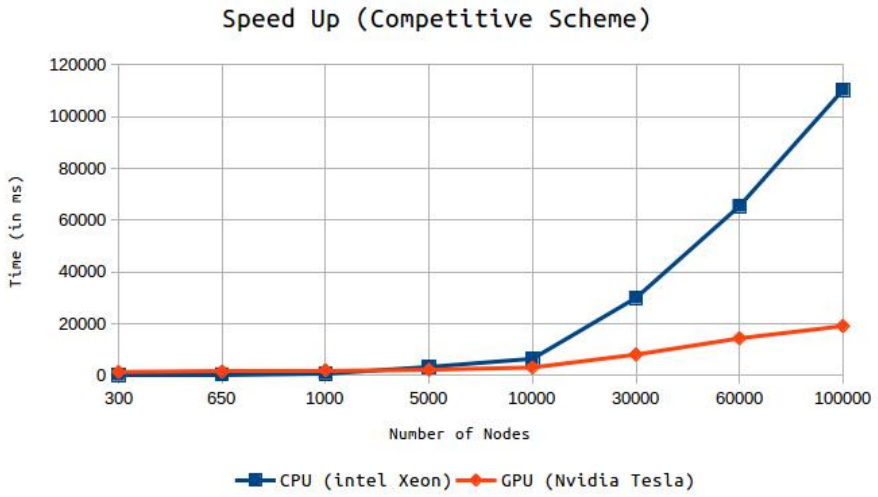


Figure 4.6: Speedup in Competitive Scheme

Opportunistic scheme: As stated earlier,due to absence of competitiveness among topics, this scheme is best suitable for CUDA implementation. We were able to accommodate about 1 million nodes and successfully ran the simulation as per model specification. Execution of the same on CPU took a really long time as expected. Fig. 4.5 shows a relative performance in execution time of sequential and parallel implementation. From the simulation results, it is evident that GPU implementation of our model have achieved a speedup of 27 for a graph of 10000 nodes with 40 competing topics in the network.

Competitive scheme: To simulate competitive behavior, upon each activity, new topic is selected from all topics present in local list of its neighbors and the global list, which requires memory of $O(T)$ to aggregate the topic weights, where T is the number of topics competing in network. As aggregate list is to be accessed by the node itself,

we have implemented it in shared memory of the CUDA block to achieve faster memory access. It should be kept in mind that shared memory is limited to 48 MB per CUDA block (in Tesla C2075), which limits the number of maximum topics present in network. If memory required for all nodes exceeds shared memory limit of device, we can still implement the same functionality by switching its location from shared memory to global memory. However, it will reduce the performance up to a ratio of global memory access latency and shared memory access latency. We ran our simulation on networks of different sizes up to 100000 nodes. Even though this scheme needs access to local list of neighbors which may be distributed un-evenly in the memory, we observed a significant speedup in execution time as shown in fig. 4.6. For a graph of 100000 nodes with 40 competing topics in the network, our GPU implementation achieves a speedup of 5.8.

Push-Gossip scheme: In this scheme, if a node takes an action, its update is being pushed to logs of its neighbors. Similar to competitive scheme, to perform probability calculations we need an extra memory of the order of number of topics present in the network. Since, this extra memory is accessed by the node itself, we have implemented it in shared memory of GPU device. Similar constraint of limited shared memory also hold here and at big scale it needs to be shifted to global memory. Although, this scheme adopts best of the features from aforementioned schemes, it also performs better than competitive one. For a graph with 100000 nodes, 40 topics and a log list of 25 instances, a significant speedup of around 5 in comparison of Intel CPU is observed (in fig. 4.7).

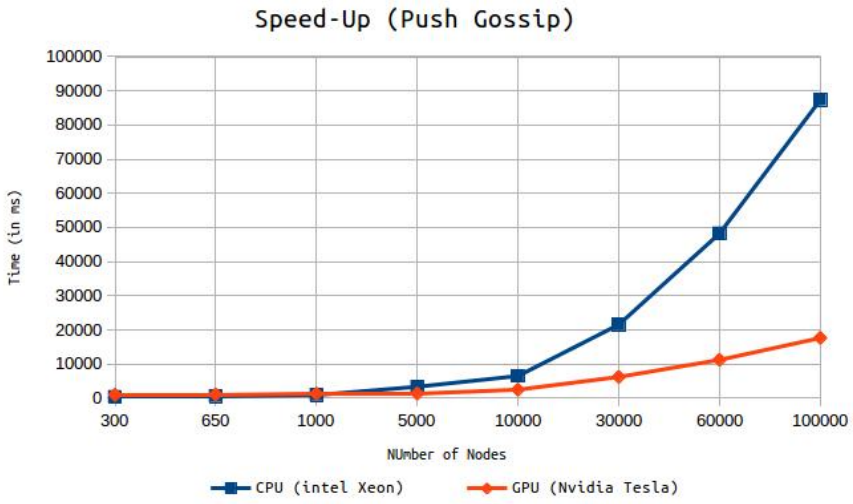


Figure 4.7: Speedup in Push-Gossip Scheme

Fig. 4.8 shows the comparative performance of different schemes in GPU (NVIDIA Tesla C2075). It is clear that opportunistic scheme takes least time in execution as expected, because there is no dependency in threads executions. Competitive performs worst as it is most interdependent scheme. Lot of execution time is spent in multiple memory accesses in global memory. Meanwhile, Push-Gossip scheme performs in between of these two schemes and captures best features from both schemes. This scheme is 1.4x times faster than the competitive scheme. Its performance can be tuned using log size as a tuning parameter. Increasing log size will allow more competition and vice versa.

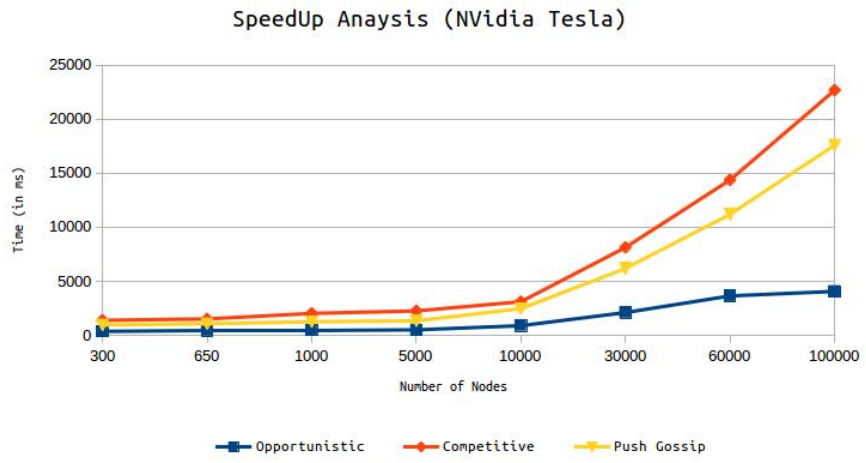


Figure 4.8: Relative Speedup in three schemes (all in GPU)

Also, We ran our simulation on subnetwork graphs from Facebook and Twitter social network available in SNAP[20] databases. These subnetwork also follows power law distribution. In our simulation, we have seen viral and non-viral regime in topic count vs. time plot similar to results obtained by S. Rajyalaxmi et al.[13]. A non-viral topic diffusion is represented in fig. 4.9. Fig. 4.10 shows diffusion of a viral topic over subnetwork from facebook(325 nodes, 5028 edges). This diffusion is very slow at the start of simulation but took pace at later time. Diffusion of viral topic in twitter subnetwork (116 nodes, 728 edges) is presented in fig. 4.12.

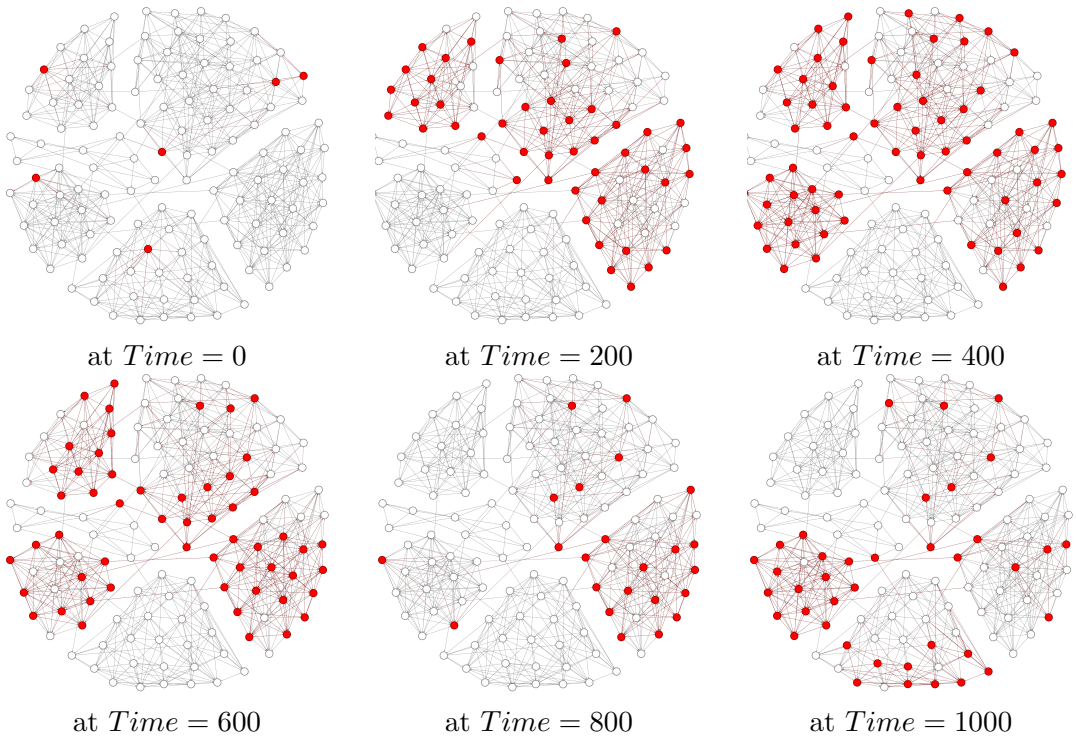


Figure 4.9: A Non-Viral topic diffusion in network (127 nodes, 1223 edges)

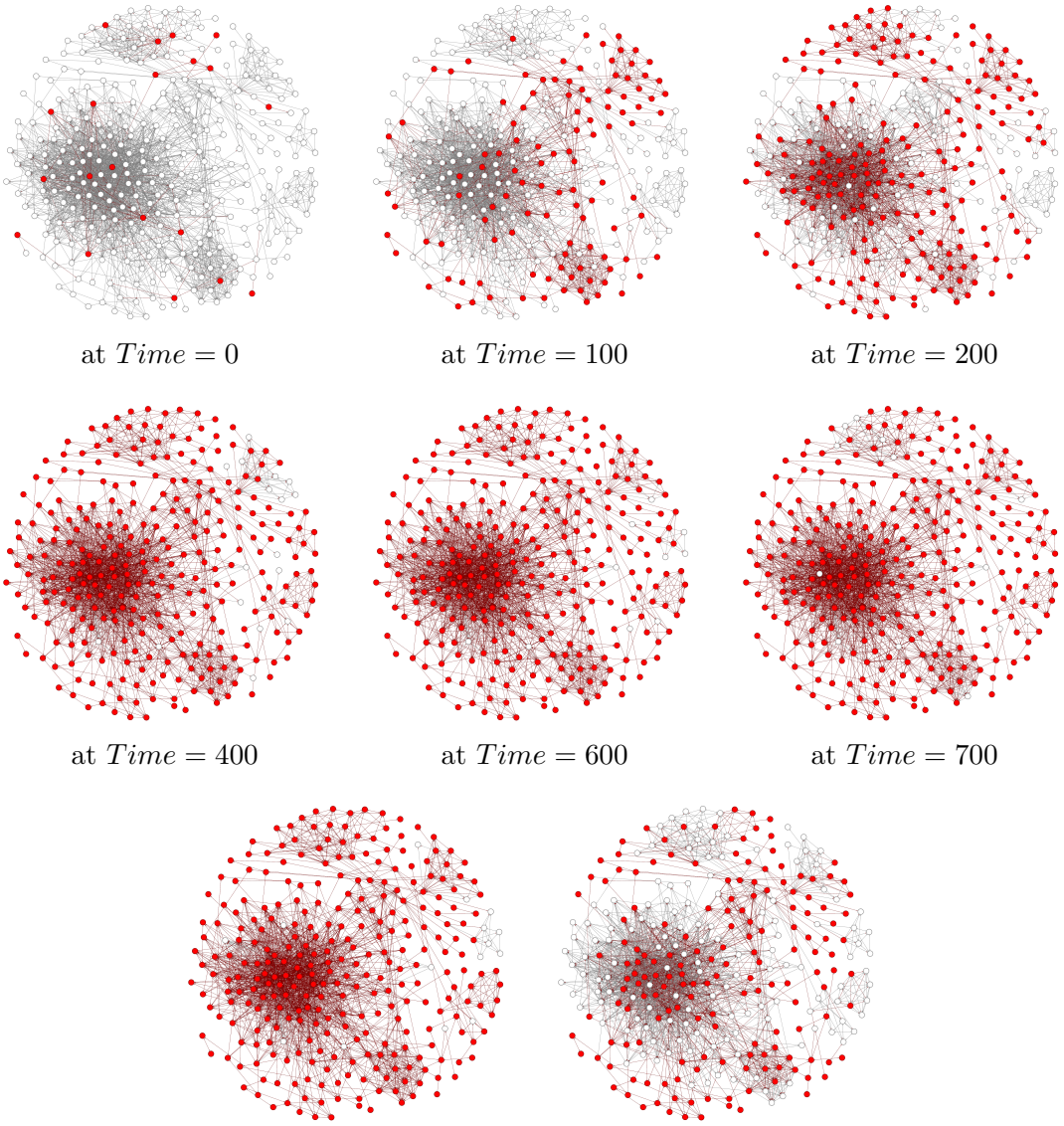


Figure 4.10: Viral topic diffusion on Facebook subgraph

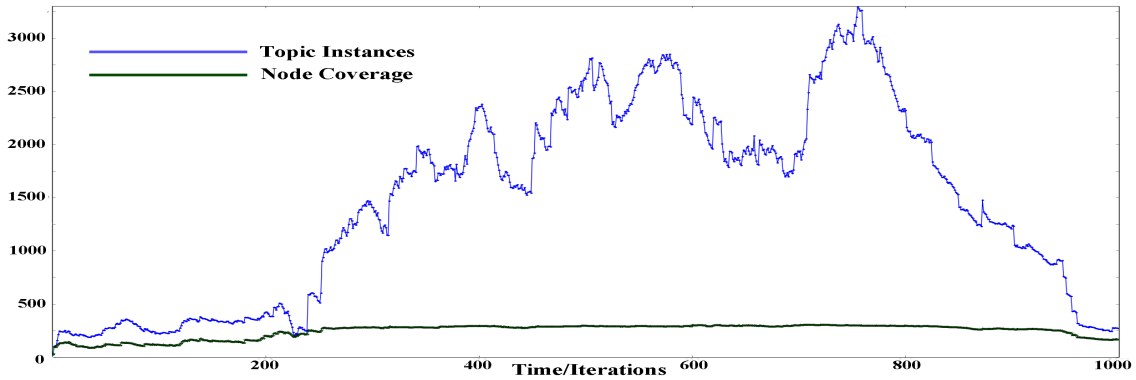


Figure 4.11: Topic instance vs. Node coverage for a single topic on Facebook subgraph

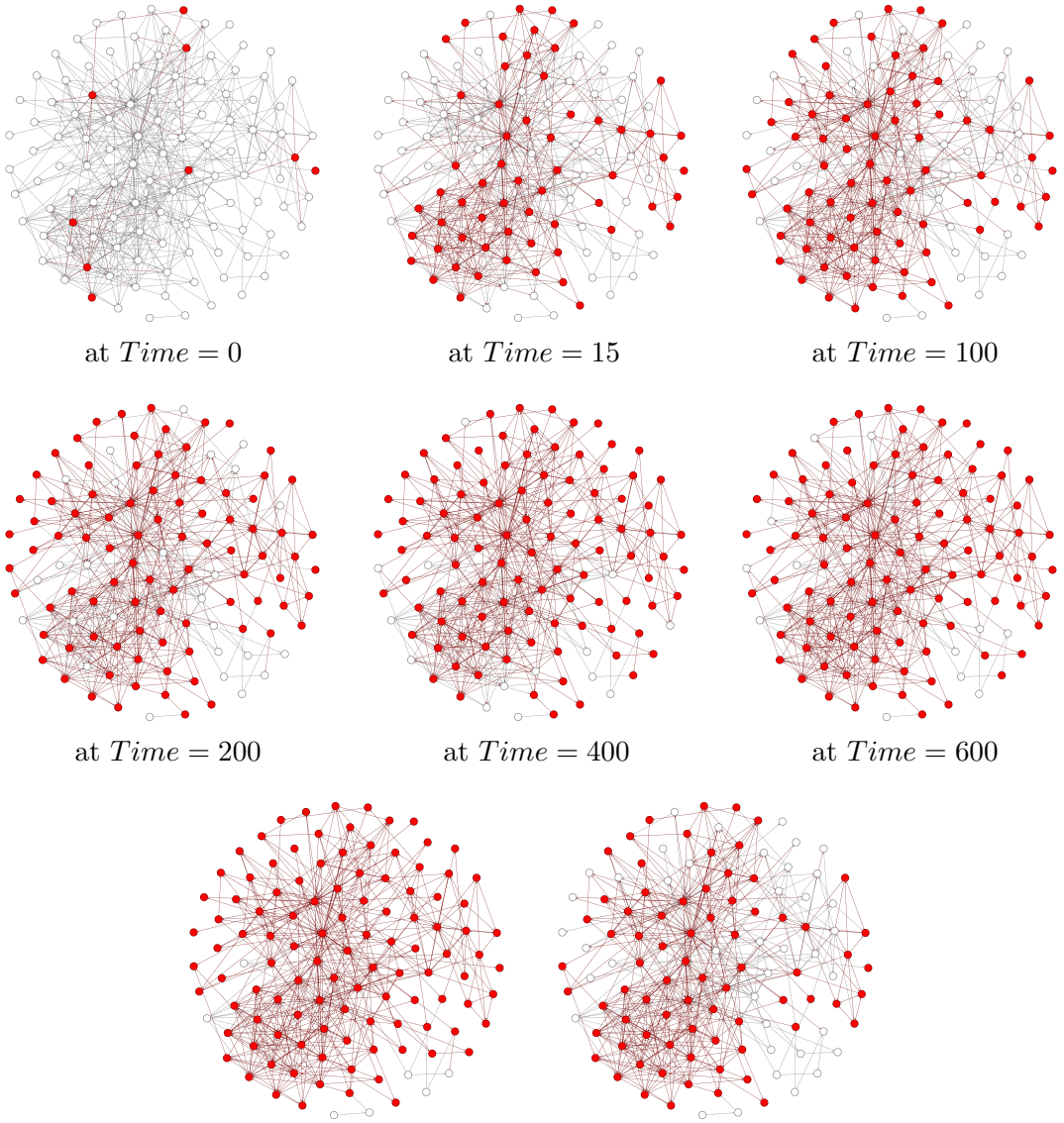


Figure 4.12: Viral topic diffusion on Twitter subgraph

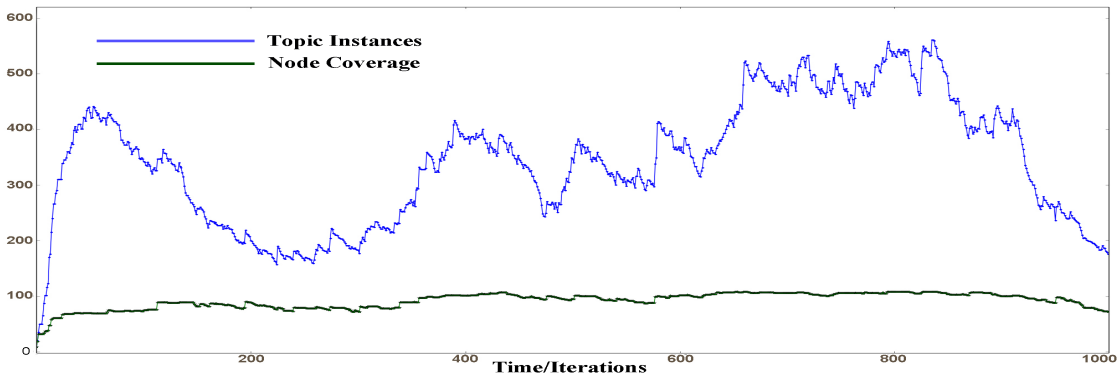


Figure 4.13: Topic instance vs. Node coverage for a single node on Twitter subgraph

4.3 Data gathering from social networks

As an attempt to validate our model behavior, we deployed a data gathering program to capture underlying network from twitter. To dump twitter network graph we wrote a program using $python - twitter$ package. This program uses $TwitterStream$ API to fetch follower and following details for users. An automated script was kept running for about three weeks and collected 21391747 nodes and 225553910 edges using following and followers information. To get the tweet data, we captured data stream on 25 #hashtags for 15 days. $python - twitter$ API [1] specifications are as follows-

1. *GetUser (self, user_id=None, screen_name=None)* : returns a single user with user_id or screen_name provided in JSON format. Returned object contains all information regarding user including name, location, time zone, account creation date, profile picture url, wallpaper url, follower count, following count, number of tweet posted since creation etc. Twitter allows 180 such requests in a 15 min. window.

2. *GetFriends(self, user_id=None, screen_name=None, cursor=-1, count=None)* : returns a list of users that user_id or screen_name follows in JSON format. Returned object contains user objects (maximum upto 200 users per request). Twitter allows 15 such requests in a 15 min. window.

3. *GetFriendIDs(self, user_id=None, screen_name=None, cursor=-1, count=None)* :returns a list of userIds that user_id or screen_name follows in JSON format. Returned object contains user objects (maximum upto 5000 users per request). Twitter allows 15 such requests in a 15 min. window.

4. *GetFollowers(self, user_id=None, screen_name=None, cursor=-1, count=None)* : returns a list of users who follows user_id or screen name provided in JSON format. Returned object contains user objects (maximum upto 200 users per request). Twitter allows 15 such requests in a 15 min. window.

5. *GetFollowers(self, user_id=None, screen_name=None, cursor=-1, count=None)* : returns a list of userIds who follows user_id or screen name provided in JSON format. Returned object contains user objects (maximum upto 5000 users per request). Twitter allows 30 such requests in a 15 min. window.

6. *GetSearch(self, term=None, geocode=None, since_id=None, max_id=None, until = None, since=None, count=15, lang=None, locale=None, result_type=m̈ixed)* : returns a list of tweets with tweetID, date of creation, re-tweet count in JSON format. Return object contains 100 tweets per request. *term* variable takes search query, which can be #hashtag or a keyword.*count* signifies the number of tweets to be fetched in this request, which is capped at 100. Other optional parameters can be used to filter tweets based on location, tweetIDs, duration, language, locale and popular/recent/mixed type. Twitter allows 180 such requests in a 15 min window.

In tweets, collected using *GetSearch*(), date of creation and re-tweet count are present however, who have re-tweeted it is missing, We have tried to create information cascade

[1] https://github.com/bear/python-twitter

using these tweet information. In future, we would like to emulate these tweets on our model to observe viral and non-viral regime patterns.

4.4 Inference

While studying simulation results on several graph with different number of nodes and degree distribution, we observed some interesting facts which explain information diffusion process. In this section, we will discuss about these facts.

Viral and Non-Viral regime: A viral regime is a special pattern observed in instantaneous topic count vs. time graph. As soon as a new topic is introduced in the network, users start talking about it. Since action time of different users are independent to each other and topic selection is implemented based on probabilistic calculations in our study, a topic may become viral in one run while may not be viral in next iteration. A viral topic shows a steady growth in terms of number of instances in network and infects large number of nodes over time. After it, as people loose interest in this topic, number of instances start decreasing and this topic moves out of the network. Fig. 4.14 shows the evolution of few topics in the network. Y axis showing number of copies on present of a topic in each iteration while X axis shows time movement. It is quite evident that couple of topic survives the network for a long time while other phase out from the competition quickly.

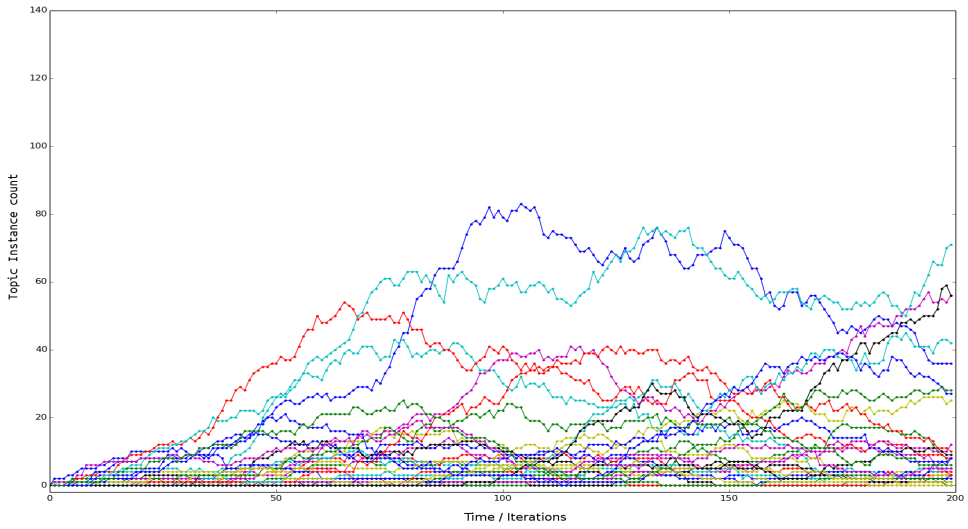


Figure 4.14: Topic instance count over time

In beginning, seed nodes talked about a topic and other followed it, which give a rise in number of instances of that particular topic. Once it is spread in the network, nodes continue to talk on this topic as per their interest towards it. These actions from several users help this topic to sustain itself in the network. It is human behavior that we loose interest in a topic over time, therefore, users in the network tend to loose interest in that topic and start talking about new topic which may takes over and spreads in the network. Real scenario of social network, also exhibits this competitive behavior among topics. A topic goes viral if it hits large number of users in the network and in general, these are the

topics of public interest. An personal media is less likely to go viral, as only close friends may find it interesting, not the rest of the world. Fig. 4.14 shows a viral and non-viral regimes in simulation which was carried out for 200 iterations.

Topic Instance count: To calculate instance counts of all topics, in simulation one need to count it individually from all nodes and this process need to be repeated for each time slot (iteration). This counting scheme causes a significant delay in each slot. Although, CUDA provides facility to print logs during program run, but the entire log will only be transferred to CPU at the end of kernel execution, which may consume significant amount of GPU memory during execution. Printing logs for large graphs for large number of iterations itself consume a lot of memory and may affects the program execution severely.

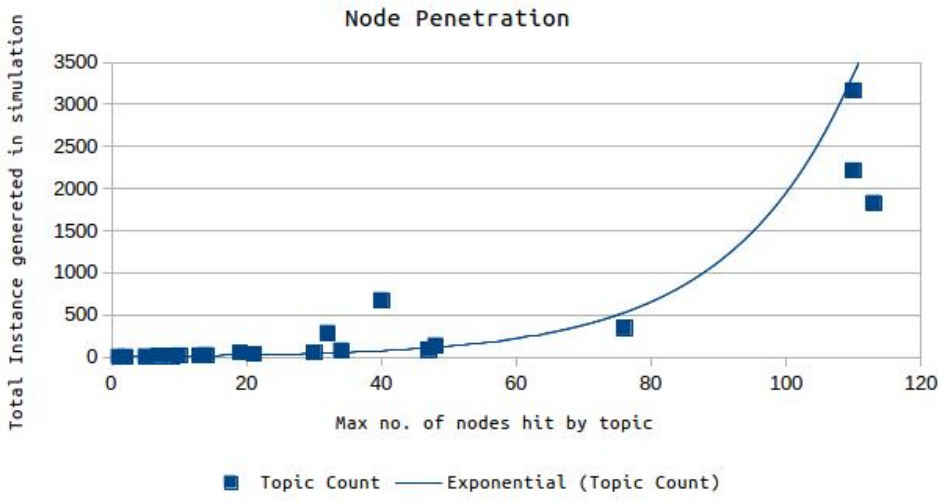


Figure 4.15: Node Penetration vs. Topic instance count (on Graph with 127 nodes)

To overcome this problem, we have established a relation in topic coverage (number of the nodes infected by the topic so far) and its actual instance count over the time. In simulation results, we found a relation in topic coverage and the actual instance count as shown in fig.4.11 for facebook network and in fig. 4.13 for twitter network. We tried to plot topic instance count vs. topic coverage [fig.4.15], and found that it follows an exponential relation. As the topic coverage increases in the network, it is more likely that topic will become more popular among people. Fig. 4.15 suggests that as soon as 76 nodes out of 127 nodes (about 60% node coverage) become infected, topic count has increased sharply. This threshold depends on the underlying graph structure. Therefore, once determined, it empirically can be utilized to predict viral behavior of the topics.

Effect of Graph Topology: To study how the graph topology affects the diffusion of the topic? we look at the number of clusters of nodes speaking on a particular topic simultaneously. Close inspection of simulation results show the evolution of the cluster sizes along with the diffusion of the topic for a viral and a non-viral topic. A non-viral topic is characterized by many small clusters disconnected throughout the lifetime of the topic. The number of clusters increases when there is a peak in the diffusion of the topic indicating that the many nodes speaking on the topic belong to clusters different from the existent ones. On the contrary, for a viral topic, we see a significant dip in the number

of clusters when the topic peaks in its diffusion. This is because of the formation of a giant cluster which encompasses a large fraction of the nodes speaking on the topic. The diffusion sees similar sized clusters when the topic is young and the formation of a giant cluster when the topic peaks. The non-viral topics see a very small fraction inside the largest cluster while the viral topics see a large fraction when it peaks in its evolution indicating that it is a giant component.

Effect of Seed Topic Distribution: In our simulation, we have observed that information diffusion is independent of seed nodes presence in the graph. We tried to relate topic diffusion with their presence in the network at t=0 (seed nodes) in fig. 4.16 and also the degree sum of these seed nodes in fig. 4.17, but didn't find any correlation between viral behavior and seed nodes. Moreover, we ran simulation with different local list initialization patterns and observed that network adjusts itself [after certain period of time, we have observed a sudden fall in topic count as weight of these topics in local lists goes to 0 (Fig. 4.18)] Thereafter, diffusion progresses at a normal pace. This period depends on model parameter α, β. If these values kept high, weight of topics in the network falls to 0 rapidly and we observe a sharp fall earlier.

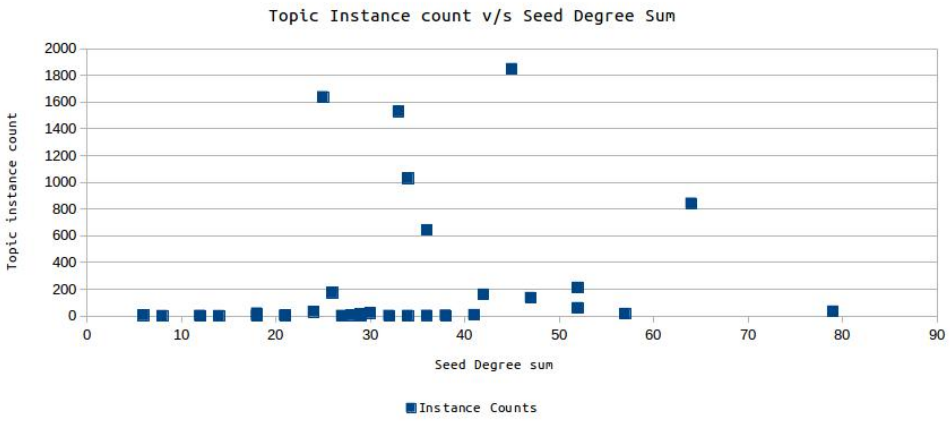


Figure 4.16: Node instance count vs. seed degree sum

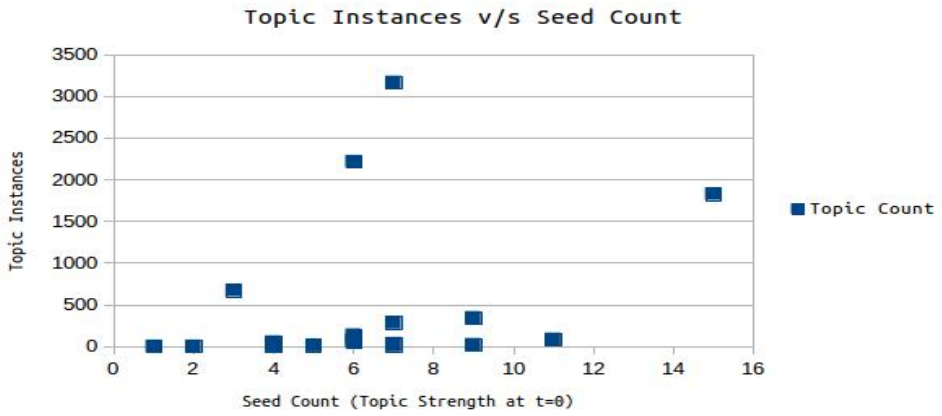


Figure 4.17: Node instance count vs. seed strength

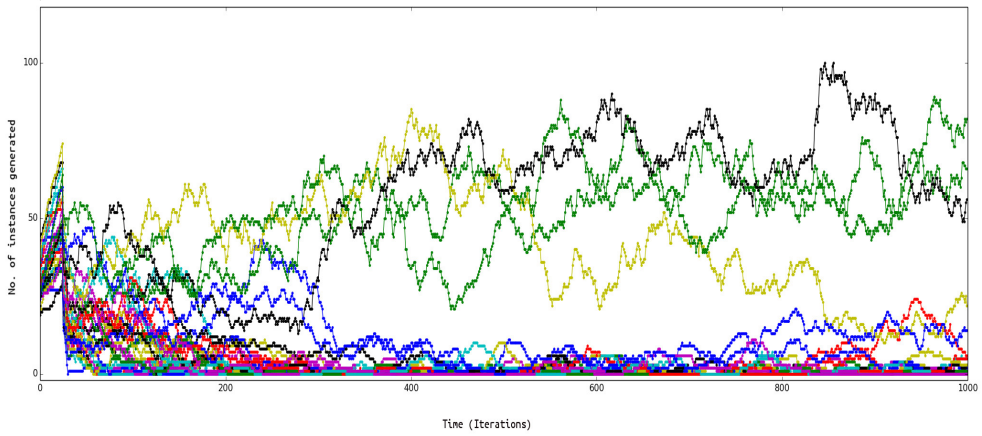


Figure 4.18: Information diffusion with local list initialization

Chapter 5

Conclusion and Future Scope

Information diffusion process study is one key problem in social network analysis. It is well known that even the fastest available models for information diffusion take sufficiently long time in sequential execution. This work shows the possibility of utilizing parallel framework to achieve faster execution and visualizing diffusion pattern in near real time. In this work, we have proposed three schemes, varied in computational complexity which captures different aspects of information diffusion. However, we recommend Push-Gossip based scheme, as it captures real social network behavior and also provides good speedup. In our work, we have proposed efficient memory utilization of all data-structures to achieve the maximum speedup.

In this work we have considered static unweighted graph, which can be extended to weighted graph by putting some affinity weights on edges. Higher the weight of a edge, follower is more likely to react to its followee actions. This weight can be calculated by observing history of their past relationships. Online social networks are dynamic in nature. Several new users join everyday and a few leave the network. Also, the relationships among users change dynamically with time. Thus, this model can be extended to handle dynamic graphs as well.

In the simulation, we have generated topics and scheduling user's activities on real world social network like facebook and twitter subgraphs. To analyze the user behavior more closely, existing simulation can be extended to social network feed data collected over a certain period of time for few hundred topics. We have tried to do this for our simulation on facebook and twitter. Twitter provides API's as discussed in section 4.3, but tweets available through this API are very less as compared to the original tweets, which may or may not provide enough picture for topic diffusion.

For efficient execution, instead of keeping information of entire feed in GPU, entire simulation can be split it into several timespans, which can be executed in sequence individually. Recently, Zhang et al.[21] proposed a new framework named GStream, which allows GPU to handle live memory updates itself. Implementing our model on Gstream can directly analyze the feed and required analysis can be performed.

Bibliography

[1] David A. Bader and Kamesh Madduri. Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray mta-2. In *Proceedings of the 2006 International Conference on Parallel Processing*, ICPP '06, pages 523–530. IEEE Computer Society, 2006.

[2] Pawan Harish and PJ Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *High performance computing–HiPC 2007*, pages 197–208. Springer, 2007.

[3] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Phys. Rev. E*, 76:036106, Sep 2007.

[4] Daniel Chavarrıa-Miranda, Mahantesh Halappanavar, and Ananth Kalyanaraman. Scaling graph community detection on the tilera many-core architecture. HiPC, 2014.

[5] Ahmet Erdem Sariyüce, Kamer Kaya, Erik Saule, and Ümit V Çatalyürek. Betweenness centrality on gpus and heterogeneous architectures. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, pages 76–85. ACM, 2013.

[6] Adrien Guille, Hakim Hacid, Cecile Favre, and Djamel A. Zighed. Information diffusion in online social networks: A survey. *SIGMOD Rec.*, 42(2):17–28, 2013.

[7] Philip N Howard, Aiden Duffy, Deen Freelon, Muzammil M Hussain, Will Mari, and Marwa Mazaid. Opening closed regimes: what was the role of social media during the arab spring? *SSRN 2595096*, 2011.

[8] Amanda Lee Hughes and Leysia Palen. Twitter adoption and use in mass convergence and emergency events. *International Journal of Emergency Management*, 6(3-4):248–260, 2009.

[9] Liang Zhao Edward Dougherty Yang Cao Chang-Tien Lu Naren Ramakrishnan Fang Jin, Wei Wang. Misinformation propagation in the age of twitter. *Computer*, 47(12):90–94, Dec. 2014.

[10] Eytan Bakshy, Itamar Rosenn, Cameron Marlow, and Lada Adamic. The role of social networks in information diffusion. In *Proceedings of the 21st international conference on World Wide Web*, pages 519–528. ACM, 2012.

[11] Nvidia CUDA Documentation. http://docs.nvidia.com/cuda/index.html.

[12] Sang Won Seo, Joohyun Kyong, and Eun-Jin Im. Social network analysis algorithm on a many-core gpu. In *Ubiquitous and Future Networks (ICUFN), 2012 Fourth International Conference on*, pages 217–218, July 2012.

[13] S. Rajyalakshmi, Amitabha Bagchi, Soham Das, and Rudra M. Tripathy. Topic diffusion and emergence of virality in social networks. *CoRR*, abs/1202.2215, 2012.

[14] Io Taxidou and Peter M Fischer. Online analysis of information diffusion in twitter. In *Proceedings of the companion publication of the 23rd international conference on World wide web companion*, pages 1313–1318. International World Wide Web Conferences Steering Committee, 2014.

[15] Jure Leskovec, Mary McGlohon, Christos Faloutsos, Natalie S Glance, and Matthew Hurst. Patterns of cascading behavior in large blog graphs. In *SDM*, volume 7, pages 551–556. SIAM, 2007.

[16] Bo Xu and Lu Liu. Information diffusion through online social networks. In *Emergency Management and Management Sciences (ICEMMS), 2010 IEEE International Conference on*, pages 53–56, Aug 2010.

[17] Feng Wang, Haiyan Wang, and Kuai Xu. Diffusive logistic model towards predicting information diffusion in online social networks. In *Distributed Computing Systems Workshops (ICDCSW), 2012 32nd International Conference on*, pages 133–139. IEEE, 2012.

[18] Duncan J Watts and Steven H Strogatz. Collective dynamics of small-worldnetworks. *nature*, 393(6684):440–442, 1998.

[19] Jon Kleinberg. The small-world phenomenon: An algorithmic perspective. In *Proceedings of the Thirty-second Annual ACM Symposium on Theory of Computing*, STOC '00, pages 163–170. ACM, 2000.

[20] Stanford Large Network Dataset Collection. http://snap.stanford.edu/data/index.html.

[21] Yongpeng Zhang and F. Mueller. Gstream: A general-purpose data streaming framework on gpu clusters. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 245–254, Sept 2011.