Summer 2015

High Performance Large Graph Analytics by Enhancing Locality

Naga Shailaja Dasari
Old Dominion University

HIGH PERFORMANCE LARGE GRAPH ANALYTICS BY ENHANCING LOCALITY

by

Naga Shailaja Dasari
B.Tech. June 2003, Kakatiya University, India
M.Tech. June 2006, Indian Institute of Technology, Kanpur, India

A Dissertation Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

COMPUTER SCIENCE

OLD DOMINION UNIVERSITY
August 2015

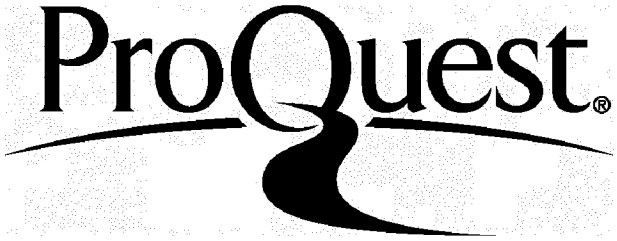
Approved by:

Desh Ranjan (Co-Director)

Mohammad Zubair (Co-Director)

Jing He (Member)

Bharat Madan (Member)



ABSTRACT

HIGH PERFORMANCE LARGE GRAPH ANALYTICS BY ENHANCING LOCALITY

Naga Shailaja Dasari
Old Dominion University, 2015
Co-Directors: Dr. Desh Ranjan and Dr. Mohammad Zubair

Graphs are widely used in a variety of domains for representing entities and their relationship to each other. Graph analytics helps to understand, detect, extract and visualize insightful relationships between different entities. Graph analytics has a wide range of applications in various domains including computational biology, commerce, intelligence, health care and transportation. The breadth of problems that require large graph analytics is growing rapidly resulting in a need for fast and efficient graph processing.

One of the major challenges in graph processing is poor locality of reference. Locality of reference refers to the phenomenon of frequently accessing the same memory location or adjacent memory locations. Applications with poor data locality reduce the effectiveness of the cache memory. They result in large number of cache misses, requiring access to high latency main memory. Therefore, it is essential to have good locality for good performance. Most graph processing applications have highly random memory access patterns. Coupled with the current large sizes of the graphs, they result in poor cache utilization. Additionally, the computation to data access ratio in many graph processing applications is very low, making it difficult to cover the memory latency using computation. It is also challenging to efficiently parallelize most graph applications. Many graphs in real world have unbalanced degree distribution. It is difficult to achieve a balanced workload for such graphs. The parallelism in graph applications is generally fine-grained in nature. This calls for efficient synchronization and communication between the processing units.

Techniques for enhancing locality have been well studied in the context of regular applications like linear algebra. Those techniques are in most cases not applicable to the graph problems. In this dissertation, we propose two techniques for enhancing locality in graph algorithms: *access transformation* and *task-set reduction. Access*

transformation can be applied to algorithms to improve the spatial locality by changing the random access pattern to sequential access. It is applicable to iterative algorithms that process random vertices/edges in each iteration. The *task-set reduction* technique can be applied to enhance the temporal locality. It is applicable to algorithms which repeatedly access the same data to perform certain task. Using the two techniques, we propose novel algorithms for three graph problems: k-core decomposition, maximal clique enumeration and triangle listing. We have implemented the algorithms. The results show that these algorithms provide significant improvement in performance and also scale well.

ACKNOWLEDGEMENTS

Foremost, I would like to thank my advisors, Dr. Desh Ranjan and Dr. Mohammad Zubair, for their continuous support and guidance. They have always been very encouraging and provided with valuable advise, both academic and non-academic. I have been very fortunate to have them as my advisors and without their support I would not have finished my thesis.

I would like to thank my committee members, Dr. Jing He and Dr. Bharat Madan, for their valuable time and support, for reviewing my thesis and for constructive feedback.

A big fat thanks to my husband, Neel, for being by my side through out the journey, for his love and support, and especially for taking care of our little daughter while I work.

I would like to express my gratitude to my parents, Narasaiah and Anjani, for their unconditional love, support, encouragement and for giving me the freedom of choice.

I would like to thank my little daughter, Lasya, for letting me work despite how much she hated it. Her smile and charm always kept me going.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

A graph is a set of vertices and edges that represent entities and the relationship between entities, respectively. Graphs are widely used in various domains. For example, the air traffic network can be represented using a graph where each vertex represents an airport and an edge represents a connection between the two airports. In a social network, each person can be represented by a vertex and the relationships can be represented by edges. In a web graph, each web page can be represented by a node and a hyperlink between web pages can be denoted by an edge. Graphs are also heavily used in many other fields including computational biology, commerce and sociometry.

The volume of data has been exploding in recent years. There is massive amounts of data being generated every minute. According to the statistics published in 2012 [1][2], every minute, Facebook users share over 700k posts , Twitter users send over 100k tweets, Instagram users share over 3600 photos, and over 500 new websites are created. Walmart receives over 1 million transactions per hour which is stored in its database which is estimated to be more than 2.5 petabytes. Now more than ever, the data collected is being made use of. The data is processed, modified, combined, analysed and visualized to extract useful information. The benefits of the extracted information can be very substantial. According to a case study published in [3], UPS, a postal service company acquired data from the sensors attached to more than 46,000 vehicles to track the speed, direction, braking and drive train performance. The data was then analysed and led to savings of over 8.4 million gallons of fuel by cutting 85 million miles off of daily routes.

In the recent years, a lot of attention has been paid for analysis of graphs. There has been significant rise in the breadth of problems requiring graph analytics. As a result, it is becoming increasingly important to efficiently solve the graph problems. There are many interesting and complex graph problems that needs to be solved for graph analysis. Unfortunately many graph problems are computationally expensive to solve. Coupled with the current large sizes of the graphs, it is highly challenging to solve many graph problems in practical amount of time. One of the major challenges

in graph processing relates to the poor locality of reference. In this thesis, we focus on the problem of locality and present two techniques, *access transformation* and *task-set reduction*, to improve the memory locality in graph applications.

In this chapter, we present a brief introduction of graph analytics and its applications. We then discuss the challenges in large graph processing. A brief description of some high performance computing systems is presented. We then present an overview of the dissertation and contributions.

1.1 BIG DATA GRAPH ANALYTICS

In this era of big data, graphs are widely being used to model data. The graph based problems are evolving in multiple disciplines including social networks, transportation, bioinformatics, health care, security and intelligence analysis. The volume of data being represented in graph structure is rapidly increasing. As a result, graph analytics has emerged as a topic of great interest. Graph analytics is applied to uncover insightful relationships between people, places, objects and other entities. The graphs conform to the three Vs associated with big data: volume, velocity and variety. The volume refers to the large amounts of data generated every second. The velocity refers to the speed at which the new data is generated. The variety refers to the different kinds of data both structured and unstructured.

Graphs analytics has large number of applications in various fields. They are used in health care to study the spread of diseases, to detect and prevent epidemics [4]. Graph analysis plays a crucial role in systems biology [5][6]. It is used in the study of protein-protein interaction complex. It is an important tool in understanding the gene expression. Graphs are used to model the gene regulatory networks. Graph analytics is used in finding motifs and patterns in large gene networks. It is used in identifying new protein complexes and for studying and modelling metabolism in various organisms. Graphs are used to represent social networks. The range of applications of social network analysis is rapidly growing. It is used in studying the spread of information and influence [7][8]. It is used for targeted advertising [9], recommender system development [10], and community detection [11]. It is also used in intelligence, in anomaly detection, for example, in uncovering terrorist networks [12].

1.2 RESEARCH CHALLENGES IN LARGE GRAPH PROCESSING

Graph analytics is emerging as a powerful tool to extract value from big data. However, there are many difficult challenges to be addressed. The irregular nature of the graph processing applications makes it difficult to efficiently utilize the computational resources. As data from different domains are mapped to the graph model and as the scale of the data continues to grow, the graph problems outgrow the current computational and memory capabilities of sequential processors. It is essential to use the parallel computing resources to solve problems of large scale. Unfortunately, it is not straight forward to directly map the graph problems to the parallel hardware. Again, the irregular structure of the graph makes it highly difficult to parallelize the graph problems. The techniques that work for regular scientific applications may not be suitable to solve the graph problems. In this section, we discuss the problems involved in large graph processing.

1.2.1 POOR LOCALITY OF REFERENCE

Locality of reference is a fundamental principle of computing. It is the principle behind the caching technique that is used to improve computer system performance. There are two kinds of locality: temporal and spatial. Temporal locality is based on the idea that when some data/instruction is referred to, it is likely that it will be referred to again within a small duration. Spatial locality is based on the idea that when data/instructions are accessed, it is likely that nearby data/instructions will be accessed. Any data that an application needs to access is accessed through cache memory. If the data is not present in cache memory, it is first brought to cache memory from the global memory(main memory) and then accessed. However, if the data is present in cache memory it can be accessed directly from cache. The access to cache is orders of magnitude faster than access to main memory. Therefore, it is important for an application to exploit the cache memory to improve its performance.

An important factor that adversely affects the performance of a graph application is poor locality of reference. A graph application, typically, proceeds by visiting vertices. Visiting a vertex refers to accessing its adjacency list or some other data related to the vertex. The order in which the vertices are visited is very random in nature for many graph applications. Therefore, those graph applications tend to have highly random memory access pattern. By random access we mean access to

memory addresses that are not sequential and that are not to the same address or adjacent addresses. The principle of locality might not be applicable when the access pattern is random. The cache utilization for many graph applications is low when processing large graphs, resulting in high data access time. Therefore, it is hard to extract good performance from such graph applications, even on serial computers.

The random access pattern also makes it difficult for the hardware prefetcher to work efficiently. Most modern processors are equipped with hardware prefetchers. The purpose of a prefetcher is to bring the data from the memory into the cache before they are needed. When the application needs to access data that has been prefetched, it can directly access it from the cache instead of waiting for it to be loaded from main memory. The prefetching mechanism can result in significant performance improvement as it reduces the number of cache misses. A prefetcher works by monitoring the data access pattern and predicting which data will be accessed. However, in the case of many graph applications, since the memory access pattern is random, it is often difficult for the prefetchers to predict which data will be accessed.

1.2.2 LOW COMPUTATION TO DATA ACCESS RATIO

Many graph applications only explore the vertices and edges of the graph without performing large computations using the accessed data. The applications spend most of the time accessing the data and there is very little computation performed on the accessed data. In applications involving large amount of computations it is often possible to hide the memory latency by overlapping the data access with computation. However, in case of aforementioned graph applications, since the computation to data access ratio is low, it is difficult to hide the memory latency using computation.

1.2.3 SEQUENTIAL NATURE OF ALGORITHMS

Most traditional graph algorithms are sequential in nature. The outcome of an iteration/task influences the following iteration/task. So the iterations/tasks can only be executed in sequence. It is difficult to efficiently map the traditional algorithms to the current parallel systems. Therefore, for many graph problems new parallel algorithms have been developed that can take advantage of the parallel resources. However, often these algorithms come at the expense of increased computational complexity. In most cases, the parallel implementations fail to outperform the most efficient sequential implementations.

1.2.4 LOAD BALANCING

Another major challenge in developing parallel graph applications is load balancing. For most graph problems, the computations are data-driven. The computations performed by an application is dictated by the graph structure. For efficient parallelization the tasks must be fairly distributed among the available computing units. In the case of graph algorithms, it is difficult to predict the work load corresponding to a task. The work load is not known until the data assigned to a task is accessed. Adding to this, many graphs in real world networks are irregular and have highly unbalanced degree distribution. This poses additional challenges in achieving load balanced partitioning.

1.2.5 SYNCHRONIZATION AND COMMUNICATION COST

The parallelism in most graph applications is fine-grained. It is difficult to divide a task into multiple tasks that can work independently. Often the tasks need to communicate and synchronize. In multicore architecture, the tasks communicate by reading and writing data into common segments of shared memory. Each core has one or more private caches. When multiple tasks are accessing the same data, it is possible to have copies of the data in multiple caches. To maintain consistency of the data, cache coherency protocols are used. The cache coherency protocols can severely impact the scalability of the application.

Synchronization is essential for the correctness of the algorithms. However, heavy synchronization can result in degraded performance. Synchronization is achieved in multicore architecture using barriers, locks and atomic operations. These constructs are expensive and often result in temporarily blocking the tasks. Therefore, it is necessary to carefully design applications so that the impact of cache coherency protocols and synchronization is minimized. In distributed memory systems, the tasks communicate by passing messages. Since most graph applications exhibit fine-grained parallelism, the communication cost can be very significant.

1.3 PARALLEL ARCHITECTURES

With the current scale of the graphs and the rate at which it is growing, it is becoming inevitable to use parallel computing. The current graphs are very large, with number of vertices and edges ranging from several millions to billions. The

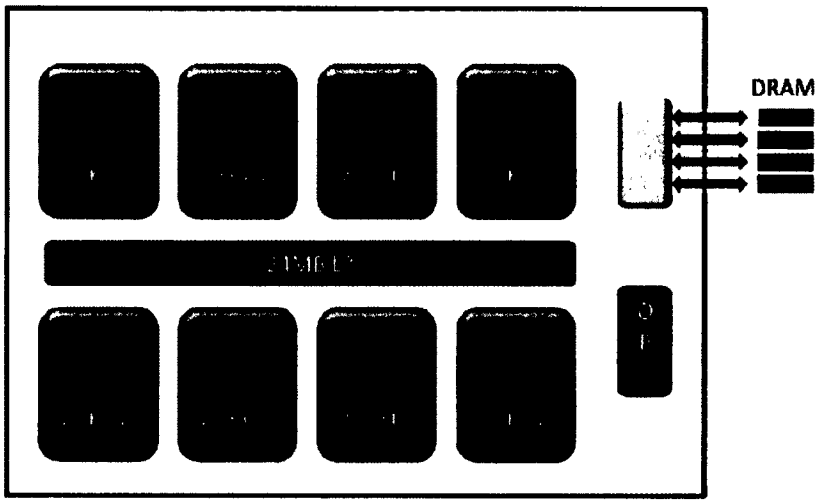
current workstations are incapable of processing such large graphs in practical time due to physical memory limitations and also the processing capacity. In recent years, there have been significant advance in parallel computing capabilities. There are different types of parallel architectures currently available. Based on the memory accessibility, they can be classified into shared memory and distributed memory architectures.

1.3.1 SHARED MEMORY ARCHITECTURE

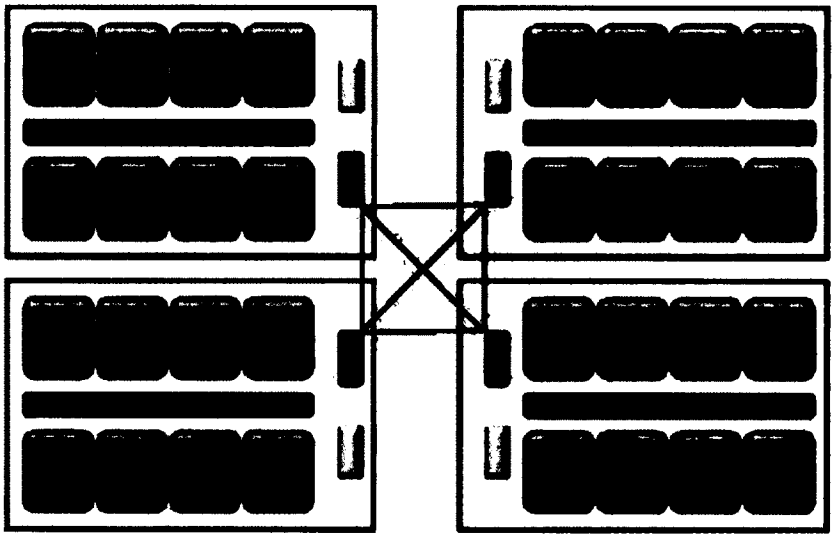
In shared memory architecture all the processors have access to all memory as global address space. The processes can run in parallel on multiple processors/cores and they communicate by reading from and writing into the shared memory. Multicore processors are the most commonly used systems with shared memory architecture. Over the past few decades, there has been a consistent improvement in the computational power of the hardware. According to Moore's law, the number of transistors in an integrated circuit has doubled approximately every two years. However, due to heating issues and increased power consumption, it has become impractical to follow the phenomenon. Instead, the industry has moved toward multicore processors, which contain multiple computing units called cores integrated on a single chip.

Multicore processors typically have multiple levels of caches and may also have a shared cache. In this dissertation, most of the results are obtained using Intel Xeon X7560 processor. It has four sockets and each socket has eight cores. Figure 1 shows an overview of the architecture. Each core has a 32KB L1 data cache and 256KB L2 cache. Additionally, there is also an L3 cache of size 24MB which is shared by all the cores in a socket. All the sockets are connected using a high speed QuickPath Interconnect which gives over a 100GB/sec bandwidth. IMC and QPI in the figure refer to integrated memory controller and QuickPath Interconnect respectively.

Shared memory machines are often classified into uniform memory access and non-uniform memory access(NUMA), based on the memory access times. In uniform access, all the processes take equal time to access the memory. These are often called as symmetric multiprocessors(SMP). In NUMA architecture, the memory access time depends on the memory location relative to the processor. These are often made by linking multiple SMPs like the X7560 architecture shown in Figure 1. Each core has access to memory in all the sockets but the access to non-local memory is



(a) Single Socket



(b) Four sockets connected using Quick Path Interconnect

Figure 1: Architectural overview of Intel Xeon X7560 processor

slower. OpenMP and Pthreads are the most common programming interfaces used for developing programs for shared memory architecture.

1.3.2 DISTRIBUTED MEMORY ARCHITECTURE

Distributed memory architecture constitutes multiple processors connected by a communication network. Each processor has its own local memory. Memory addresses of one processor do not map with another processor. Interprocess communication is achieved using message passing. The individual processors in a distributed architecture are usually made of commodity hardware.

MPI is the most commonly used interface for distributed computing. In distributed memory machines, the users are responsible to distribute the data among the processors and assigning tasks to the processors. When a processor needs to access the data in another processor, it is usually the task of the programmer to define how the data will be communicated. As the processors are connected by a network, the remote access time can take much longer time than the local access time, based on the network. The communication time is an important aspect to be considered when developing applications for distributed systems. The major advantage of distributed systems is its scalability. The computing and memory capacity of a distributed system can easily be extended by adding more processors. However, as the number of processors in a distributed system increases, so does the chance of hardware, software or network failures. Since programming on distributed systems can be an arduous task, there are many frameworks developed to make the task of programming easier. These frameworks provide features like fault tolerance, load balancing, scalability and reliability. The most popular frameworks include MapReduce [13], Spark [14], Pregel [15] and Graphlab [16]. Among these Pregel and Graphlab are developed for graph processing.

1.4 OVERVIEW OF DISSERTATION AND CONTRIBUTIONS

The performance of many graph processing applications is dominated by memory access time. Therefore, it is critical to exploit the data locality to improve the performance. Graph processing application have highly random memory access pattern. When the graph size is large, random access to large data results in frequent cache misses, resulting in degraded performance. Optimizing the data locality is well studied for regular applications like linear algebra. Blocking is a well known optimization

technique that improves the effective cache utilization [17]. The idea of blocking is to organize the memory accesses such that a small subset of data is loaded into the cache and is used/reused. It ensures that the data remains in cache across multiple accesses. The technique has been proven to be very effective for regular applications.

In this dissertation, we propose two techniques to improve the locality in graph algorithms: *access transformation* and *task-set reduction*. *Access transformation* technique changes the random memory access pattern in an application to more of a sequential access pattern. In many graph algorithms, the vertices/edges are systematically processed based on some property. In each iteration, a subset of vertices are processed. Most often the vertices are processed in a random order and the order is determined by some data structure. The key idea of *access transformation* technique is to use *scan-and-extract* operation in each iteration which sequentially scans all the vertices and extracts the vertices that are to be processed in a given iteration. The order of vertices obtained by the *scan-and-extract* operation is more likely to result in sequential access improving the spacial locality of the algorithm. We show the applicability of the technique using the k-core decomposition algorithm and triangle listing algorithm.

The *task-set reduction* technique focuses on improving the temporal locality. We define *task-set* as the collection of data that is repeatedly accessed to process a task. If the size of *task-set* is very large, then repeated random accesses to the data in *task-set* can result in large number of cache misses, severely impacting the performance. It is, therefore, very crucial to keep the *task-set* to a minimal size. *Task-set* reduction can be achieved in different ways like compression, blocking and elimination. Compression refers to storing the data using minimal amount of memory. By carefully examining the nature of the data and possible values of the data, it is often possible to reduce the size of the data structure that stores the data. For example, representing data in bit format can result in significant reduction in *task-set* size. The blocking technique [17] used for regular applications can also be used for *task-set reduction*. The idea is to partition the tasks such that each task works on a smaller *task-set*. However, unlike in regular applications, it may not be easily applicable. It might require significant changes to the algorithm to utilize blocking technique. Elimination refers to disposing of data structures which store data that can be extracted using other sources/data structures.

In this dissertation, we propose algorithms using these techniques for three graph

problems: k-core decomposition, maximal clique enumeration and triangle listing. The k-core of a graph is the largest induced subgraph with minimum degree k. The largest value of k that a vertex belongs to a k-core is called *core number* of the vertex. The k-core decomposition problem is to find the *core number* of all the vertices in a graph. It has applications in many areas including network analysis, computational biology and graph visualization. The primary reason for it being widely used is the availability of an $O(n+m)$ algorithm. The algorithm was proposed by Batagelj and Zaversnik [18] and is considered the state-of-the-art algorithm for k-core decomposition. However, the algorithm is less suitable for parallelization and to the best of our knowledge there is no algorithm proposed for k-core decomposition on multicore processors. Also, the algorithm has not been experimentally analyzed for large graphs. In Chapter 2, we present an experimental analysis of the algorithm of Batagelj and Zaversnik and propose a new algorithm, *ParK*, that uses the *access transformation* technique to improve the memory locality. We provide an experimental analysis of the algorithm using graphs with up to 65 million vertices and 1.8 billion edges. We compare the *ParK* algorithm with state-of-the-art algorithm and show that it is up to 6 times faster than the state-of-the-art algorithm. We also provide a parallel methodology and show that the algorithm is amenable to parallelization on multicore architecture. We present experimental results obtained using a 4 socket Nehalem-EX processor which has 8 cores per socket which show that the algorithm scales up to 21 times using 32 cores.

A clique in a graph is a subgraph in which every pair of vertices is connected by an edge. A maximal clique is a clique which is not contained in any other clique. Maximal clique enumeration(MCE) problem is to find all the maximal cliques in a graph. MCE is a fundamental problem in graph theory. It plays a vital role in many network analysis applications and in computational biology. MCE is an extensively studied problem [60][61][32][65][69]. Recently, Eppstein et al. [32] proposed a state-of-the-art sequential algorithm that uses degeneracy based ordering of vertices to improve the efficiency. In Chapter 3, we present an analysis of *task-set* size of Eppstein et al.'s algorithm. We propose a new algorithm using the *task-set reduction* technique. The new algorithm uses a new bit-based data structure. The new data structure not only reduces the *task-set* size significantly but also improves the performance of the algorithm by enabling the use of bit-parallelism. We illustrate the significance of degeneracy ordering in load balancing and experimentally evaluate the impact of

scheduling on the performance of the algorithm. We present experimental results on several types of synthetic and real-world graphs with up to 50 million vertices and 100 million edges. We show that our approach outperforms Eppstein et al.'s approach by up to 4 times and also scales up to 29 times when run on a multicore machine with 32 cores. We have also implemented the new algorithm on distributed architecture and the experimental results show that the algorithm scales well, upto 106 times using 128 processes.

A triangle in a graph refers to a clique of size 3. Triangle listing problem is to find all the triangles in a graph. The triangle counting/listing problems are of high interest in network analysis applications. They are primarily used in finding a key statistical property of a graph called clustering coefficient. Many algorithms for the triangle listing problem exist in the literature [92][93][94]. Out of those, the *edge-iterator* algorithm [93] is the most widely used algorithm. The algorithm repeatedly accesses the adjacency lists of the vertices and in random order resulting in poor memory locality. In Chapter 4, we propose a new algorithm, called *window-iterator* that uses the *access transformation* and *task-set reduction* techniques to improve the locality. Unlike, the *edge-iterator* algorithm, the *window-iterator* algorithm has limited number of iterations, each iteration working on a smaller *task-set*. The *window-iterator* outperforms the *edge-iterator* algorithm for large graphs(upto 1.4 times) and the gap increases as the graph size increases. We have implemented the approach for multicore architecture. Our experimental results show that the new algorithm scales well, more than 29 times using 32 cores.

CHAPTER 2

ACCESS TRANSFORMATION

Graphs are ubiquitous. There are many interesting and complex graph problems that have applications in different domains. Graph problems have been well studied and there exist efficient algorithms for most of the problems. The main focus of the traditional algorithms was to reduce the computational complexity. Though many algorithms are NP-hard, since the graphs were small in size, the problems were solvable in practical amount of time. However, since the graph problems are being applied in a variety of domains and as the graph sizes are rapidly growing, it is becoming increasingly important to redesign traditional algorithms considering various other factors like memory access pattern, data structures and memory bandwidth.

One of the major factors governing performance of graph algorithms is poor locality of reference. The memory access pattern in most graph algorithms is highly random. The random access pattern coupled with the large size of the graphs results in poor utilization of the cache memory. Most of the data accesses result in cache misses and the data has to be accessed from the main memory which has greater latency, degrading the performance of the application.

In this chapter, we present a technique called *access transformation* which improves the locality of reference by changing the memory access pattern. Many graph algorithms are based on systematic exploration of the graphs. They traverse the nodes in some order that is specific to the problem. For example, in BFS, the nodes are traversed based on their distance from the source node, in k-core decomposition algorithm, the nodes are traversed based on their degree. The algorithms explore the graphs in multiple iterations, processing a subset of nodes in each iteration. However, the order of vertices processed in an iteration is mostly random which can severely impact the performance.

To show the impact of random access, we ran an experiment which performs read and writes on all the elements in an integer array in random order. The graph in Figure 2 clearly shows that the memory latency increases as the data size increases. The array size is shown in millions(m) and billions(b).

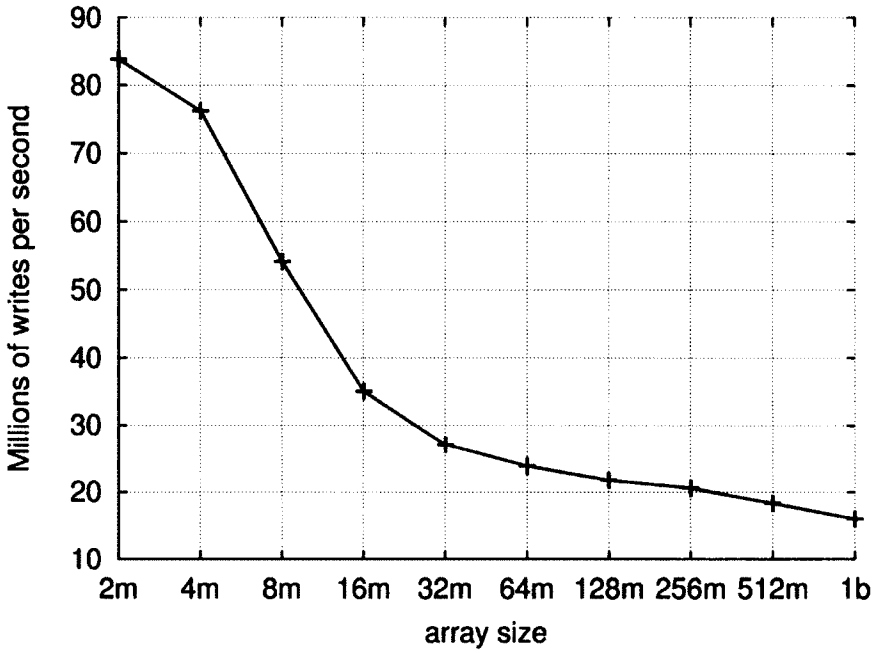


Figure 2: Plot showing the memory latency for random writes

The *access transformation* technique refers to changing the random access pattern to sequential access. In some algorithms it may be straight forward to achieve it while in some cases it may be required to redesign the algorithm. Often, some data structure is used to store problem specific data such as the vertices to be processed in the next iteration, the order of vertices, or the distance of a vertex from a source vertex. This data structure, in general, governs the access pattern of the next iteration. The main idea behind *access transformation* technique is to use *scan-and-extract* operation to extract the data when required instead of storing it in a data structure. By eliminating the random access causing data structure, the operation results in a relatively more sequential access pattern thus improving the performance of the application.

The *scan-and-extract* operation reduces the random nature of the access pattern. However, it comes at the expense of increased number of operations which are required for the scan operation. As, the scan operation is performed in each iteration, if the number of iterations is too high, the overall time taken for the scan operation can be significant and might result in degraded performance. Therefore, the number

of iterations and the cost of *scan-and-extract* operation must be considered when applying the *access transformation* technique.

The technique is inspired from the paper [19] which proposes a new read-based algorithm for BFS. The traditional BFS algorithms are queue-based in which a queue is used to store the vertices to be processed. Processing a vertex involves accessing its adjacency list(array) and adding its neighbors to the queue if they have not already been processed. The vertices are added in random order to the queue resulting in a highly random access pattern. The new algorithm eliminates the use of queue by scanning all the vertices to extract the vertices that are to be processed in the current iteration. Since the vertices extracted are not in random order, the algorithm improves the opportunity of sequential access pattern and is shown to outperform the traditional BFS algorithms.

The *access transformation* technique is applicable to algorithms with limited number of iterations that process a subset of vertices in each iteration. The applicability of the technique is, nevertheless, not limited to those algorithms. It is a more general approach and can also be applied in other scenarios. However, the algorithms might need to be redesigned to limit the number of iterations. For example, the triangle listing problem has highly random access pattern but the algorithm has large number of iterations. In Chapter 4, we show how the algorithm can be redesigned to limit the number of iterations facilitating the use of *access transformation* technique and also the *task-set reduction* technique discussed in the next chapter.

There is also an added advantage to the *access transformation* technique. The technique, in general, results in an approach that is amicable to parallelization. Since the random access pattern limits the benefits of parallelism, by transforming the access pattern to sequential the technique results in a better scalable approach. We have seen that the *access transformation* technique can be used at the expense of increased cost of *scan-and-extract* operation. However, the *scan-and-extract* operation is embarrassingly parallel and the computational time for the operation can be greatly reduced by using parallelism.

In the rest of the chapter, we show how the *access transformation* technique can be applied to the k-core decomposition algorithm. We first present, in Section 2.1, some definitions and notations and then in Section 2.2 explain in detail the state-of-the-art algorithm for k-core decomposition focusing on its memory access pattern. In Section 2.3, we discuss how the *access transformation* technique can be used for

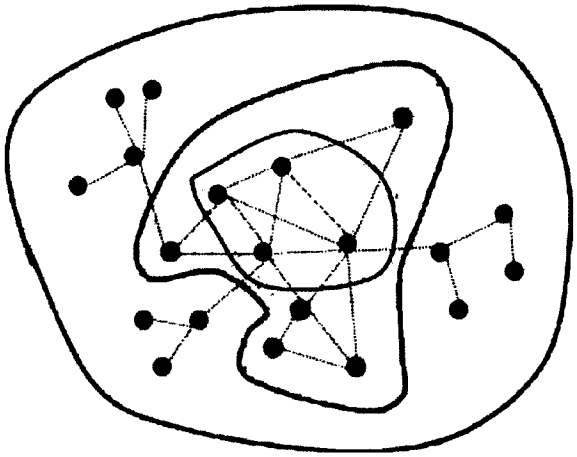


Figure 3: An example graph showing different cores

this problem and present a new algorithm, called *ParK*, that adopts the technique. The parallel methodology of *ParK* algorithm is discussed in Section 2.4. In Section 2.5, we present the experimental results using various graphs from different datasets and show that the new algorithm outperforms the start-of-the-art algorithm by upto 6 times and the performance gap becomes larger as the graph size grows. Also the new algorithm is scalable resulting in speed-up of upto 21 time using 32 cores.

2.1 K-CORE DECOMPOSITION

The k-core of a graph is the largest induced subgraph with minimum degree k. The notion of a core was first introduced in 1983 by Seidman et al. [20]. Since then, it has been extensively studied and used in applications in many areas including network analysis, computation biology and visualization. k-core has been primarily applied in identifying the cohesive subgroups in a network. Many notions can be considered for identifying such groups including cliques, k-plexes, n-cliques [21]. While most other approaches are computationally expensive, k-core decomposition can be computed in linear time.

2.1.1 DEFINITION AND NOTATIONS

Let $G = (V, E)$ be a graph where V is the set of vertices and E is the set of edges and let $n = |V|$ and $m = |E|$. The k-core of the graph G, is the largest

induced subgraph in which every vertex has degree at least k. The *core number* or *coreness* of a vertex v, is the largest value of k for which v belongs to the k-core i.e. $core(v) = max\{k|v \in k\text{-core}\}$. Note that $(k+1)$-core is a subset of k core. A k-shell of a graph G is the subgraph induced by the set of vertices in G whose core number is k, i.e. the vertices that belong to k-core but not $(k+1)$-core. In Figure 3, all the vertices inside the blue, green and red boundaries belong to 1-core, 2-core and 3-core respectively. The vertices colored in blue, green and red belong to 1-shell, 2-shell and 3-shell respectively and have core numbers 1, 2 and 3 respectively. The problem of k-core decomposition of the graph is to find all the k-cores of the graph or in other words, find the core numbers of all the vertices in the graph. In the rest of the chapter, we use $N(v)$ to denote the neighborhood of v (note that $v \notin N(v)$) and $core(v)$ to denote the core number of v. The degree of vertex v is denoted by $deg(v)$. n and m denote the number of vertices and edges in the graph, respectively.

2.1.2 APPLICATIONS

k-core decomposition has been used in analyzing and understanding the internet topology [22][23]. It has been used in the study of influential spreaders in complex networks [24]. It was shown that the most efficient spreaders are those located within the core of the network. It was used in detecting dense communities in large, social networks [25][26]. k-core decomposition is considered as an important tool in visualization [27][28][29]. In computational biology it was used in analyzing and detecting protein interactions [30] and analyzing gene networks [31]. k-core decomposition is used as a pre-processing step in other graph problems like finding maximal and maximum cliques [32][33]. It is considered an important tool in network analysis and is included in network analysis packages [34][35][16].

2.2 RELATED WORK

The algorithm for computing the core numbers of all the vertices in the graph is shown in Figure 4. It involves repeatedly removing the minimum degree vertices from the graph. The degree of the vertex when it is being removed is the core number of the vertex. The function *getSmallestDegreeVertex* in Figure 4 returns the smallest degree vertex in the remaining graph. The main challenge in k-core decomposition is to find, in each iteration, the minimum degree vertex in the remaining graph. A simple way is to scan the degrees of all the vertices in the graph. Clearly, this method

```
1: while G is not empty do
2:     v = getSmallestDegreeVertex(G)
3:     core(v) = deg(v)
4:     for each vertex u ∈ N(v) do
5:         decrement deg(u) by 1
6:     end for
7:     delete v from G
8: end while
```

Figure 4: The k-core decomposition algorithm outline

is highly inefficient. Another method is to maintain a sorted list of vertices. This is the idea behind the BZ algorithm explained in the next section.

2.2.1 THE BZ ALGORITHM

The algorithm for k-core decomposition that is considered state-of-the-art is proposed by Batagelj and Zaversnik (we refer to it as BZ algorithm in the rest of the chapter). It is a linear time algorithm. It solves the problem of finding the minimum degree vertex in the graph by maintaining a list of vertices sorted in increasing order of degree. In each iteration, when a vertex is processed and the degree of its neighbors is reduced, the neighbors are moved to the appropriate position in the sorted list. The algorithm uses count sort to compute the initial sorted list.

Figure 5 shows the BZ algorithms. The algorithm takes the graph G as input. It uses four arrays: deg, vert, pos and bin, to keep tract of their degree and the order of the vertices to be processed. The arrays *deg*, *vert*, *pos* are of size n, where n is the number of vertices in the graph. The array *bin* is of size $M + 1$ where M is the maximum degree of the graph. The function *getSortedArray* takes the graph as input, It initializes the four arrays and returns them as output. The array *deg* is initialized to contain the degree of the vertices in the graph. Note that the vertices are numbered from 0 to $n - 1$. $deg[i]$ represents the degree of vertex i. The *vert* array contains all the vertices in the graph in increasing order of degree. The *pos* array stores the positions/indexes of vertices in *vert* array. For example, if vertex u is at index i in *vert* array i.e $vert[i] = u$, then $pos[u] = i$. The sorted array *vert* can be viewed as an array of bins, each bin containing a set of vertices of same degree. So, the array *vert* is nothing but bin of degree 0 vertices followed by bin of degree 1

```
1: procedure kcoreBZ(G)
2:     (deg, vert, pos, bin) = getSortedArray(G)
3:     for i = 0 to n − 1 do
4:         v = vert[i]
5:         for each vertex u ∈ N(v) do
6:             if deg[u] > deg[v] then
7:                 pu = pos[u]
8:                 pw = bin[deg[u]]              ▷ get the position of first vertex in bin
9:                 w = vert[pw]                              ▷ get the first vertex in bin
10:                if u ≠ w then
11:                    vert[pu] = w      vert[pw] = u          ▷ swap u and w in vert
12:                    pos[w] = pu       pos[u] = pw   ▷ update their positions in pos
13:                end if
14:                increment bin[deg[u]] by 1
15:                decrement deg[u] by 1
16:            end if
17:        end for
18:    end for
19: end procedure
20: function getSortedArray(G)
21:    maxDeg = 0
22:    for i = 0 to n − 1 do
23:        deg[i] = getDegree(i, G)
24:        if deg[i] > maxDeg then maxDeg = deg[i] end if
25:    end for
26:    for i = 0 to maxDeg do bin[i] = 0 end for
27:    for i = 0 to n − 1 do increment bin[deg[i]] by 1 end for
28:    start = 0
29:    for i = 0 to maxDeg do
30:        num = bin[i]
31:        bin[i] = start
32:        increment start by num
33:    end for
34:    for i = 0 to n − 1 do
35:        pos[i] = bin[deg[i]]
36:        vert[pos[i]] = i
37:        increment bin[deg[i]] by 1
38:    end for
39:    for i = maxDeg down to 1 do bin[i] = bin[i − 1] end for
40:    return (deg, vert, pos, bin)
41: end function
```

Figure 5: The algorithm of Batagelj et al.

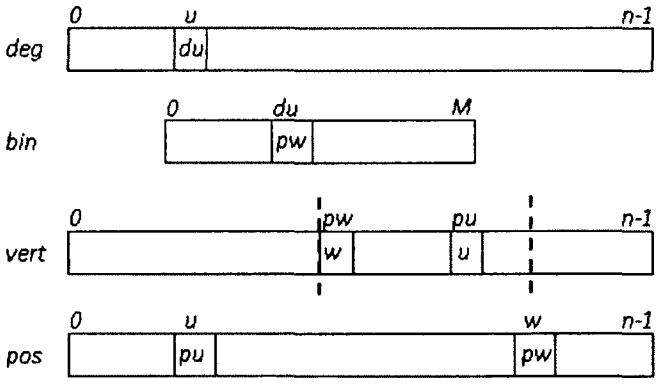


Figure 6: A figure showing the memory accesses required for an iteration in BZ algorithm

vertices and so on. The array *bin* contains the index of the first vertex in each bin i.e. $bin[i]$ contains the index of the first vertex with degree i in the *vert* array.

The function *getSortedArray* uses count sort to compute the sorted array *vert* which can be computed in $O(n)$ time. The function *getDegree* in line 23 returns the degree of a vertex. Once the initial four arrays are returned by the *getSortedArray* function, the vertices are processed one at a time in the order given in *vert* array. Processing a vertex v involves reducing the degree of all its neighbors by 1 and updating the *vert*, *pos* and *bin* arrays accordingly. Whenever, the degree of a vertex is decremented (by 1) it needs to be moved to appropriate bin (to the preceding bin). To achieve that, the vertex is swapped with the first vertex in the current bin by updating appropriate values in *vert* and *pos* arrays. And then, *bin* array is updated such that the vertex belongs to the preceding bin. The time complexity of the algorithm is shown to be $O(n + m)$.

Memory access pattern in BZ algorithm

We have seen that BZ algorithm provides a linear time solution for k-core decomposition and is considered the state-of-the-art algorithm. However, we observe that the memory access pattern of the algorithm is more random in nature. As can be seen in Figure 5, the algorithm uses four arrays *deg*, *vert*, *pos* and *bin*. It can be observed that the memory access pattern of these arrays is highly random. Each iteration requires random read and write access to these arrays. Figure 6 shows the

memory accesses required when a vertex v is processed and its neighbor u's degree is reduced. *du* in the figure denotes $deg[u]$. The operation requires read and write access to $deg[u]$, $vert[pu]$, $vert[pw]$, $pos[u]$, $pos[w]$ and $bin[du]$. Notice that all these data elements are in random positions in the four arrays. While the size of *bin* array may be small, the sizes of the other three arrays can be very large. So there is a high chance that the access to each of these data elements incurs a cache miss and requires access to main memory. The access pattern of the adjacency lists also impacts the performance. It can be seen from Figure 5 that the adjacency list of a vertex is accessed exactly once in the algorithm. The order in which the adjacency lists are accessed is governed by the *vert* array. Though, the accesses are more sequential in the initial iterations, as the vertices are processed and as the *vert* array is updated, the access pattern of adjacency lists becomes more random in nature.

Parallelizing the BZ algorithm

We have seen in Section 2.2.1 that the *vert* array can be viewed as an array of bins. Each bin consists of vertices of same degree. Parallelizing the BZ algorithm for multicore architecture can be done by distributing all the vertices in a bin to the available processing units. Each processing unit processes the vertices assigned to it. Recall that processing a vertex involves reducing the degree of its neighbors and accessing multiple locations in the four arrays (for example, in Figure 6 the locations $deg[u]$, $vert[pu]$, $vert[pw]$, $pos[u]$, $pos[w]$ and $bin[du]$ are accessed). However, since multiple processing units are sharing the four arrays and reading and writing into the arrays, it might result in race conditions and inconsistent results. To avoid that, synchronization constructs like locks need to be used. For example, in Figure 6, all the data elements $deg[u]$, $vert[pu]$, $vert[pw]$, $pos[u]$, $pos[w]$ and $bin[du]$ should be locked before reading from and writing into them. Due to the high cost of synchronization constructs which are very expensive, the BZ algorithm is not amicable to parallelization.

2.2.2 DISTRIBUTED ALGORITHM

There is a distributed algorithm proposed for k-core decomposition [36]. Each process is assigned a set of vertices and is responsible for calculating the core numbers of the vertices assigned to it. If there are p processes, numbered 0 to $p-1$, then a vertex i is assigned to the process numbered $i \ mod \ p$. For each vertex assigned to

a process, it stores an array consisting of core numbers of its neighbors. The core number of each vertex is initialized to the degree of the vertex and is updated during the decomposition process. The algorithm is based on the idea that the core number of a vertex can be calculated based on the core number of its neighbors according to the following theorem:

Theorem: For each vertex $u \in V$, $core(u) = k$ if and only if

1. there is a subset $V_k \subseteq N(u)$ such that $|V_k| = k$ and $\forall v \in V_k : core(v) \geq k$
2. there is no subset $V_{k+1} \subseteq N(u)$ such that $|V_{k+1}| = k + 1$ and $\forall v \in V_{k+1} : core(v) \geq k + 1$

The core number of the vertices is calculated using the core number of their neighbors. Once the core number of a vertex is updated it is communicated to the other processes to which its neighbors are assigned. This is repeated until core number of none of the vertices is updated. Unlike the BZ algorithm, in which the adjacency list of a vertex is accessed only once, the adjacency list of a vertex is accessed multiple times i.e. whenever the core number of any of its neighbors is updated. This approach results in significant increase in number of operations. Therefore, it is more suitable to a distributed environment where there are large number of computing nodes and each computing node is assigned only a few vertices. However, this approach is less suitable for multicore architecture with only a limited number of threads and each thread is assigned large number of vertices. Since each thread has to repeatedly access the adjacency lists of the vertices assigned to it, the working set for the thread consists of all its vertices and their neighbors which can be too large to fit in cache memory thus resulting in poor locality of reference.

There are approaches proposed for for k-core decomposition of dynamic networks [37][38][39]. These approaches primarily focus on efficiently maintaining the core numbers of the vertices as the graph changes over time.

2.3 *PARK* ALGORITHM

The k-core decomposition problem is a good candidate for applying the *access transformation* technique. It uses different arrays (*vert*, *pos*, *bin*) that result in random memory access pattern. Though the original approach, i.e. BZ, requires n iterations, it can easily be modified to limit the number of iterations to maximum degree, M. We propose a new algorithm for k-core decomposition called *ParK* (parallel

```
1: procedure kcoreParK(G)
2:     todo = n
3:     level = 0
4:     for i = 0 to n − 1 do deg[i] = getDegree(i, G) end for
5:     while todo > 0 do
6:         curr =scan-and-extract(deg, level)
7:         while |curr| > 0 do
8:             decrement todo by |curr|
9:             next = processSublevel(curr, deg, level)
10:            curr = next
11:        end while
12:        increment level by 1
13:    end while
14: end procedure
15: function scan-and-extract(deg, level)
16:    curr = ∅
17:    for i = 0 to n − 1 do
18:        if deg[i] = level then
19:            add i to curr
20:        end if
21:    end for
22:    return curr
23: end function
24: function processSublevel(curr, deg, level)
25:    next = ∅
26:    for each vertex v in curr do
27:        for each vertex u adjacent to v do
28:            if deg[u] > level then
29:                decrement deg[u] by 1
30:                if deg[u] = level then
31:                    add u to next
32:                end if
33:            end if
34:        end for
35:    end for
36:    return next
37: end function
```

Figure 7: *ParK* algorithm

k-core) which adapts the *access transformation* technique. The algorithm eliminates the use of the three arrays (*vert*, *pos* and *bin*) and uses *scan-and-extract* operation to get the list of vertices to be processed in an iteration.

2.3.1 DESCRIPTION

The *ParK* algorithm processes the vertices in levels. Processing a vertex refers to accessing its adjacency list and reducing the degree of its neighbors that have not already been processed. In level i, all the vertices in shell i are processed (recall that k-shell contains all the vertices that belong to k-core but not $k+1$-core). The *ParK* algorithm is based on the idea that instead of maintaining a sorted array of vertices, we can generate the array at each level. The outline of the *ParK* algorithm is given in Figure 7. The algorithm uses three arrays *deg*, *curr* and *next*. The *deg* array is initialized to contain the degree of the vertices similar to the BZ algorithm. Note that the vertices in the graph are numbered from 0 to $n-1$. $deg[v]$ contains the degree of vertex v. As the vertices are processed and degree of their neighbors are reduced, the *deg* array is updated. The *deg* array at the end of *kcoreParK* procedure contains the core numbers of all the vertices in the graph.

Processing a level l is done in two phases: scan phase and loop phase. In scan phase (which is same as *scan-and-extract* operation in *access transformation* technique), the *deg* array is scanned and the vertices that are to be processed in the current level are extracted into *curr* array. Note that the vertices in *curr* are in sequential order i.e. if $u = curr[i]$ and $v = curr[j]$ where $i < j$ then $u < v$. The scan phase is performed using *scan-and-extract* function given in Figure 7. The loop phase consists of one or more sub-levels (or iterations). In each sub-level, all the vertices in *curr* array are processed. When processing a vertex v in *curr*, if any neighbor vertex, u, is moved to the current level i.e. $deg[u]$ is reduced to l, then u is added to *next* array. At the end of each sub-level, the contents of *next* are transferred to *curr* so that they can be processed in the next sub-level. The lines 7 through 11 in Figure 7 correspond to the loop phase. The *processSublevel* function in Figure 7 processes the vertices in current sub-level and returns the array of vertices that are to be processed in the next sub-level.

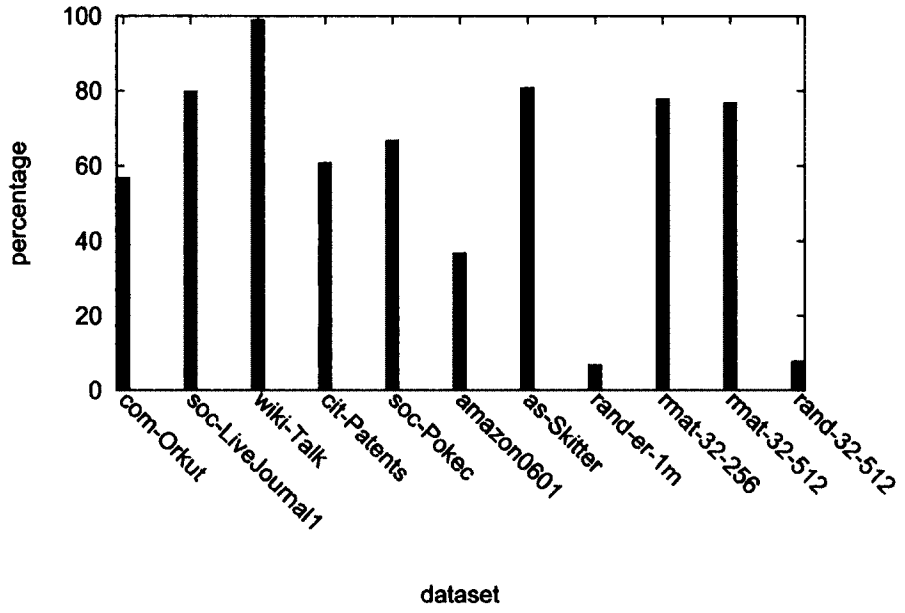


Figure 8: A plot showing the percentage of vertices processed in first sub-level of all the levels

2.3.2 MEMORY ACCESS PATTERN OF THE *PARK* ALGORITHM

As can be seen in Figure 7, the *ParK* algorithm uses three arrays *deg*, *curr* and *next* arrays. The read and write accesses to *curr* and *next* arrays is sequential. Also, the access to *deg* array in *scan-and-extract* function is sequential. However, the loop phase requires random access to the *deg* array (line 28 in Figure 7). Note that, BZ algorithm also performs the same random accesses to the *deg* array.

The access pattern of the adjacency lists in *ParK* algorithm is governed by the order in which vertices are added to the *curr* and *next* arrays. In the *scan-and-extract* function, the order of vertices added to the *curr* array is guaranteed to be sequential. Therefore, the first sub-level in each level of the algorithm results in sequential access of the adjacency lists. However, in subsequent sub-levels there is no such guarantee and the adjacency lists are accessed in random order. From our experiments, we observe that, for most graphs, the majority of vertices are processed in first sub-level. The plot in the Figure 8 shows the percentage of vertices processed in first sub-level in all the levels.

2.3.3 ANALYSIS OF THE ALGORITHM

In this section, we analyse the time complexity of the *ParK* algorithm.

Lemma 1. The maximum number of levels in *ParK* algorithm is k_{max}, where k_{max} is the largest value of k for which k-core is present in the graph.

Proof: The algorithm uses a variable *todo* to keep track of the number of vertices to be processed. It can be seen from the procedure *kcoreParK* in Figure 7 that *todo* is decremented in each sub-level by the number of vertices processed in the sub-level. Therefore, *todo* always contains the count of number of vertices to be processed. Since at the end of level k_{max} all the vertices have been processed and the value of *todo* is zero, the maximum number of levels is k_{max}.

Lemma 2. The combined time taken for scan phase in all the levels is $O(k_{max}n)$

Proof: The scan phase in each level takes $O(n)$ and since there are k_{max} levels, the combined scan time for all the levels is $O(k_{max}n)$

Lemma 3. The combined time taken for loop phase in all the levels is $O(m)$

Proof: In a single call to *processSublevel* a subset of vertices is processed. Combinedly in all the calls to *processSublevel*, all the n vertices are processed. Note that, each vertex is processed exactly once i.e. when its degree becomes equal to the current level. We have seen that processing a vertex v includes reducing the degree for each of its neighbor if it has not already been processed which takes $O(d_v)$ time where d_v is the degree of vertex v. Therefore, to process all the n vertices it takes $O(m)$ time.

Combining lemmas 2 and 3 (and $O(m)$ time for line 4), the computational complexity of *ParK* algorithm is $O(k_{max}n + m)$. Though the computational complexity of BZ algorithm, which is $O(n+m)$, is less compared to the *ParK* algorithm, our experimental results show that *ParK* algorithm outperforms BZ algorithm. The reason is that by using the scan phase, *ParK* significantly reduces the number of random memory access resulting in better locality of reference. The BZ algorithm requires random read and write accesses to three different arrays of size n while in *ParK* algorithm requires random access to only one array. Though the theoretical time taken for the scan phase seems significant, in practice the time taken for scan phase is less compared to the time saved due to the scan phase. Another major advantage of scan phase is that it is embarrassingly parallel. It can easily and efficiently be distributed among different processors and can scale linearly.

2.4 PARALLEL METHODOLOGY OF *PARK* ALGORITHM

In this section, we describe a level-synchronous approach to parallelizing the *ParK* algorithm. We have seen that the *ParK* algorithm processes the vertices in levels. Since the number of levels is limited, there is generally sufficient degree of parallelism available in each level. Parallelizing the *ParK* algorithm involves individually parallelizing the two phases in the algorithm: scan phase and loop phase. Parallelizing the scan phase is simple and trivial. The n vertices are equally divided among the t threads and each thread scans n/t vertices. Note that distributing the n vertices among t threads can be done in several ways. To minimize cache misses, contiguous chunks of vertices are assigned to the threads. To process each vertex, a thread reads its degree and if it is equal to the current level it adds the vertex to *curr* array. Since there are only a few operations performed for each vertex, the load is well balanced resulting in linear speed-up.

Though, it is simple to parallelize the scan phase, there is one issue to be addressed. It is to be noted that, the array *curr* is shared between all the threads and multiple threads writing to it may result in race conditions. The parallel version of $scan - and - extract$ function that addresses the issue of race conditions is shown in Figure 9. The function $atomicIncrement(idx, 1)$ increments the value of idx by 1 and returns its old value atomically (implemented using *atomic capture* construct in OpenMP). Using atomic operations the race conditions are eliminated. However, the atomic operations are expensive and too many atomic operations can significantly downgrade the performance. To reduce the number of atomic operations, the vertices are added in batches instead of a single vertex. We use a local buffer of size b. Instead of adding each vertex to *curr*, they are first added to the local buffer. When the buffer is full, idx is atomically incremented by b and all the vertices are transferred from local buffer to *curr*. This reduces the number of atomic operations by a factor of b. Note that, to avoid cache invalidation we choose the size of local buffer to be a multiple of cache line size.

The major component in loop phase is the function *processSublevel*. We have seen that in *processSublevel* all the vertices in *curr* are processed i.e. the degree of all their neighbors are reduced and if any of the neighbors belong to the current level, it is added to *next*. Parallelizing the function is done by equally distributing the vertices in *curr* to all the threads. However, it might result in race conditions as multiple threads access the *deg* and *next* arrays. To avoid race conditions, all updates to

these arrays are performed atomically. However, it is possible that the degree of a neighbor u, i.e. $deg[u]$ is reduced to a value less than the current level value. For example, let $deg[u] = level + 1$ and two or more threads execute the line 17 at the same time, test positive for the condition and execute line 18. The resultant value of $deg[u]$ will be less than *level* which the correct value of $deg[u]$ is *level*. This issue is fixed using the lines 19 through 21. Vertices are added to the *next* array in batches similar to the way vertices are added to *curr* in the *scan* phase i.e. using atomic increments and local buffers.

The individual phases are executed by the threads in parallel without much interference (except for the atomic operations). However, after each phase, the threads need to synchronize before beginning the next phase operations. Note that the loop phase consists of multiple sub-levels(iterations) with a call to the *processSub-level* function in each sub-level. Therefore loop-phase is also level-synchronous as the threads synchronize after each sub-level. To synchronize the threads we use OpenMP *barrier* construct. Synchronizing all the threads is an expensive operation. It blocks all the threads until the last thread completes the work assigned. High usage of synchronization constructs can degrade the performance significantly. However, we have found that for most graphs, the number of levels and sub-levels is limited. Therefore the approach in general results in good speed-up. We observe that there is an alternative approach to parallelizing the loop phase which eliminates the need to synchronize after each sub-level. The idea of the approach is that in the first sub-level when a thread encounters a vertex that needs to be added to the *next* array, instead of adding it to the array it stores in a temporary local array and then process the vertex by itself at a later point of time. Consequently, all the vertices that correspond to a level are processed in the first sub-level. We have experimented using both the approaches and noticed that, though the second approach reduces the synchronization cost, it results in severe load imbalance. Therefore, the level-synchronous approach performs better in general compared to the second approach.

2.5 EXPERIMENTAL RESULTS

All the results presented in this section are obtained using a four socket 2.27GHz Xeon X7560(Nehalem-EX) with 256 GB shared memory and running 64-bit Ubuntu 12.04. Each socket consists of 8 cores. Each core has a private 32 KB L1 cache and 256 KB L2 cache. A 24 MB L3 cache is shared by all the cores in a socket.

Table 1: Details of graphs and time(in seconds) taken by BZ and *ParK* Algorithms. n, m and k_{max} denote the number of vertices and edges (both in millions) and the maximum core value of the graphs

					ParK	
graph	n	m	k_{max}	*BZ*	*total*	*scan*
amazon0601	0.4	2.4	10	0.16	0.09	0.005
web-BerkStan	0.6	6.6	201	0.23	0.22	0.11
web-Google	0.9	4.3	44	0.31	0.19	0.03
wiki-Talk	2.4	4.6	131	0.35	0.45	0.25
as-Skitter	1.7	11.1	111	0.84	0.49	0.16
soc-Pokec	1.6	30.6	47	2.29	0.83	0.07
cit-Patents	3.8	16.5	64	3.42	1.44	0.22
rand-er-1m	1.0	95.0	160	4.36	1.85	0.12
com-Orkut	3.1	117.2	253	18.02	5.48	0.64
soc-LiveJournal1	4.8	69.0	362	5.47	3.04	1.42
rmat-32-256	32.0	256.0	29	147.67	24.16	1.86
rmat-32-512	32.0	512.0	59	288.1	41.17	3.65
rand-32-512	32.0	512.0	23	231.38	51.5	1.47
com-Friendster	65.6	1806.0	289	981.58	158.68	36.07

All the implementation is done using C programming language and compiled using gcc compiler with -O3 optimization flag. The parallel implementation is done using OpenMP.

For our experiments we have used several graphs from the Stanford Large Network Collection [40] and also synthetic graphs. The synthetic graphs are generates using GTGraph [41], a graph generator tool. The details of the graphs in the dataset are as follows:

- *amazon0601*: Amazon product co-purchasing network. If a product i is frequently co-purchased with product j, the graph contains and edge from i to j.
- *web-BerkStan*: Berkeley Stanford web graph. Vertices represent pages from berkeley.edu and stanford.edu and edges represent hyperlinks between them.
- *web-Google*: Google web graph. Vertices represent web pages and edges represent hyperlinks between the pages.
- *wiki-Talk*: Wikipedia talk network. Vertices represent users and an edge from vertex i to vertex j indicates that user i edited the page of user j atleast once.
- *as-Skitter*: Internet topology graph i.e graph representing the network topology of the internet.
- *soc-Pokec*, *com-Orkut*, *soc-LiveJournal1*, *com-Friendster*: Social networks. Vertices represent users and the edge represents friendship between users.
- *cit-Patents*: Patent citation network. Vertices represent patents and the edges represent citations.
- *rand-er-1m*: Synthetic graph generated using GTGraph. It is generated using Erdos–Renyi graph model with probability 10^{-3}
- *rmat-32-256*, *rmat-32-512*: Synthetic graphs generated using GTGraph. They are generated using the R-MAT model with default parameter values i.e. $(a, b, c, d) = (0.45, 0.15, 0.15, 0.25)$
- *rand-32-512*: Synthetic graph generated using GTGraph. It graph generated by adding each edge to a randomly chosen pair of vertices.

Additional properties of the graph, i.e number of vertices, edges and maximum core value, k_{max}, are given in Table 3. Also given in table are the timing results of the BZ algorithm and the sequential *ParK* algorithm. Note that we have included a column that shows the time taken for the scan phase of the *ParK* algorithm. All the timing results shown in table are in seconds. All the graphs are treated as undirected graphs and the timing results include the time taken for initialization.It can be clearly seen from the table that the *ParK* algorithm outperforms the state-of-the-art BZ algorithm for all the graphs (except *wiki-Talk*)) and the performance gap widens as the graph size increases. We observe that there are two reasons for *ParK* algorithm not performing better than BZ algorithm for the wiki-Talk graph. The first reason is that more than 70 percent of the vertices have degree 1. This is beneficial for the BZ algorithm as the adjacency lists of these vertices are accessed in sequential order. The second reason is that the time taken for scan phase is a significant(more than half) portion of the total time taken. However, since the scan phase is embarrassingly parallel, the time taken for scan phase can be made negligible with the use of parallelism and the *ParK* algorithm performs better as it is better scalable than the BZ algorithm. To verify the cache performance we used a tool called [42]. For *soc-LiveJournal1* graph, the BZ algorithm resulted in 227 million(148m read + 79m write) L1 data cache misses and 46m(31m read + 15m write) L3 cache misses while the sequential *ParK* resulted in 202m(192m read + 10m write) L1 cache misses and 32m(22m read + 10m write).

We measure the performance of the *ParK* algorithm in terms of millions of edges per second which is computed using $m_a/time$ where m_a is the number of edges accessed and *time* is the running time of the algorithm in seconds. Since the *ParK* algorithm processes all the vertices exactly once by accessing all of their neighbors, each edge in the graph is accessed exactly twice and so $m_a = 2m$. For the results to correctly reflect the performance, we exclude the initialization time from the running time. To avoid unexpected behaviour, for all our experiments we pin the threads to specific cores such that threads 0 to 7, 8 to 15, 16 to 23, 24 to 31 run on 1st, 2nd, 3rd and 4th socket respectively.

Figures 10a and 10b plot the speedup and processing rates respectively for three graphs: *rmat-32-512*, *rand-32-512* and *com-Friendster*. We can see that the approach scales well for both *rmat-32-512* and *rand-32-512* graphs. Also, the processing rate

increases gradually with the increase in number of threads. However, for the *com-Friendster* graph, the approach does not scale considerably and shows only limited growth in processing rate. We closely analyze the graphs to understand the behavior of the approach. We consider the following factors to analyze a graph: number of levels, number of sub-levels and percentage of vertices processed in each level. We have seen that the task of processing the vertices in sub-level is distributed among multiple threads and the threads are synchronized after each sub-level. If number of vertices processed in a level is large, the threads spend most of the time in processing the vertices and the synchronization overhead incurred after every sub-level is minimum. However, if there are only a few vertices in a level and the number of sub-levels is large, the threads spend most of their time at the synchronization barrier resulting in huge synchronization overhead.

Figures 2.5, 2.5 and 2.5 plot the number of sub-levels and percentage of vertices processed in each level for the graphs *rand-32-512*, *rmat-32-512* and *com-Friendster* respectively. Note that the missing percentage of vertices in the plots belong to level 0. In *rand-32-512* graph, most of the levels have low percentage of vertices. However, since these levels have only few sub-levels, the synchronization overhead is minimum and so the approach is able to scale well. In case of *rmat-32-512* graph, all the levels process approximately the same number of vertices (around 1 to 2 percent which is 320,000 to 640,000) which is large enough to keep the threads busy for longer time than the synchronization time. And, there are only few levels with large number of sub-levels. Therefore, the approach achieves good speed-up for the graph. In *com-Friendster* graph, however, the majority of the vertices are processed in the lower levels (level less than 50). Interestingly, the number of sub-levels is also low for these levels. All the remaining levels (beyond 50), process very few vertices and large number of sub-levels. This incurs in huge synchronization overhead justifying the speedup and processing rate shown in Figure 10.

2.6 SUMMARY

In this chapter, we have discussed the *access transformation* technique to improve locality of reference. The technique helps in reducing the random memory access nature of an algorithm. The technique is mainly applicable to an iterative approach with limited number of iterations. The main idea of the technique is to use scan-and-extract operation to extract the required data the directs the order in which

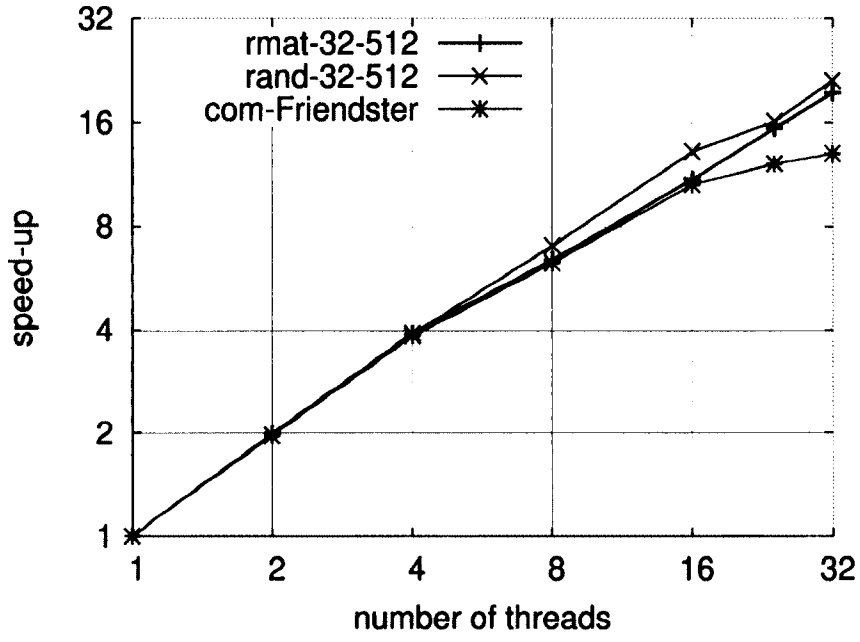
the vertices/edges are processed. We demonstrate the effectiveness of the technique using k-core decomposition problem.

We have defined the k-core decomposition problem and discussed the state-of-the-art algorithm for k-core decomposition. We have analysed the access pattern of the algorithm and pointed out the random nature of the algorithm and also showed that the algorithm is not amicable to parallelization due to synchronization issues. We then proposed a new new algorithm, called *ParK*, that adapts the *access transformation* technique. The algorithm eliminates the use of some data structures that result in random access and improves the opportunity for sequential access.

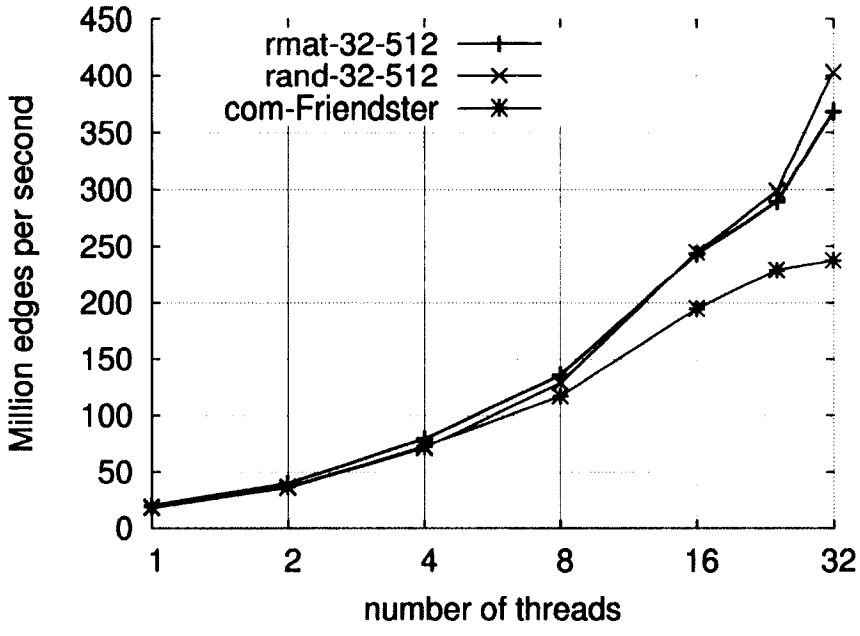
We then discussed the parallelization of the *ParK* algorithm. The scan phase of the algorithm is embarrassingly parallel while the loop phase requires synchronizing the threads after each sub-level. The parallelization methodology discussed in this chapter can be improved further by reducing the synchronization overhead caused due to large number of sub-levels. Since *ParK* algorithm is a level-synchronous approach similar to the BFS algorithm of Agarwal et al., the parallelization techniques used in [43] can be applied to *ParK*. For example, as the inter-socket atomic operations cannot scale efficiently across sockets, they are avoided using a channel mechanism. The vertices can be divided among the sockets and the threads process only the vertices that are assigned to the socket in which the thread is running. Any vertex that is to be processed and is assigned to other socket is placed in a socket queue of the corresponding socket. This confines the atomic operations to the sockets. There are also other techniques [44] proposed to improve the level-synchronous approaches which can be applied to the *ParK* algorithm to reduce the synchronization cost.

```
1: function scan-and-extract(deg, level)
2:     curr = ∅
3:     idx = 0
4:     for i = 0 to n − 1 do in parallel
5:         if deg[i] = level then
6:             a = atomicIncrement(idx, 1)
7:             curr[a] = i
8:         end if
9:     end for
10:    return curr
11: end function
12: function processSublevel(curr, deg, level)
13:    idx = 0
14:    next = ∅
15:    for each vertex v in curr do in parallel
16:        for each vertex u adjacent to v do
17:            if deg[u] > level then
18:                a = atomicDecrement(deg[u], 1)
19:                if a ≤ level then
20:                    atomicIncrement(deg[u], 1)
21:                end if
22:                if a − 1 = level then
23:                    b = atomicIncrement(idx, 1)
24:                    next[b] = u
25:                end if
26:            end if
27:        end for
28:    end for
29:    return next
30: end function
```

Figure 9: Parallel version of *ParK* algorithm



(a) speed-up



(b) processing rates

Figure 10: Scalability and Performance results for different graphs

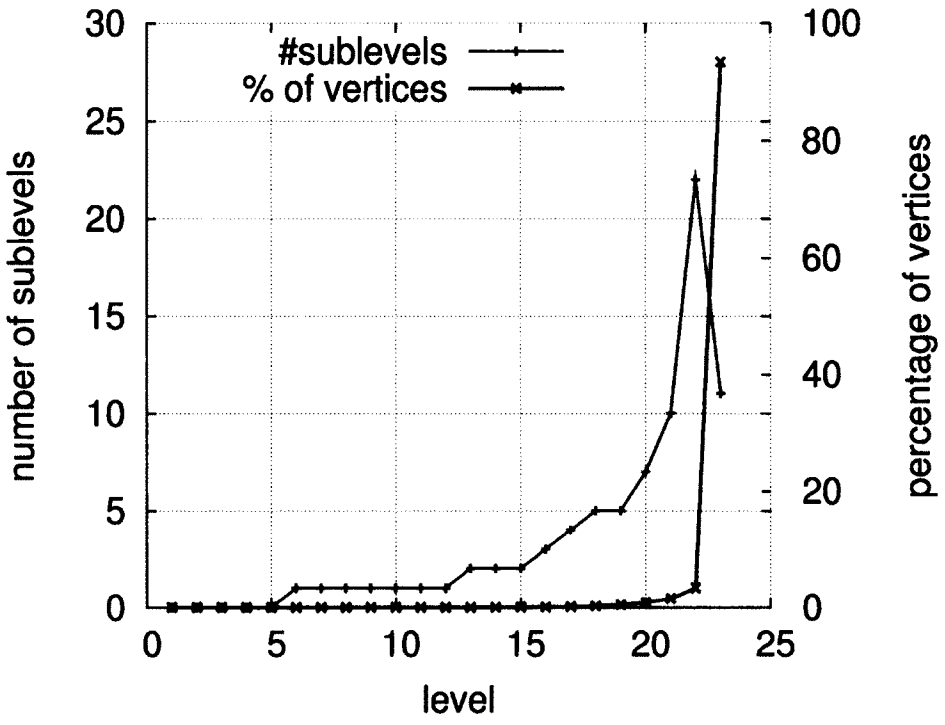


Figure 11: Plots showing the number of sub-levels in a level and the percentage of vertices processed in a level for *rand-32-512* graph

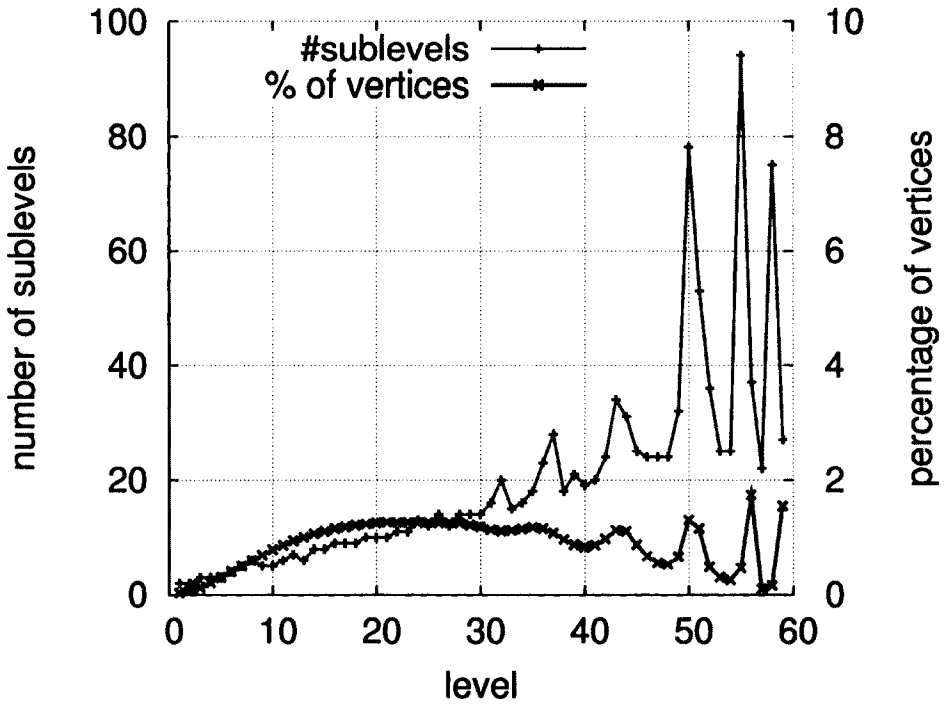


Figure 12: Plots showing the number of sub-levels in a level and the percentage of vertices processed in a level for *rmat-32-512* graph

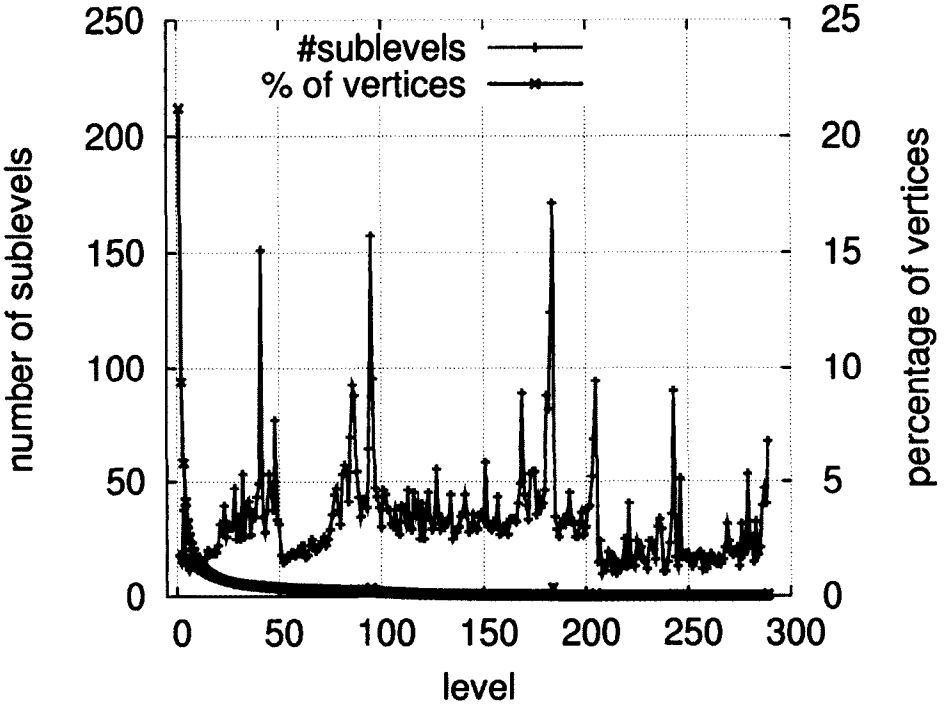


Figure 13: Plots showing the number of sub-levels in a level and the percentage of vertices processed in a level for *com-Friendster* graph

CHAPTER 3

TASK-SET REDUCTION

In the previous chapter, we have seen the impact of memory access pattern on memory locality. We have discussed a technique called *access transformation* which reduces the random nature of the memory access pattern to improve locality. In this chapter, we propose another technique, called *task-set reduction*, to improve locality. While the *access transformation* technique improves the spatial locality by focusing on the memory access pattern, the *task-set reduction* technique improves the temporal locality by focusing on the size of the data.

We define *task-set* as the collection of data that is repeatedly accessed to process a task. We argue that the size of *task-set* plays a critical role in memory locality and hence in the overall performance of an algorithm. We have seen that the memory access pattern in graph algorithms is generally very random in nature. Therefore, for most graph algorithms the *task-set* is repeatedly accessed and in random order. If the size of *task-set* is larger compared to the size of last level cache, then the number of cache misses can be large and can significantly degrade the performance. We ran a simple benchmark to verify the impact of *task-set* size on the memory performance. The results are reported in Figure 14. As it can be seen, as the size of the *task-set* increases, the memory latency increases. Therefore it is important to keep the size of *task-set* as minimal as possible.

Reducing the *task-set* size can be achieved in different ways. For example, for the k-core decomposition problem explained in the previous chapter, the *task-set* size of the BZ algorithm is $3n$ where n, is number of vertices(ignoring the size of *bin* array). For a graph with 1 million vertices, the *task-set* size is around 12MB (3million * 4 bytes for integer). If the cache size is not large enough to hold the *task-set*, it can severely hurt the performance. We have proposed a new algorithm for k-core decomposition. Though the new algorithm improves performance by reducing the random nature of the memory access pattern, it is to be noted that the algorithm also reduces the *task-set* size from $3n$ to n(as only *deg* array is used). Therefore, the scan-and-extract operation used in the algorithm also resulted in reduced *task-set* size.

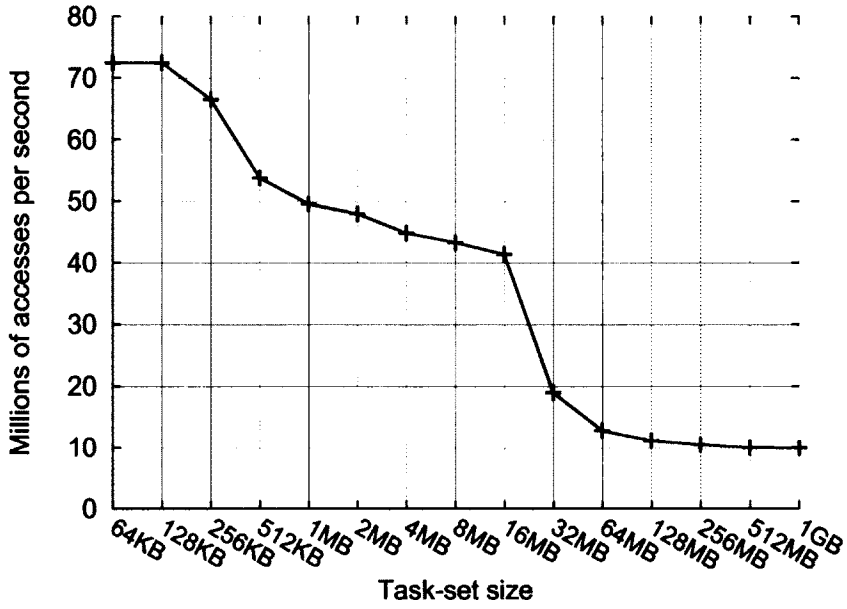


Figure 14: Plot showing the memory latency relative to the *task-set* size

Another method that can be used for reducing the size of *task-set* is compression. Compression refers to storing the data using minimal amount of memory. Graphs algorithms use different task related data, like the degree of vertices, distance of vertices from a source vertex. The data is stored by the program in some data structure like array. By carefully examining the nature of the data and possible values of the data, it is often possible to reduce the size of the data structure that stores the data. For example, many graph applications repeatedly access the degree of vertices which is generally stored in an integer array. An integer uses 4 bytes of memory and store any value from 0 to 2^32 (unsigned integers). However, for most graphs the maximum degree is very less and do not require 4 bytes to store the degree. Therefore, the degree array can be compressed, 2 to 4 times for most graphs, based on the number of bits required to store the maximum degree.

Another method that is commonly used for regular applications is called blocking [17]. In this technique, the memory accesses are organized in such a way that a small subset of data is loaded into the cache and is used/reused. For graph algorithms, the blocking technique can be applied by diving a task into smaller sub-tasks such that each sub-task works on a smaller *task-set*. We apply this method of *task-set*

reduction to the triangle listing problem explained in Chapter 4.

While in some cases reduction of *task-set* is easy to achieve, it generally requires careful examination of the algorithm and the data required to process a task. In this chapter, we present a new algorithm for maximal clique enumeration problem which uses the *task-set reduction* technique to improve the memory locality. We first define the problem and present some popular algorithms for maximal clique enumeration. We then analyze the state-of-the-art algorithm (Eppstein et al.'s algorithm) [32][45] in terms of the size of the *task-set*. We propose a new algorithm for maximal clique enumeration, called *pbitMCE*, that significantly reduces the *task-set* size by using bit representation of data. The experimental results comparing the new algorithm with the state-of-the-art algorithm are presented. We show that the new algorithm outperforms the other algorithms for most graphs. We also present results that show that the algorithm scales well, up to 29 times using 32 cores on multicore and up to 106 times using 128 processes on distributed memory architecture.

3.1 MAXIMAL CLIQUE ENUMERATION

Clique is a fundamental concept in graph theory. In a graph $G(V, E)$, a clique is a complete subgraph, i.e., a subgraph in which every pair of vertices is connected by an edge. A maximal clique is a clique that is not contained in any other clique. A maximum clique on the other hand is the largest clique in the graph. In Figure 15 , $\{1, 2, 3\}$, $\{1, 3, 4\}$ and $\{3, 4, 5, 6\}$ are maximal cliques and $\{3, 4, 5, 6\}$ is a maximum clique. $\{4, 5, 6\}$ is a clique but not maximal clique as it is contained in $\{3, 4, 5, 6\}$.

Clique finding plays a vital role in many applications. It plays a major role in analyzing social networks. Cliques in social networks represent a group of people who are closely tied together that share common interests. Community detection is a common task in social network analysis and clique finding plays a major role in community detection [46][47]. Clique finding is also used in other analysis application like social hierarchy detection using email communications [48], in the recovery of depth from stereoscopic image data [49], in data mining for discovery of association rules [50].

Another area that cliques are widely used is bioinformatics. It is common to represent the biological data like protein structure in the form of a graph. Finding cliques is a major part in detecting protein-protein interaction complex [51], motif discovery [52], detect structural motifs from protein similarities [53] and aligning 3D

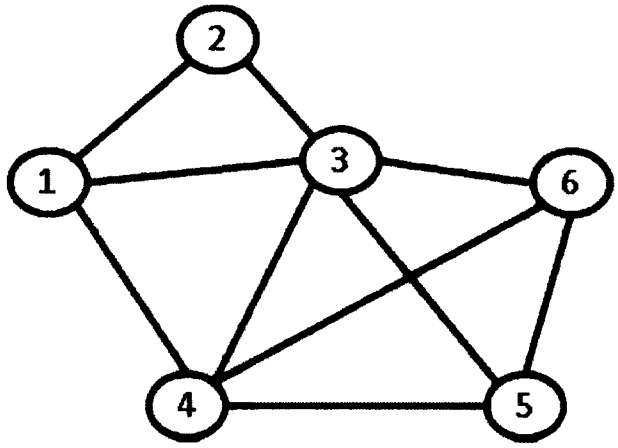


Figure 15: A simple graph

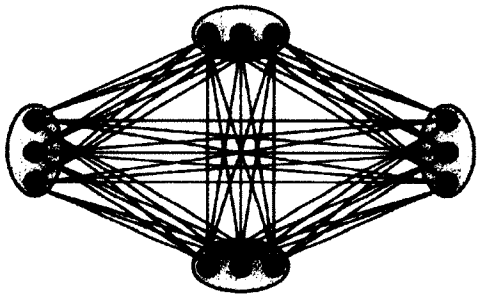


Figure 16: A Moon-Moser graph

structures [54] and many other applications [55][56][57].

The problem of finding all the maximal cliques in a graph is referred to as maximal clique enumeration(MCE). MCE is equivalent to finding all maximal independent sets in a complimentary graph. An algorithm which finds all the maximal cliques in a graph takes exponential time in worst-case as there can be exponential number of cliques in a graph. It has been shown that there can be $3^{n/3}$ maximal cliques in the worst case [58], where n is the number of vertices. Figure 16 shows such a graph. These kind of graphs are referred to as Moon-Moser graphs. However, in reality, these kind of graphs are highly unlikely to occur.

3.2 RELATED WORK

Clique finding is an extensively studied problem. There are a large number of approaches available in the literature. The first attempt to finding cliques was made in 1957 [59]. It was applied for analysis of sociometric data. In this section, a brief overview of some of the approaches for maximal clique enumeration is presented. First we briefly describe some of the sequential approaches. There are relatively very few parallel algorithms available in literature. We describe the parallel algorithms later in the section. Then we describe in detail three popular sequential algorithms for MCE: the BK algorithm [60], the Tomita et al.'s algorithm [61] and the Eppstein et al.'s algorithm [32].

3.2.1 SEQUENTIAL ALGORITHMS

The initial algorithms proposed for MCE were referred to as point removal methods. The cliques of a graph G are generated from the cliques in the graph $G \setminus \{v\}$ which is obtained by removing v from G. Approaches that are based on point removal method are given in [62][63][64]. These approaches initially generate a set of cliques, C, that may contain duplicate or non-maximal cliques. The non-maximal and duplicate cliques are then filtered out in some way to extract the final set of maximal cliques. For the purpose of filtering, the set C must be stored in memory and in general, the size of C is much greater in size that these approaches quickly fall into memory problems. These approaches therefore could not be applied to larger graphs.

The real breakthrough in MCE algorithms came in 1973 when two different approaches were proposed, one by Bron and Kerbosch [60] and the other by Akkoyunlu [65]. Unlike the previous approaches, these approaches do not generate non-maximal and duplicate cliques. They use different techniques to avoid generating those cliques. For example, the BK algorithm by Bron and Kerbosch uses a special data structure that stores the already explored paths so that the paths which might result in duplicate or non-maximal cliques are not revisited.

The algorithm by Akkoyunlu is a depth first search algorithm. It recursively partitions the graph such that each maximal clique can be generated by one of the partitions. Similar to Akkoyunlu's algorithm, BK algorithm is also a depth first search algorithm. The BK algorithm maintains three lists throughout the procedure:

the *compsub* list, the *not* list and the *cand* list. At each step, the *compsub* list consists of the vertices that form a clique and possibly a maximal clique. The *cand* list contains the vertices that are connected to all the vertices in *compsub* list and are candidates to be added to the current clique. The *not* list also contains the vertices that are connected to all the vertices in the *compsub* list but adding these vertices to the current clique results in duplicate cliques. The BK algorithm proceeds by selecting a vertex from the *cand* list and updating the three lists based on the currently selected vertex. If the *cand* and *not* lists are both empty, then the vertices in the *compsub* list constitute a maximal clique. The BK algorithm is described in much detail later in the Section 3.2.3.

There were two variations of the BK algorithm proposed in [60] : the basic version and the pivoting version. In the pivoting version, at each level, a vertex called pivot is selected from the candidate list that meets some criteria. The number of recursive calls depend on the pivot vertex selected. The details of the pivoting version of the BK algorithm are presented later in section. Though both Akkoyunlu's algorithm and BK algorithm perform equally well, the BK algorithm has been more widely used due to its simplicity. Many variations and extension to the BK algorithm have been proposed. These approaches vary based on their pivot selection rule. Johnston et al. [66] proposed a number of variations of the BK algorithm. It was shown that the original BK algorithm performed well against the other variations. Another set of variations was presented by Koch et al. [67] and Tomita et al. [61]. Tomita et al. also provided the theoretical time complexity for their approach and showed that their approach is worst case optimal. The time complexity is shown to be $O(3^{n/3})$. Later Karande et al. [68] investigated the three approaches: the BK algorithm, the Koch et al.'s variation and Tomita et al.'s variation. They tried to bridge the gap between the three approaches and showed that Tomita et al.'s approach is a variation of the BK algorithm based on an unexplored observation made by Koch et al. We present the details of the Tomita et al.'s algorithm later in the Section 3.2.4.

Another set of algorithms have been proposed and are referred to in the literature as reverse search algorithms. The first reverse search algorithm was proposed by Tsukiyama et al. [69]. The time complexity for this algorithm is $O(nm\mu)$ where n is the number of vertices, m is the number of edges and μ is the number of maximal cliques. The results of Tsukiyama et al. are further generalized by Lawler et al. [70] and an improvement for the algorithm is presented by Chiba et al. [71] with

computational complexity $O(a(G)m\mu)$ where $a(G)$ is the arboricity of the graph. Later, Makino and Uno [72] proposed new algorithms based on Tsukiyama et al's algorithm. One of their variations has a computational complexity $O(\Delta^4\mu)$ where Δ is the maximal degree of G. They presented a number of experimental results and showed that their algorithm is considerably faster than the Tsukiyama et al's algorithm. However, the experimental results presented in Tomita et al. that compare Tomita et al.'s approach with that of Chiba et al's and Makino and Uno's algorithm show that in practice, Tomita et al.'s algorithm performs well compared to the other approaches.

The main disadvantage of the Tomita et al's approach is that both the theoretical computational complexity and the experimental running time were based on adjacency matrix representation of the input graph. Adjacency matrix has an advantage of taking a constant time for checking the adjacency of two vertices. However, for large sparse graphs, which most real world graphs are, adjacency matrix is not practical due to memory limitations.

Recently, another notable contribution to the MCE problem was made by Eppstein et al. [32][45]. Eppstein et al.'s approach was based on the BK algorithm and Tomita et al.'s pivot selection rule. Eppstein et al. showed that the ordering of the vertices plays a vital role in the overall performance of an algorithm. They use an ordering based on the degeneracy of the graph. Degeneracy is the smallest number d such that every subgraph of the graph has atleast one vertex with degree less than or equal to d. The degeneracy ordering of vertices is the ordering in which every vertex has at most d neighbors that have order greater than itself. Eppstein et al.'s approach is based on Tomita et al.'s approach but uses adjacency list representation of the input graph, thus making it applicable to larger graphs. By using the degeneracy ordering and a special data structure to store only the current working set, Eppstein et al.'s approach achieves a computation complexity of $O(dn3^{d/3})$ where d is the degeneracy and n is the number of vertices of the input graph. Eppstein et al. also presented considerable amount of experimental results and showed that their approach is faster than the Tomita et al.'s algorithm for most graphs, sometimes faster by a large factor. Eppstein et al.'s approach is described in detail in Section 3.2.5.

3.2.2 PARALLEL ALGORITHMS

While there have been many sequential approaches present in the literature, there are only a handful of parallel approaches available for MCE. In addition to solving the problem of MCE, the parallel algorithms should also consider various factors like load balancing and scalability. The parallel algorithms should be able to effectively use the underlying hardware to efficiently enumerate the maximal cliques. In this section, we present different parallel algorithms and how they address the challenges involved in parallelizing.

The first parallel approach was proposed in [73]. It is referred to as *pCliques*. *pCliques* is based on an the approach of Kose et al. [74]. It generates the cliques in increasing order of clique size. The cliques of size $k+1$ are computed using the cliques of size k. So the cliques of size k must be present in memory and scanned multiple times to generate larger size cliques. This makes the approach memory intensive and impractical for large scale graphs. In [73] the authors also presented experimental results for *pCliques*. *pCliques* was only able to achieve a speedup of 91 using 256 processors.

Peamc [75] is another parallel algorithm proposed for MCE. It uses triangle structure as the basis. At each level it generates a set of vertices that form triangles with the current vertex and its neighbors. It proceeds recursively by selecting a vertex from the set of vertices generated from the previous level. By selecting the vertices in the increasing order *peamc* avoids the chance of generating duplicate cliques. However, it might generate non-maximal cliques. It uses an additional filtering step to check if the clique is maximal. *peamc* uses a simple strategy for parallelizing. The vertices are distributed to the available computing nodes and the nodes independently work on the vertices assigned. Most real world graphs follow power-law degree distribution. *peamc* does not work well for these graphs as the load is not balanced. Some computing nodes terminate earlier while other nodes are still processing. The experimental results presented in [75] show that *peamc* is able to achieve a speedup of 23 with 30 processes.

[76] presents a state-of-the- art parallel MCE algorithm. We refer to the algorithm as PSMCE. PSMCE parallelizes the original BK algorithm. It uses different strategies to efficiently parallelize the BK algorithm. PSMCE uses a special data structure called *candidate path structure* which is a basic unit of work that can be shared between processes. We have seen that the BK algorithm uses three lists: the *cand* list, the *not* list and the *compsub* list. The *candidate path structure* essentially

```
1: procedure BK(P, X, R)
2:     if P is empty then
3:         if X is empty then
4:             R is a maximal clique
5:         end if
6:     else
7:         for each vertex v in P do
8:             call BK(P ∩ N(v), X ∩ N(v), R ∪ {v})
9:             P = P \ v
10:            X = X ∪ v
11:        end for
12:    end if
13: end procedure
```

Figure 17: The BK algorithm

consists of *cand* list, *not* list, the current vertex and the level of the search tree node.

PSMCE uses a simple strategy for parallelization. Like *peamc*, it equally distributes all the vertices to the available processing units. However, the major contribution of PSMCE is its more refined level of load balancing. It achieves such fine level of load balancing by using the *candidate path data* structure and a stack structure. At each level, a candidate path(cp) structure is removed from the stack structure and using the cp it generates the *candidate path structures* for the next level and pushes into the stack structure. If a processing unit completes the work assigned to it, instead of terminating, it requests work from a randomly selected process or task. The process that receives request can share its work by removing some of the candidate path structures from its stack structure and sending them to the requesting process. In [76] PSMCE, the authors presented the experimental results of PSMCE and showed that PSMCE scales linearly up to 2048 processes.

dMaximalCliques [77] is a distributed algorithm for maximal clique enumeration. It is based on Tsukiyama's algorithm. Similar to *peamc*, this algorithm does not include any strategies for dynamic load balancing and hence was not able to scale well.

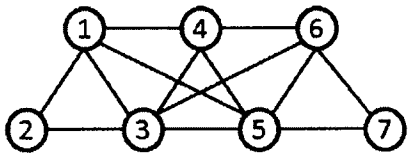
3.2.3 THE BK ALGORITHM

The BK algorithm(short for Bron–Kerbosch algorithm) is the most commonly

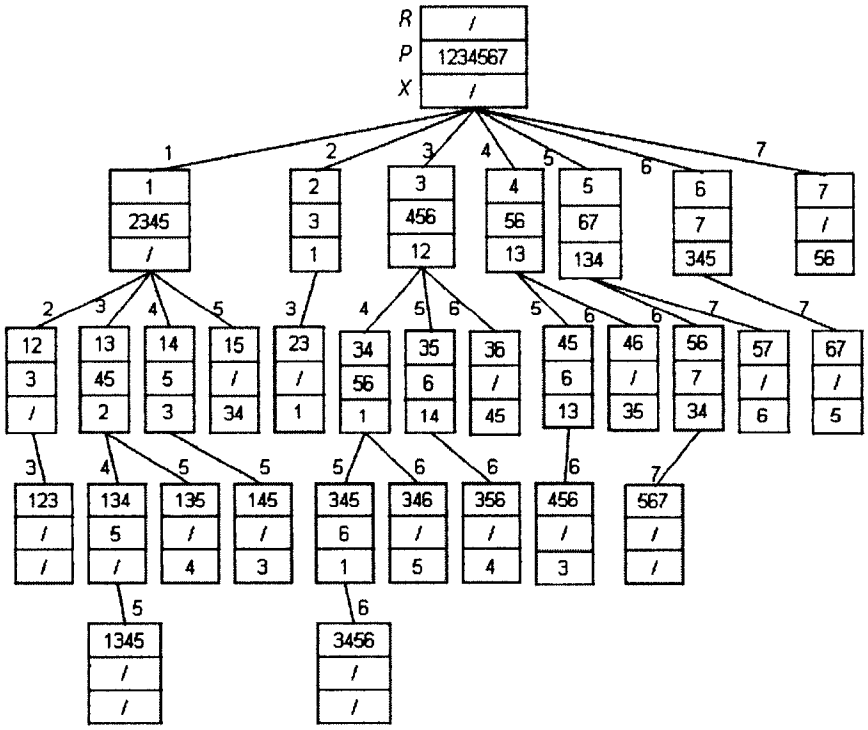
used algorithm for clique finding. It is a recursive backtracking algorithm. It uses three lists throughout the enumeration process: *cand* list, *not* list and *compsub* list, commonly denoted as P, X and R respectively (their sizes are denoted by p, x and r respectively). The set of vertices in R form a clique, but it might not be maximal. The idea is to extend R until it forms a maximal clique. P contains the set of vertices that are adjacent to all the vertices in R and are potential candidates to be added to the maximal clique. X list also contains the vertices that are adjacent to all the vertices in R list but adding these vertices to the clique results in duplicate or non-maximal cliques. The BK algorithm is presented in Figure 17. $N(v)$ in the algorithm represents the adjacency list of a vertex v (note that $v \notin N(v)$). In each call to BK procedure, it iterates p times once for each vertex in P. In each iteration, a vertex from P is added to R, new sets of vertices of P and X are computed and input to a subsequent recursive call. The BK algorithm can be viewed as exploring a search tree. Figure 18b shows the search tree corresponding to the graph in Figure 18a. Each node in the tree represents a call to BK procedure. The algorithm proceeds by exploring the search tree in a depth first style, backtracking when P becomes empty. Each node in the search tree has three rows showing the *compsub* list(R), *cand* list(P) and *not* list(X) that are input to the recursive call. The leaf nodes with empty *cand* and *not* list correspond to maximal cliques.

3.2.4 THE TOMITA ET AL.'S ALGORITHM

The algorithm described in previous section is the basic version of the BK algorithm. Another variation of the BK algorithm exists that involves a technique called pivoting. We have seen in the previous section that the BK algorithm makes P recursive calls. Pivoting improves the basic BK algorithm by reducing the number of recursive calls. This is done by selecting a vertex u called pivot , and all the subsequent maximal cliques must contain a non-neighbor of u. This reduces the number of recursive calls by $|P \cap N(u)|$. A number of variations for selecting the pivot vertex were proposed [61][67]. The variation by Tomita et al. [61] is shown to be best in theory and practice. Tomita et al.'s pivot selection method is to select a vertex u from $P \cup X$ that maximizes $|P \cap N(u)|$. Obviously, this results in minimum number of recursive calls among all possible pivots. We denote Tomita et al.'s algorithm by TTT in the rest of the chapter. The TTT algorithm is shown in Figure 19 and the TTT search tree for the example graph in Figure 18a is shown in 20. By comparing



(a)



(b)

Figure 18: An example graph and its BK search tree

```
1: procedure TTT(P, X, R)
2:     if P is empty then
3:         if X is empty then
4:             R is a maximal clique
5:         end if
6:     else
7:         choose a pivot vertex u in P ∪ X that maximizes |P ∩ N(u)|
8:         for each vertex v in P \ N(u) do
9:             call TTT(P ∩ N(v), X ∩ N(v), R ∪ N(v))
10:            P = P \ v
11:            X = X ∪ v
12:        end for
13:    end if
14: end procedure
```

Figure 19: The Tomita et al.'s algorithm

Figures 18b and 20 it can be clearly seen that the TTT algorithm results in reduced number of child nodes compared to the BK algorithm, i.e. TTT algorithm reduces the number of recursive calls.

The analysis and the implementation of the TTT algorithm presented in [61] is based on adjacency matrix representation of the input graph. The two major components of the algorithm are pivot selection and set intersections(lines 7 and 9 respectively in Figure 19). To select the pivot vertex, we need to perform $P \cap N(u)$ for each $u \in P \cup X$. By using adjacency matrix representation, the pivot selection can be done in $O(p(p+x))$ where $p = |P|$ and $x = |X|$. Similarly, the set intersections in line 9 can be performed in $O(p + x)$. The worst case time complexity for TTT is shown to be $O(3^{n/3})$. However, most real world graphs are large sparse graphs and the adjacency matrix representation is impractical for such graphs.

3.2.5 THE EPPSTEIN ET AL.'S ALGORITHM

To overcome the drawback of the TTT algorithm, Eppstein et al. proposed a new algorithm. In the rest of the chapter, we refer to the new algorithm as ELS algorithm. The ELS algorithm is based on TTT algorithm but uses adjacency list representation of the graph. The ELS algorithm is given in Figure 21. Unlike the BK and TTT algorithms, ELS algorithm explores multiple search trees, one search tree corresponding to each vertex in the graph. This makes it a more suitable choice

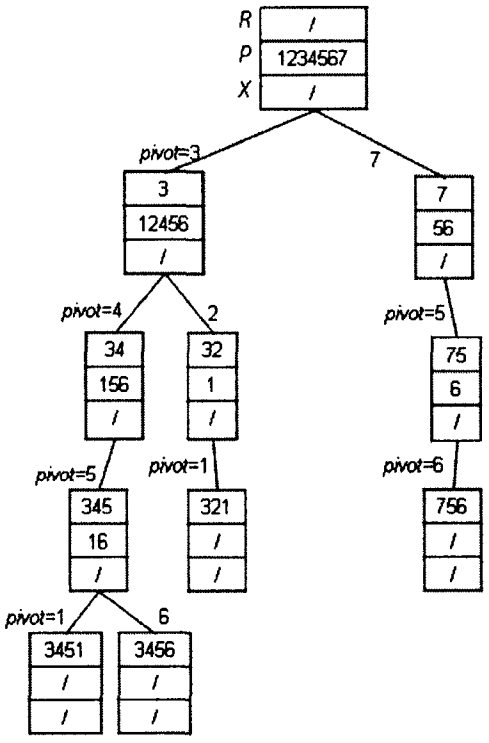


Figure 20: TTT search tree for the graph in Figure 18a

for being used in a parallel approach. Once the root node for the search tree is obtained, the rest of the search tree exploration is similar to the TTT algorithm. ELS uses an initial ordering of vertices based on degeneracy. Degeneracy of a graph G is the smallest number d such that every subgraph of G has atleast one vertex with degree at most d. Degeneracy ordering is the ordering of vertices such that every vertex has at most d neighbors that come later in the ordering. Degeneracy ordering of vertices in a graph can be computed in linear time by using the k-core decomposition algorithm explained in Chapter 2. The order in which the vertices are deleted from the graph in the k-core decomposition algorithm is the degeneracy ordering. The degeneracy ordering for the graph in Figure 18a is $\{2, 7, 1, 3, 4, 5, 6\}$.

For each vertex v in the graph, an initial list of vertices in P, X and R are computed based on the degeneracy ordering of vertices and input to the TTT procedure. The set P is computed by adding all the neighbors of v that come later in the ordering. Similarly, the set X is computed by adding all the neighbors of v that come

1: **procedure** $ELS(V, E)$
2: **for** each vertex v_i in the degeneracy ordering $v_0, v_1, ..., v_{n-1}$ **do**
3: $P = N(v_i) \cap \{v_{i+1}, ..., v_{n-1}\}$
4: $X = N(v_i) \cap \{v_0, ..., v_{i-1}\}$
5: call TTT(P, X, $\{v_i\}$)
6: **end for**
7: **end procedure**

Figure 21: The Eppstein et al.'s algorithm

before v in the ordering. Degeneracy ordering makes sure that there are no more than d neighbors that come later in the ordering. Therefore, the size of P is limited to d in the outermost call to TTT (line 5 in Figure 21). This reduces the number of recursive calls within the outermost TTT call. The ELS search trees for the graph in Figure 18a can be seen in Figure 22. The figure shows 7 search trees, each tree corresponding to a vertex.

The ELS algorithm uses adjacency list representation for the input graph. After computing the initial R, P and X lists, the ELS algorithm makes a call to TTT procedure from which point the enumeration is done using TTT algorithm. We have seen that TTT procedure consists of two major components: pivot selection and set intersections. Since TTT algorithm uses adjacency matrix representation, the two components take $O(p(p + x))$ time. This time complexity is no more valid when adjacency list representation is used. In case of adjacency matrix representation the operation $P \cap N(u)$ for some vertex u can be done in $O(p)$ time since it only takes constant time to check if a vertex belongs to $N(u)$. In the case of adjacency list representation the same operation takes $O(p \cdot |N(u)|)$ time, if both P and $N(u)$ are unsorted, takes $O(p \cdot log(|N(u)|))$ time, if $N(u)$ is sorted, takes $O(p+|N(u)|)$ time, if both P and $N(u)$ are sorted. For pivot selection, $p + x$ such intersection operations need to be performed.

To make pivot selection fast, the ELS approach employs a subgraph representation, $H_{P,X}$(we refer to it as *hypergraph* in this chapter. Note that, this *hypergraph* is not the same as standard *hypergraph* defined in the literature). The *hypergraph* contains all the vertices in $P \cup X$ at the current level, and edges connecting a vertex in P to a vertex in $P \cup X$. Using the *hypergraph* representation, ELS computes the pivot vertex in $O(|P|(|P| + |X|)$ and the set intersections in $O(|P|^2(|P| + |X|)$.

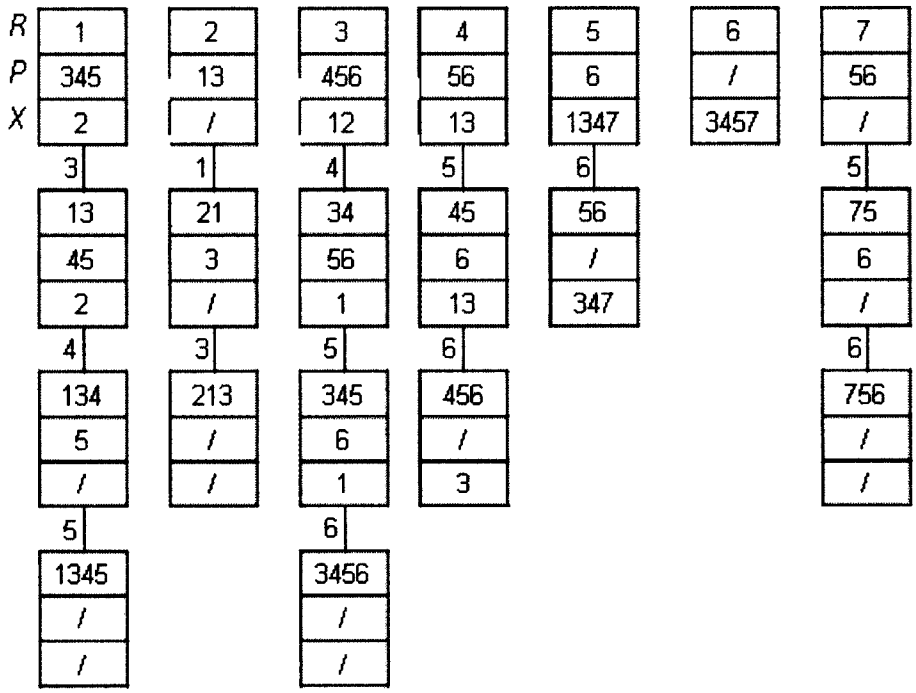


Figure 22: ELS search trees for the graph in Figure 18a. The degeneracy ordering for the graph is $\{2, 7, 1, 3, 4, 5, 6\}$. Each search tree in the figure corresponds to a vertex in the graph.

The running time of ELS is shown to be $O(dn3^{d/3})$, where d is the degeneracy of the graph, which is worst case optimal for large sparse graphs. More details of the *hypergraph* are given in the next section.

A similar approach that explores multiple search trees is used in PSMCE [76] with a difference in the ordering and the underlying algorithm used. PSMCE uses the initial ordering of the vertices as is, that is the ordering based on the vertex numbers while ELS uses degeneracy ordering. While ELS uses the Tomita et al.'s pivoting rule, i.e selects a pivot vertex u from $P \cup X$ that maximizes $P \cap N(u)$, PSMCE selects a pivot vertex u from P that maximizes $P \cap N(u)$.

Task-set size of ELS algorithm

We have seen that the ELS algorithm has separate tasks defined for each vertex. Compared to most other graph algorithms, which generally have very large *task-sets*,

the ELS algorithm has relatively smaller *task-set*. The initial *task-set* of the ELS algorithm consists of the *hypergraph*. As the enumeration proceeds, intermediate data is added to the *task-set*. Notice that, for each recursive call a new *cand* and *not* list is to be generated. The maximum size of a *hypergraph* is Md where M is the maximum degree and d is the degeneracy. However, in reality, it is much smaller than that. The size of intermediate data depends on the granularity of the task. If the search tree corresponding to a vertex is large with high depth and large number of branches then the intermediate data can be significant. In general, from our experiments we have noticed that in most cases the *task-set* is not small enough to fit in the faster caches(L1 and L2). The *task-set* may fit in the slower cache(L3) but since L3 is a shared cache, large *task-sets* can result in cache contention and can have significant impact on performance.

3.3 *PBITMCE* APPROACH

We propose an algorithm, called *pbitMCE*, which reduces the size of *task-set* by using bit representation. *pbitMCE* employs a novel data structure called partial bit adjacency matrix(*pbam*) which stores the initial *hypergraph* in a compressed form. Also, most of the intermediate data is stored in bit format, reducing the overall size of the *task-set*. *pbitMCE* approach not only reduces the memory requirement but also facilitates the use of bit-parallelism. Due to the intrinsic parallelism of the bit operations, the number of operations can be greatly reduced, by a factor of up to ω, the computer word size.

3.3.1 DEGENERACY ORDERING

Like in ELS algorithm, before the enumeration process, the vertices are reordered by degeneracy. As we have seen, degeneracy of a graph G is the smallest number d such that every subgraph of G has atleast one vertex with degree at most d. Degeneracy ordering is the ordering of vertices such that every vertex has at most d neighbors that come later in the ordering. Degeneracy ordering of vertices in a graph can be computed using the k-core decomposition algorithm given in Chapter 2. In *pbitMCE*, the advantage of using degeneracy ordering is two fold. The first is that, like in ELS, it reduces the number of recursive calls improving the overall performance. The second advantage comes in the context of parallelization. Although not explicitly, degeneracy ordering significantly contributes to load balancing. The role

of degeneracy ordering on load balancing is explained in Section 3.5.1.

3.3.2 PRE-PROCESSING

As we have seen in the previous section, the degeneracy ordering places the vertices such that each vertex has no more than d neighbors that come later in the ordering. In the rest of the section, we refer to the neighbors of a vertex v that come before v in the degeneracy ordering as *pre*-neighbors of v and the neighbors of v that come after v as *post*-neighbors of v. During enumeration, most of the time only the *post*-neighbors of vertices are used. The *pre*-neighbors of a vertex v are accessed only once during the enumeration process, i.e. to compute the root node of search tree of v. So to avoid unnecessary processing of the *pre*-neighbors, we partition the adjacency list into two separate lists: *pre*-adjacency list and *post*-adjacency list. We denote these lists as *preN* and *postN* respectively. By the property of degeneracy, the maximum size of *post*-adjacency list of any vertex in the graph is d, the degeneracy of the graph. The initial adjacency list can be discarded at this point as it is no longer required.

3.3.3 PARTIAL BIT ADJACENCY MATRIX

As in ELS, *pbitMCE* also uses Tomita et al.'s TTT algorithm as the basis. We have seen that the TTT uses adjacency matrix representation of the graph. The best asset of adjacency matrix is its constant lookup time. However, for large sparse graphs, which most real world graphs are, the adjacency matrix representation is not feasible because of the memory limitations. We propose a method by which we can take advantage of the constant lookup time of adjacency matrix representation and yet meeting the memory constraints.

Notice that to process the TTT call in line 5 of procedure *ELS*(Figure 21), the only data that is required is the list of vertices in P and X and their adjacency lists i.e a subgraph, denoted by $S(V_s, E_s)$, with $V_s = P \cup X$ and $E_s = \{(u,v)|u \in P,\ v \in \{P \cup X\}\}$. Note that we did not include the edges connecting a vertex in X with another vertex in X since such edges are never used in the TTT algorithm. This subgraph is represented using a partial bit adjacency matrix (*pbam*). *pbam* is essentially a set of bit vectors of size P, each corresponding to a vertex in the subgraph. Each bit in a bit vector corresponds to a vertex in P. If a vertex u in $P \cup X$ is connected to a vertex v in P, then the bit corresponding to v is set to 1 in the

bit vector corresponding to u. Before each initial call to TTT in line 5 in procedure *ELS*, we construct a *pbam* and pass it to the TTT procedure. The construction of *pbam* is described in the following section.

Construction of Partial Bit Adjacency Matrix

To construct *pbam* we use a technique called renumbering. Renumbering maps the vertices in $P \cup X$ which are originally numbered in the range $[0 \ldots |V| - 1]$ to a new number in the range $[0 \ldots p + x - 1]$ where $p = |P|$ and $x = |X|$. Each vertex in P is assigned a unique number in the range $[0 \ldots p - 1]$ and each vertex in the set X is assigned a unique number in the range $[p \ldots p + x - 1]$. Let $\eta(v)$ denote the new number assigned to a vertex v. The key value pairs $(v, \eta(v))$ are stored in a hash table. We denote a bit vector corresponding to a vertex v by $B_{\eta(v)}$. If a vertex u in $P \cup X$ is connected to a vertex v in P then a bit is set at index $\eta(v)$ in bit vector $B_{\eta(u)}$. The bits are set in *pbam* by iterating through the *post*-adjacency lists of the vertices in $P \cup X$. The structure is called partial bit adjacency matrix because like adjacency matrix it only takes a constant lookup time but it only has a portion of the adjacency matrix.

The procedure for constructing *pbam*(*Constructpbam*) is given in Figure 23. The function takes P and X as input and outputs *pbam* which is a set of bit vectors. The operations in lines 25 and 27 requires querying the hash table. If a vertex is not present in the hash table then the query returns -1. Assuming constant time to retrieve a value from hash table, *pbam* can be constructed in $O((p + x)d)$ time where d is the degeneracy of the graph. Note that d is the maximum possible size of *post*-adjacency list of any vertex in the graph. The maximum size of *pbam* is Md/ω where M is the maximum degree of the graph and ω is the computer word size.

Figure 24a shows an example graph with vertices arranged in degeneracy order. The initial subgraph of 2 is highlighted in the figure. The *post*-neighbors and *pre*-neighbors of 2 are shown in yellow and green respectively and the edges are shown in red. Figure 24c shows the vertices and the new numbers assigned to the vertices i.e. $(v, \eta(v))$ for all vertices in the subgraph. Figure 24d shows the *pbam* representation of the subgraph. Also shown in graph is the hypergraph representation used in ELS algorithm(Figure 24b). To give an idea of the memory requirement, consider the *cit-Patents* graph from the Stanford large network collection(details in Section 2.5). It has 3.7 million vertices and 16.5 million vertices. It has degeneracy 64, so the

maximum size of a bit vector is 8 bytes. The maximum degree for the graph is 793. Therefore, the maximum size required for *pbam* is $793 * 8 = 6344$ bytes which is small enough to fit in L1 cache(which is 8KB for the machine used in out experiments).

3.3.4 ENUMERATION

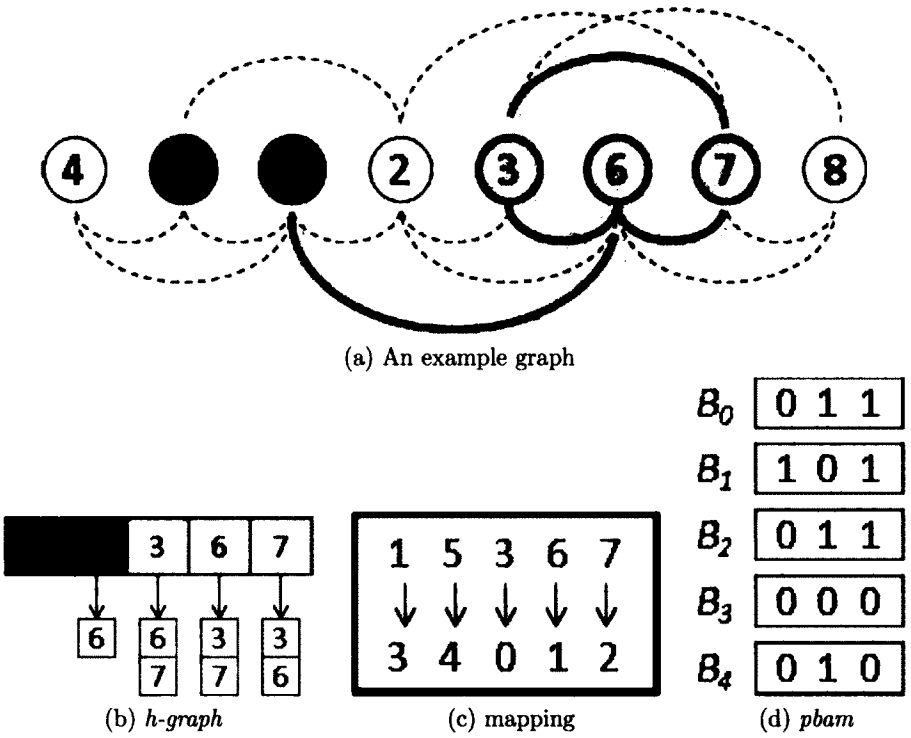
Figure 23 shows the *pbitMCE* algorithm. For each vertex v_i in the graph, P_{v_i} and X_{v_i} in lines 3 and 9 represent the *post*-adjacency list and *pre*-adjacency list of v_i respectively. During the construction of *pbam* all the vertices in P_{v_i} and X_{v_i} are assigned new numbers and these new numbers are used in the rest of the enumeration process instead of the original vertex numbers. Once the *pbam* is constructed all the processing is done using the bit vectors. The *cand* list P in each iteration is represented using a bit vector(by setting appropriate bits to 1). Notice the input parameters passed to the TTT procedure in line 9. In addition to the P, X and R lists, we also pass *pbam* and bit vector representation of P. The function *getBitVector* returns the bit vector representation of the input. In the following sections, we explain how *pbam* can be used for efficiently performing the two major components in TTT algorithm: pivot selection and set intersections.

Pivot selection using *pbam*

Pivot selection is a crucial part of the algorithm and takes a significant portion of the total enumeration time. We have seen that in TTT algorithm a vertex $u \in P \cup X$ that maximizes $|P \cap N(u)|$ is selected as pivot vertex. This operation requires a total of $|P| + |X|$ intersection operations. Therefore efficiently performing an intersection operation is crucial to the overall performance of the approach. In *pbitMCE* approach, we pass the bit vector representation of P, B_P, as input parameter to the *TTT* procedure along with *pbam*. The intersection operation $P \cap N(u)$ can be easily performed by doing logical AND of bit vectors B_P and B_u. $|P \cap N(u)|$ can be obtained by counting the number of set bits in the resultant bit vector. Since we need to perform $|P| + |X|$ such intersection operations, the pivot selection can be performed in $O((|P| + |X|)p_v)$ time where p_v is the size of a bit vector(which is equal to the size of *post*-adjacency list of vertex v, the vertex whose search tree is being explored.

```
1: procedure pbitMCE(V, E)
2:     for each vertex v_i in the degeneracy ordering v_0, v_1, ..., v_{n-1} do
3:         P_{v_i} = N(v_i) ∩ {v_{i+1}, ..., v_{n-1}}
4:         X_{v_i} = N(v_i) ∩ {v_0, ..., v_{i-1}}
5:         pbam_{v_i} = Constructpbam(P_{v_i}, X_{v_i})
6:         η(P_{v_i}) = {0, 1, ..., |P_{v_i}| − 1}
7:         η(N_{v_i}) = {|P_{v_i}|, ..., |P_{v_i}| + |X_{v_i}| − 1|}
8:         B_P = getBitVector(η(P_{v_i}))
9:         call TTT(η(P_{v_i}), η(X_{v_i}), {v_i}, pbam_{v_i}, B_P)
10:    end for
11: end procedure
12: function Constructpbam(P, X)
13:    counter=0
14:    for each vertex v in P do
15:        η(v) = counter
16:        increment counter by 1
17:        add (v, η(v)) to hash table
18:    end for
19:    for each vertex v in X do
20:        η(v) = counter
21:        increment counter by 1
22:        add (v, η(v)) to hash table
23:    end for
24:    for each vertex v in P ∪ X do
25:        x = η(v)                          ▷ obtained using hash table
26:        for each vertex u in postN(v) do
27:            y = η(u)                      ▷ obtained using hash table
28:            if y >= 0 and y < |P| then
29:                set bit at index y in B_x
30:                if x >= 0 and x < |P| then
31:                    set bit at index x in B_y
32:                end if
33:            end if
34:        end for
35:    end for
36:    return B = {B_0, B_1, ··· , B_{|P∪X|−1}}
37: end function
```

Figure 23: The *pbitMCE* algorithm



(a) An example graph

(b) *h-graph* (c) mapping (d) *pbam*

Figure 24: An example $h - graph$ and *pbam*

Set intersection using *pbam*

The set intersections in line 9 of TTT procedure in Figure 19 are also major components of the TTT algorithm. The operation $P \cup N(v)$ is straight forward to perform using *pbam*. Each vertex $u \in P$ can be checked if it is present in $N(v)$ in constant time. If uth bit in bit vector of v i.e. B_v is set to 1 then it is present in $N(v)$. However, the operation $X \cup N(v)$ is a little tricky. We cannot check if a vertex $u \in X$ is present in $N(v)$ using bit vector of v since B_v only stores the *post*-neighbors of v but u is a *pre*-neighbor of u(if it is a neighbor). However, we can check if u is adjacency to v using the the bit vector of u. If u and v are neighbors, then vth bit will be set in B_u. The set intersections can therefore be performed in $(|P| + |X|)$ time. Let $newP = (P \cap N(v))$. Notice that we also need to input the bit vector representation of $newP$ to the TTT procedure(see line 9 in procedure *pbitMCE* in Figure 23). This can be obtained by performing logical AND on the bit vectors B_P and B_v.

Table 2: Comparing *pbam* and *hypergraph*

Operation	*hypergraph*	*pbam*
construction	$O(d(\|P\| + \|X\|)$	$O(d(\|P\| + \|X\|)$
updating	$O(\|P\|^2(\|P\| + \|X\|))$	-
pivot selection	$O(\|P\|(\|P\| + \|X\|))$	$O(d(\|P\| + \|X\|))$
set intersection	$O(\|P\|(\|P\| + \|X\|))$	$O(\|P\| + \|X\|)$

3.3.5 HYPERGRAPH VS *PBAM*

We have seen in Section 3.2.5 that ELS algorithm uses a *hypergraph* structure to improve the performance of the approach. When processing the search tree of a vertex v, the *hypergraph* structure initially stores for each neighbor u of v an array consisting of neighbors of u in $postN(v)$. Like, *pbam*, construction of hypergraph takes $O(d(|P| + |X|))$ time where d is degeneracy. However, *pbam* doesn't need to be modified once it is constructed while *hypergraph* needs to be updated for every iteration in each recursive call. Figure 24 shows an example graph and the *hypergraph* and *pbam* representations. In each recursive call to TTT procedure, updating the *hypergraph* incurs an additional cost of $O(|P|^2(|P| + |X|))$ time. Computing the pivot vertex using *hypergraph* takes $O(|P|(|P| + |X|)$ time (recall that *pbam* takes $O(P_v(|P| + |X|))$ time). Set intersections using *hypergraph* take $O(|P|(|P| + |X|))$ time(while with *pbam* it takes $O(|P| + |X|)$. Table 2 summarizes the computational complexities for each of the steps using *pbam* and *hypergraph*.

3.3.6 COMPUTATIONAL COMPLEXITY

The time complexity of *pbitMCE* approach can be computed similar to the ELS approach in [32]. Let $D(p, x)$ be the running time of TTT procedure using *pbam*. By the description of TTT procedure and the complexities of individual operations

described in the previous section, D satisfies the following recurrence relation:

$$D(p,x) \leq \begin{cases} max_k\{kD(p-k,x)\} + d(p+x) & \text{if } p > 0 \\ c & \text{if } p = 0 \end{cases}$$

where c is a constant greater than 0. Eventhough in [32], the second term i.e. $d(p+x$ is replaced by $c_1p^2(p+x)$ in the above relation, solving both the recurrence relations result in same complexity $O((d+x)3^{p/3})$. The time complexity for the *pbitMCE* approach is therefore:

$$\sum_v O((d+|X_v|)3^{|P_v|/3}) = O((dn+m)3^{d/3}) = O(dn3^{d/3})$$

3.3.7 OPTIMIZATION

We have seen that pivot selection process involves performing logical AND operation between two bit vectors. The maximum size of a bit vector is d bits. To intersect two bit vectors d/ω logical AND operations are required where ω is the computer word size. This is efficient for higher levels of the search tree where most of the bits are set in bit vector of P. However as we reach the lower levels, the size of P reduces and the bit vector of P is mostly sparse. Even then, the intersection of two bit vectors require d/ω logical AND operation. This is highly inefficient. To overcome this drawback we introduce hierarchical renumbering.

Hierarchical renumbering

After the candidate list size reduces to a certain value τ, we construct a new partial bit adjacency matrix by using the renumbering logic explained in Section 3.3.3. The new *pbam* constructed is much smaller in size compared to the previous one. This new smaller matrix is used in further processing instead of the old larger *pbam*. This addresses the problem of requiring d/ω logical operations. It now only requires τ/ω operations instead. This process can be repeated when the candidate list size is further reduced. However, if this process is repeated more often, there is a possibility that the time taken for renumbering and constructing the partial adjacency matrices dominates the overall time taken for enumeration. Based on empirical observations, for all our experiments we have used two levels of renumbering one at $\tau = 128$ and the other at $\tau = 32$ i.e. whenever the candidate list size reduces to 128 or 32 we do renumbering and construct new partial adjacency matrices.

3.4 SEQUENTIAL PERFORMANCE RESULTS

All the results we present in this section are obtained on a four socket 2.27GHz Xeon X7560(Nehalem-EX) with 256 GB shared memory and running 64-bit Ubuntu 12.04. Each socket consists of 8 cores. Each core has a private 32 KB L1 cache and 256 KB L2 cache. A 24 MB L3 cache is shared by all the cores in a socket. All the implementation is done using C programming language and compiled using gcc compiler with -O3 optimization flag. The parallel implementation is done using OpenMP. We have used compressed sparse row(CSR) format to store the graph.

We experimented with several networks obtained from different collections. Table 3 describes the properties of the graphs used in our experiments. All the graphs are treated as undirected graphs. $|C|$, d, M in the table represent the number of cliques in the graph, degeneracy and maximum degree of the graph respectively. The description of the datasets used for experiments is given below:

Dataset 1: Stanford large network collection [40]

- *roadNet-CA*, *roadNet-TX*: Road networks. Intersections and endpoints are represented by vertices and roads connecting them are represented by undirected edges.
- *soc-Pokec*, *wiki-Talk*, *cit-Patents*, *web-BerkStan*, *web-Google*, *amazon0601* and *as-Skitter* are described in Section 2.5.

Dataset 2: Florida sparse matrix collection [78]. All the four graphs are from the 10th Dimacs challenge group [79]

- *coPapersDBLP*, *coPapersCiteseer*: Citation networks
- *channel-b050*: Graph from numerical simulations
- *europe-osm*: Street network

Dataset 3: Synthetic graphs generated using GTGraph, a graph generator [41].

- *rmat-10m-100m*: Generated using R-MAT model with default parameter values in GTgraph i.e. $(a, b, c, d) = (0.45, 0.15, 0.15, 0.25)$
- *er-1m*: Generated using Erdos–Renyi graph model with probability 10^{-3}.

Dataset 4: Graphs used in [45] to evaluate Eppstein et al.'s algorithm. We have selected the graphs that either take considerable amount of time or for which Eppstein et al.'s approach is slower than other approaches.

- *biogrid-yeast*: A protein-protein interaction network
- *random-100-0.9*, *random-300-0.6*, *random-500-0.5*, *random-1000-0.3*: Random graph with edge densities 0.9, 0.6, 0.5, 0.3 respectively.
- *p_hat300-2*: A DIMACS challenge graph [80] that has been algorithmically generated and is intended as difficult example for clique finding algorithms.
- *m-m-51*: A Moon-Moser graph [58].

We compare our approach with Eppstein et al.'s approach which is considered the state-of-the-art approach for maximal clique enumeration. We also compare with the Tomita et al.'s approach. We have seen that the Tomita et al.'s approach uses adjacency matrix representation. However, for some of the graphs used in our experiments, the adjacency matrix representation is not feasible. Therefore, for large graphs(datasets 1,2,3) we have used an adjacency list implementation of Tomita et al.'s algorithm and for smaller graphs(dataset 4) we use the original Tomita et al's approach, i.e. with adjacency matrix representation. The code for both the approaches is obtained from [81]. A comparison of time taken by different approaches is presented in Table 3. All these results are obtained by running the experiments on the multicore machine using a single thread. *pbitMCE-b* and *pbitMCE* in the table refer to the versions of *pbitMCE* without and with optimization discussed in Section 3.3.7 respectively. Unless otherwise stated, *pbitMCE* refers to the optimized version. All the results shown refer to the time taken in seconds.

It can be clearly seen from the Table 3 that *pbitMCE* is faster than the ELS algorithm for all the graphs and by upto 4.2 times. *pbitMCE* is faster than TTT algorithm for most graphs, sometimes by a larger factor. The adjacency list variation of TTT algorithm, in general performs better for extremely sparse graphs(*roadnet-TX*, *roadnet-CA*, *europe-osm*). For these graphs, both ELS and *pbitMCE* are 3 to 4 times slower than TTT algorithm. However, for other graphs, TTT algorithm is slower than ELS and *pbitMCE* by a large factor(> 300). For smaller dense graphs, the TTT algorithm with adjacency matrix is expected to perform well compared to ELS and *pbitMCE*. The graphs in dataset 4 are examples of such graphs. The ELS algorithm is

Table 3: Experimental results on different datasets

dataset	$\|V\|$	$\|E\|$	$\|C\|$	d	M	TTT	ELS	$pbitMCE\text{-}b$	$pbitMCE$
wiki-Talk	2,394,385	4,659,565	86,333,306	131	100,029	>1000	143.12	72.74	38.85
cit-Patents	3,774,768	16,518,947	14,787,032	64	793	23.74	58.76	24	19.77
soc-Pokec	1,632,803	22,301,964	19,376,873	47	20,518	94.15	102.88	34.45	31.69
as-Skitter	1,696,415	11,095,298	37,322,355	111	35,455	2725.48	103.94	39.37	26.09
amazon0601	403,394	2,443,408	1,023,572	10	5,504	2.09	5.18	1.8	1.7
web-BerkStan	685,231	6,649,470	3,405,813	201	168,460	70.64	16.1	5.89	5.65
web-Google	875,713	4,322,051	1,417,580	44	12,664	9.08	9.17	3.4	3.18
roadnet-TX	1,379,917	1,921,660	1,763,318	3	24	0.62	4.32	2.12	2.06
roadnet-CA	1,965,206	2,766,607	2,537,996	3	24	0.9	6.18	3.11	2.98
coPapersDBLP	540,486	15,245,729	139,340	336	3,299	44.11	35.08	21.04	20.17
coPapersCiteseer	434,102	16,036,720	86,303	844	1,188	47.91	42.81	30.34	29.4
channel-b050	3,774,768	33,037,894	61,142,398	9	18	44.01	65.37	30.92	26.57
europe-osm	50,912,018	54,054,660	53,933,091	3	13	21.68	140.33	72.65	69.36
rmat-10m-100m	10,000,000	100,000,000	101,384,668	22	2,208	363.97	521.35	179.86	168.73
er-1m	1,000,000	94,977,606	92,662,224	160	262	458.34	615.95	270.13	261.28
biogrid-yeast	6,008	156,945	738,613	64	2,557	1.14	1.47	10.15	0.53
random-100-0.9	100	4,473	240,998,654	81	97	105.76	262.51	62.39	62.44
random-300-0.6	300	27,013	72,454,791	162	204	49	160.78	80.52	45.41
random-500-0.5	500	62,571	103,686,974	225	291	76.53	274.97	140.05	79.61
random-1000-0.3	1,000	149,998	15,671,489	266	349	12.86	48.36	20.05	13.63
phat-300-2	300	21,928	79,917,408	98	229	39.78	104.63	46.46	25.75
m-m-51	51	1,224	129,140,163	48	48	21.12	43.51	10.51	10.69

slower than the TTT algorithm for all the graphs in the dataset. However, *pbitMCE* is faster than TTT algorithm for most graphs including the Moon-Moser graph, *m-m-51*, which is a highly dense graph with largest number of maximal cliques possible among all graphs with 51 vertices. *pbitMCE* is slower for two graphs(*random-500-0.5*, *random-1000-0.3*) but by a very small factor(< 1.1). Also, it can be seen from the table that the optimization improves the performance for all the graphs tested and the improvement is significant for some of the graphs(*wiki-Talk*, for example) depending on the structure of the search trees explored. Note that the timing results shown for all the approaches do not include the time taken to read the input file and constructing the adjacency list or matrix. Also, the timing results do not include the time taken to write the output to a file.

3.5 PARALLEL METHODOLOGY AND EXPERIMENTAL RESULTS

The *pbitMCE* algorithm clearly defines multiple tasks, each corresponding to a vertex. Thus, the task of parallelizing the code is as simple as distributing the vertices between different processing units. However as the workload corresponding to the vertices can widely vary, balancing the workload is a challenging issue that needs to be addressed.

3.5.1 LOAD BALANCING

Most of the real world graphs are power-law degree distributed i.e few vertices have relatively very high degree compared to the majority of vertices. For many graph problems, it is highly challenging to achieve a balanced workload for these graphs. *pbitMCE* relies on degeneracy ordering and scheduling for balancing the workload between the processing units. Note that, in our parallelization results, we didn't include the degeneracy ordering. We only focus on the enumeration process as degeneracy ordering is a pre-processing step and parallel computation of degeneracy ordering is discussed in Chapter 2.

Degeneracy

As we have seen, the ELS and *pbitMCE* approaches use degeneracy ordering of vertices before starting the enumeration process. In the parallel context, the degeneracy ordering has an added advantage. It facilitates a balanced distribution

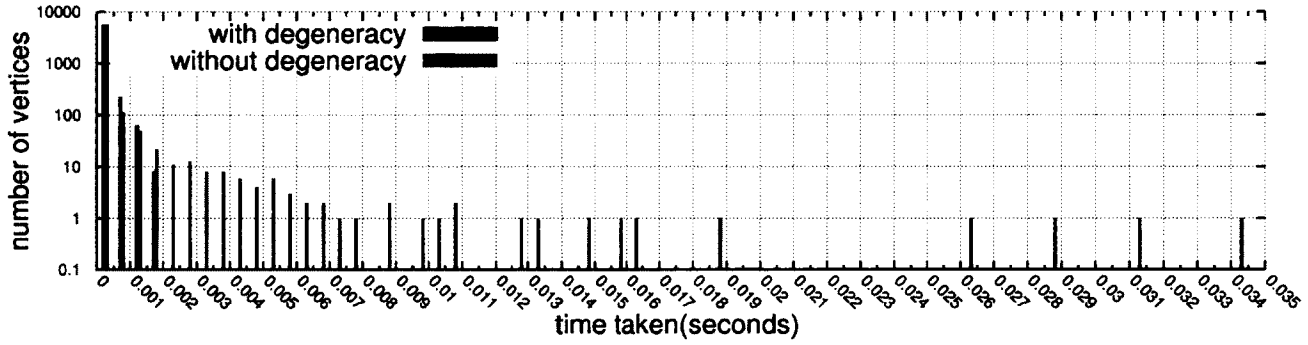


Figure 25: Plot showing the impact of degeneracy on load balance in *biogrid-yeast* graph

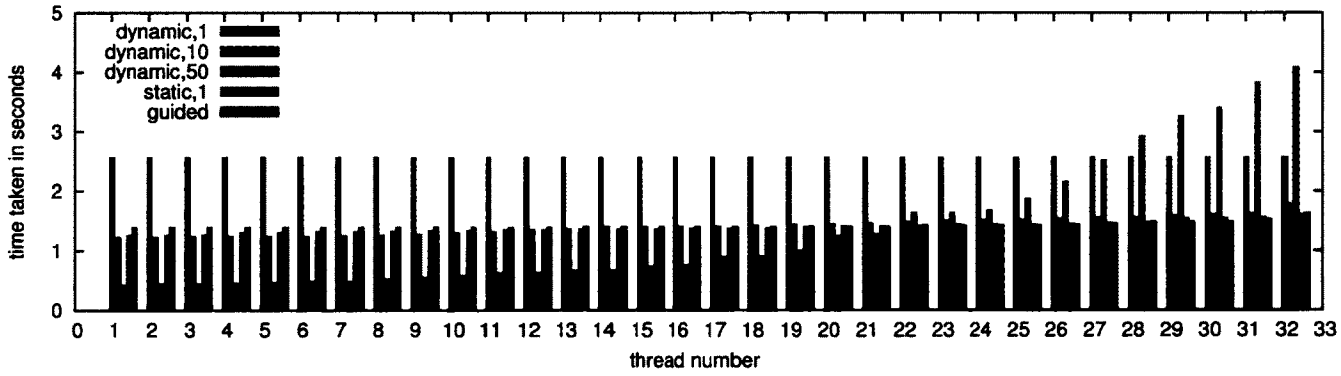


Figure 26: Plot showing the impact of different scheduling types on the load balance and overall time taken

of workload. Observe from the TTT and ELS algorithms that a maximal clique C is generated by exploring the tree corresponding to the least numbered vertex in C. If a vertex with high degree is placed early in the ordering, its candidate list p would be large and so exploring its search tree takes more time compared to the other vertices. This problem is alleviated by using degeneracy ordering. The degeneracy ordering places the vertices in such a way that no vertex has more than d neighbors that come later in the ordering. This reduces the candidate list size and so the workload corresponding to large degree vertices and increases the workload corresponding to their neighbors with smaller degree. The impact of degeneracy on load balancing can be clearly seen from Figure 25. The plots show the distribution of time taken by the vertices of *biogrid-yeast* graph. Notice the difference in time taken by different vertices. When degeneracy ordering is not used, there are few vertices that take considerably more amount of time compared to the vast majority of vertices. Whereas, when degeneracy ordering is used the difference between time taken by different vertices is greatly diminished.

Scheduling

The degeneracy ordering alleviates the problem of workload imbalance between all the vertices. However, the problem is not completely eliminated. The type of scheduling used for distributing the vertices between the computing nodes also impacts the workload imbalance. OpenMP provides different types of scheduling options including dynamic, static and guided. Since the vertices have varying amount of workload, the dynamic scheduling is the most appropriate in this case. However, dynamic scheduling results in huge synchronization overhead. Static scheduling, on the other hand, has no synchronization overhead, but might result in load imbalance. We have experimented with different scheduling options with varying chunk sizes. Guided scheduling is similar to dynamic scheduling except that the chunk sizes decrease as the number of vertices to be assigned decreases. The plot in Figure 26 shows the time taken by each thread using dynamic scheduling with chunk sizes 1, 10 and 50, static and guided scheduling. The timing results are obtained using the *wiki-Talk* graph. It can be seen that, the dynamic scheduling with chunk size 1 resulted in a much balanced load but took longer time than other scheduling options. Dynamic scheduling with chunk size 50 resulted in a wide variation in time taken by the threads. The static and guided scheduling resulted in similar performance with

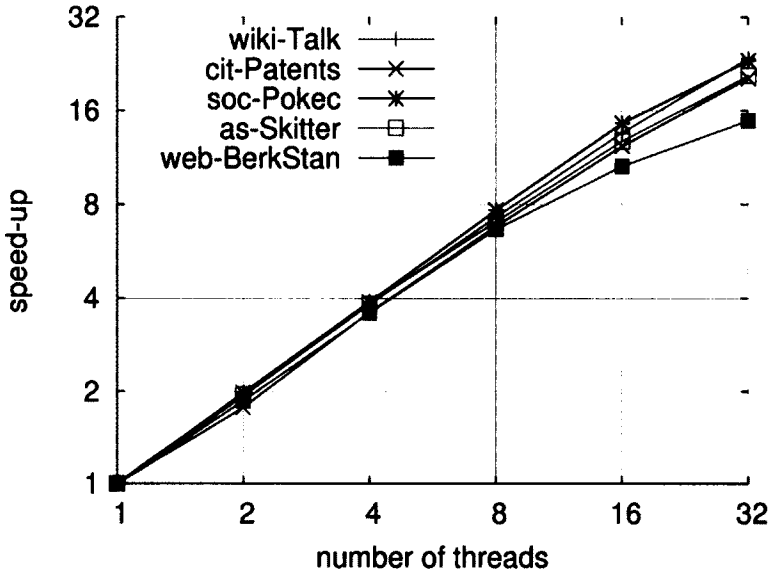


Figure 27: Scalability plot for dataset 1

only a slight variation in time taken by the threads.

We have experimented with other load balancing methods like work stealing. Any attempt for further load balancing involves communication between the processing units through data sharing. This resulted in synchronization overhead and significantly degraded the performance.

3.5.2 SCALABILITY

Figures 27 28, 29 and 30 plot the speed-up obtained for the graphs in the four datasets. The speed-up is defined as the ratio of processing time taken by t threads over 1 thread. For each graph, we have chosen the scheduling mechanism that resulted in maximum speed-up. Our algorithm scales well to all the 32 threads available giving a speed-up of upto 29. The only exception is the *m-m-51* graph which is a Moon-Moser graph [58]. The graph has 51 vertices with degeneracy of 48. The minimum time that *pbitMCE* takes to finish the enumeration using sufficiently large number of processing units is $max(T_v)$, for $v \in V$, where T_v is the time taken to perform computation corresponding to a vertex v. For *m-m-51* this minimum time is achieved with 8 threads and so *pbitMCE* cannot scale-up beyond 8. However, *m-m-51* is a highly dense graph and these kind of graphs are uncommon in practice.

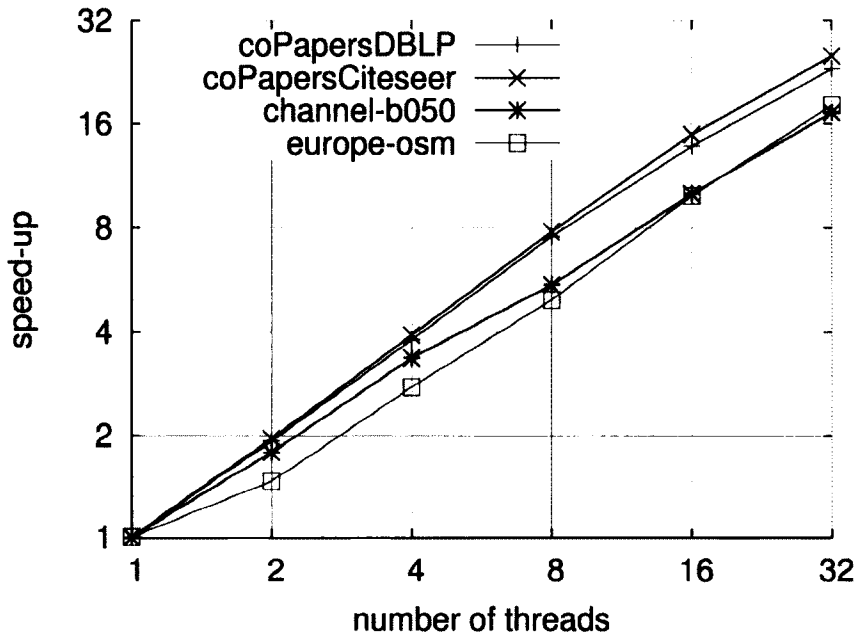


Figure 28: Scalability plot for dataset 2

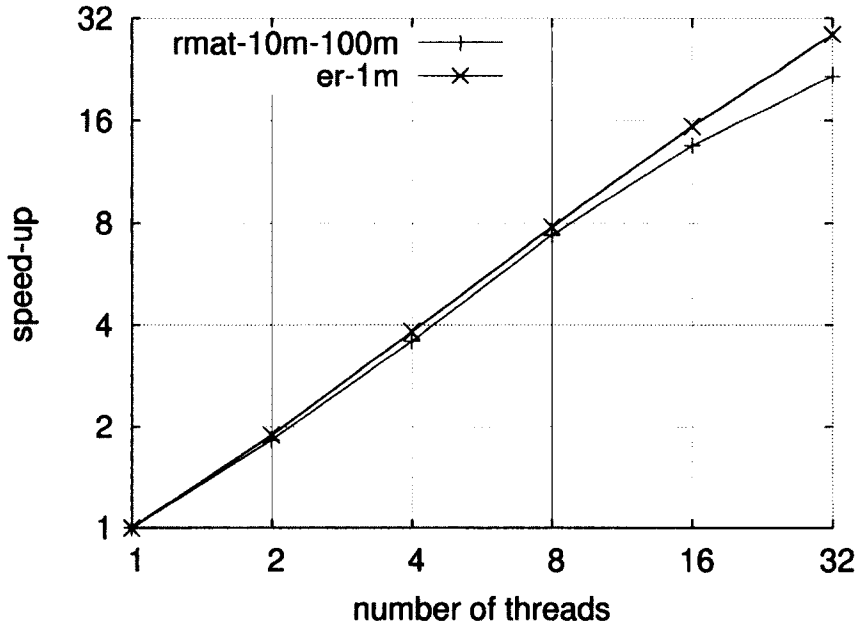


Figure 29: Scalability plot for dataset 3

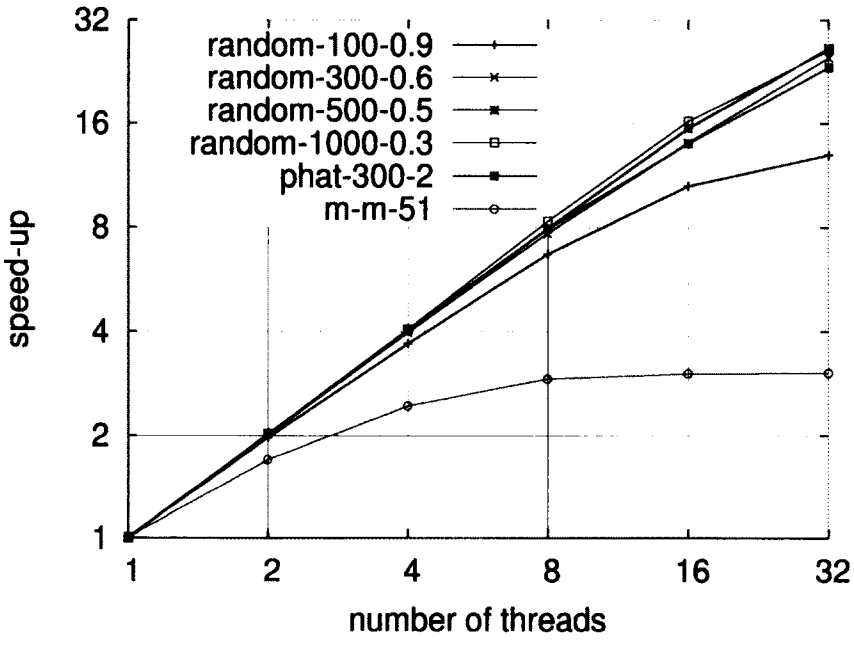


Figure 30: Scalability plot for dataset 4

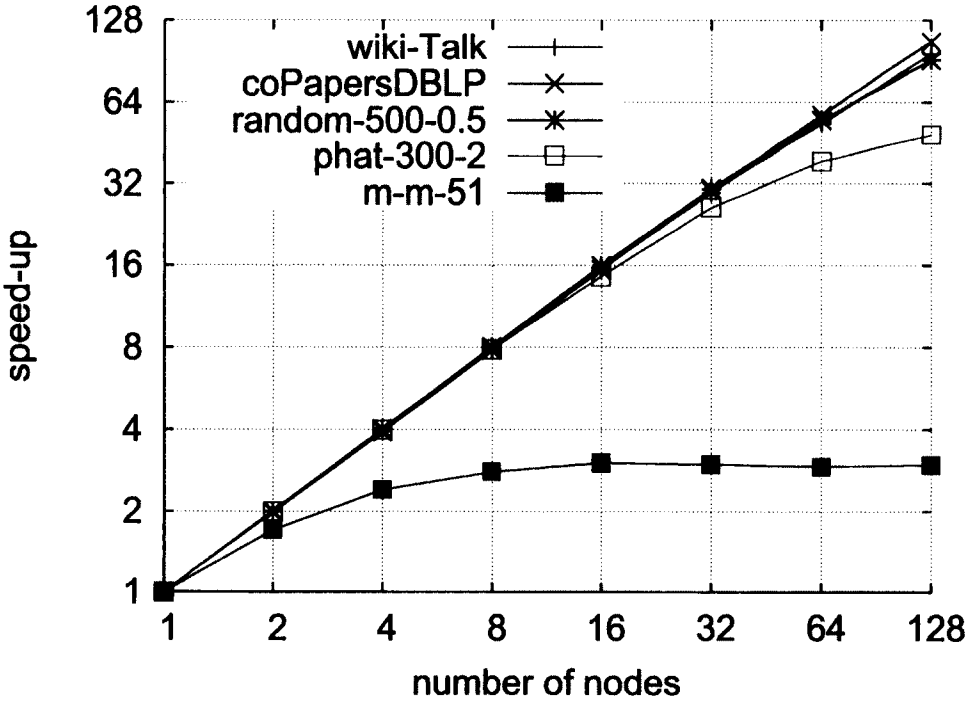


Figure 31: Scalability on distributed memory architecture

3.5.3 RESULTS ON DISTRIBUTED ARCHITECTURE

We have also performed some experiments on a distributed system to show that the *pbitMCE* algorithm can also be used in a distributed environment. The experiments were performed on a cluster with 32 nodes, each with Intel Xeon E5504 processor with 4 cores. MPI is used for inter-process communication and the code is compiled using mpicc. The pre-processing is done by all the processes and the data i.e. *pre-* and *post*-adjacency lists are stored locally at all the nodes. The task of enumeration is distributed among the processes and workload is dynamically balances. The initial distribution of workload is done by equally assigning the vertices to them. The processes independently work by exploring the search trees corresponding to the vertices assigned to them. When a process completes the work assigned to it, it selects a computing node at random and requests for work. If it doesn't receive work, it requests another process. The process which receives the request shares its work by assigning some of its vertices to the requesting node. A process terminates when it completes all its assigned work and doesn't receive work from any of the other processes. Figure 31 shows the scalability results for some of the graphs from the four datasets. It can be seen that *pbitMCE* scales upto 106 times using 128 processes. The speed-up is poor for *m-m-51* graph for the same reason as in the case of multicore architecture.

3.6 *PBITMCE* ON HADOOP FRAMEWORK

In the previous section, we have discussed how to implement *pbitMCE* algorithm for distributed environment using MPI. The implementation assumes that the each node in the cluster has a copy of the entire graph. For very large graphs, the assumption may not be practical. In this section, we discuss the implementation of *pbitMCE* algorithm for a distributed environment where the graphs is partitioned and distributed among the nodes.

We have seen in Section 3.3 that in the *pbitMCE* algorithm, before the enumeration process, the vertices are ordered based on degeneracy. Degeneracy ordering can be computed in linear time. However, when the graph is distributed, it is challenging to compute the ordering as it requires extensive communication between the nodes. In some cases, computing the degeneracy ordering can take significant time than the time saved in enumeration by using the ordering. In such cases, a different ordering,

like degree ordering, which can be quickly computed may be a better choice. In this section, we experimentally study the impact of various orderings on the performance of *pbitMCE* algorithm in the context of MapReduce framework. Our experiments show that the degree ordering performs comparable to the degeneracy ordering for most graphs.

As we have seen, MCE is an extensively studied problem. There are many algorithms both sequential and parallel algorithms proposed for MCE. Wu et al. [82] proposed an approach for MCE using MapReduce framework. The approach is based on a variation of BK algorithm. The approach partitions the enumeration process into tasks, each task corresponding to a vertex. All the data that is required for a task is first collected at a node. This is achieved by emitting the adjacency list of a vertex to all its neighbors. This can result in enormous amount intermediate data transfer between nodes. Also, the approach does not address the issue of unbalanced load which is crucial, especially for power-law degree distributed graphs.

To aid with processing of large scale data on clusters, many frameworks have been developed including MapReduce [13], Pregel [15] and Graphlab [16]. These frameworks have built-in capabilities including fault tolerance, storage and retrieval of large files, communication and synchronization between processes. These frameworks also simplify the task of parallel programming which otherwise is very hard. However, it is not always easy to modify the approaches to use a framework.

MapReduce is a widely used programming model for processing large data sets in parallel on a cluster. A MapReduce program consists of two key functions: map and reduce. In the map function, the input values are converted into key value pairs. These key value pairs are then shuffled and sorted and a key with all its values are input to a reduce function. The reduce function then produces the desired output. Hadoop is a software framework that uses MapReduce programming model for large scale data processing. Hadoop uses its own file system called Hadoop distributed file system (HDFS) to effectively store and retrieve the files.

One of the major challenges in implementing *pbitMCE* on Hadoop framework is to compute the degeneracy ordering of the vertices. As discussed in Section 3.3.1, the degeneracy ordering can be obtained in linear time by repeatedly removing the smallest degree vertex and its edges from the graph. Though it only takes linear time, the time taken can be significant for larger graphs. Also generating degeneracy ordering in a distributed environment where the graph is partitioned across multiple

computing nodes is challenging. Alberto et al. [36] have proposed a distributed algorithm for k-core decomposition, which is equivalent to degeneracy ordering. The proposed approach has been used in an evaluation study of different frameworks including Hadoop. The results presented in [83] show that the Hadoop framework takes longer time when compared to other frameworks. The results show that for the *web-BerkStan* graph(Table 3) the degeneracy ordering takes more than 2000 seconds on a cluster with 32 nodes while the time for clique enumeration takes less than 70 seconds on a much smaller cluster. Clearly, using degeneracy ordering in this case is not advisable. However, *pbitMCE* can be easily modified to use any vertex ordering. Some of the choices for vertex ordering include degree ordering, random ordering, original ordering and partial degeneracy ordering.

pbitMCE produces all and only the maximal cliques without any duplication irrespective of the vertex ordering used. In [32], Eppstein et al. proved the correctness of the ELS algorithm with respect to the degeneracy ordering. The same proof can be applied to any given ordering of vertices. A clique C will be generated by the minimum ordered vertex v in C. For any other vertex in C, v will be in the not list ,X, and so the clique C will not be repeated. Compared to the degeneracy ordering, the other orderings i.e. degree, random, original and partial degeneracy orderings can be generated in significantly less time. We present the implementation of *pbitMCE* using Hadoop framework and empirically compare the performance of different orderings. Note that we only focus on comparing the performance of *pbitMCE* using different orderings to see if the degeneracy ordering can be replaced by other ordering. We don't compute the ordering using the Hadoop framework, instead we use a precomputed ordering stored in a map file.

3.6.1 IMPLEMENTATION

We have seen that *pbitMCE* works by exploring multiple search trees, each corresponding to a vertex in the graph. We have also seen that *pbitMCE* uses *pbam* to perform the enumeration. To construct *pbam* corresponding to a vertex v the list of neighbors of v i.e $N(v)$ and the *post* adjacency list of each u in $N(v)$ i.e. $postN(u)$ for each $u \in N(v)$ are required. To extract the required data from the input, we use three different MapReduce jobs. Two files, one containing the graph with each edge in a line, and the other containing a map file with a vertex of the graph and its new order in each line are given as input.

1: **function** MAPPER1$_1$
Input: $key = u$, $value = v$
2: emit(u, v)
3: emit(v, u)
4: **end function**

5: **function** MAPPER1$_2$
Input: $key = u$, $value = O_u$
6: emit(u, O_u)
7: **end function**

8: **function** REDUCER1
Input: $key = u$, $values = (O_u, v_1, v_2 \ldots)$
9: $N_u = \emptyset$
10: remove O_u from *values*
11: **for** each vertex v in *values* **do**
12: add v to N_u
13: **end for**
14: emit((u, O_u), N_u)
15: **end function**

Figure 32: First job

The first job given in Algorithm 4 is a simple and commonly used job in graph algorithms. The input files containing edge information and mapping information are processed and converted to adjacency lists. The job consists of two mappers and a reducer. The first mapper receives as input a set of lines from the file containing the graph while the second mapper is input a set of lines from the map file containing the new order of each vertex. The map methods simply forward the input keys and values that they receive. The reducer receives as input a vertex, its set of neighbors and also its new order. The reducer converts the set of values into adjacency list and emits the vertex and its new order along with its adjacency lists.

After the first job, each record contains, a vertex, its new order and its adjacency list. The second job is used to obtain the new order of each vertex in the adjacency list. This data is required to partition the adjacency list into *pre* and *post* adjacency lists. As we have seen in the Section 3.3.2, partitioning is done to avoid processing of unnecessary data as we only need the *post* adjacency lists to construct *pbam*. Note that we can skip using the second job and instead perform partitioning in the third

```
1: function MAPPER2
   Input: key = (u, O_u), value = N_u
2:     for each vertex v in N_u do
3:         emit(v, (u, O_u))
4:     end for
5:     emit(u, (u, O_u))
6: end function

7: function REDUCER2
   Input: key = u, values=((u, O_u), (v_1, O_v1), (v_2, O_v2)...)
8:     preN_u = ∅
9:     postN_u = ∅
10:    remove (u, O_u) from values
11:    for each (v, O_v) in values do
12:        if O_v > O_u then
13:            add v to postN_u
14:        else
15:            add v to preN_u
16:        end if
17:    end for
18:    emit((u, O_u), (preN_u, postN_u))
19: end function
```

Figure 33: Second job

job explained later in the section. However, this results in enormous of data that needs to be communicated. For example, in the wiki-Talk graph(Table 3), one of the vertices has a degree of more than 100,000 and its adjacency list requires 400K bytes. If the adjacency list is not partitioned, the whole adjacency list needs to be emitted(in the third job) for each vertex in the adjacency list, i.e the data of size 100,000 * 400K bytes = 40G bytes are to be emitted. If partitioning is used based on degeneracy ordering, since the degeneracy of the graph is 131, the maximum size of *post* adjacency list would be 524 bytes and only 100,000 * 524 bytes = 52M bytes are to be emitted. Hence, partitioning plays a significant role in minimizing the data required for communication.

To generate the required information the second job uses a mapper and a reducer(Figure 33). For each input that the mapper receives which consists of a vertex u, its new order O_u, and its adjacency list N_u, the mapper emits the value (u, O_u) with each vertex in N_u as the key. Each record that is input to the reducer contains

```
1: function MAPPER3
   Input: key = (u, O_u), value = (preN_u, postN_u)
2:     for each vertex v in preN_u ∪ postN(u) do
3:         emit(v, (u, O_u, postN_u))
4:     end for
5:     emit(u, (u, O_u, postN_u)))
6: end function

7: function REDUCER3
8:     Input: key = u, values=((u, O_u, postN_u), (v_1, O_v1, postN_v1),
   (v_2, O_v2, postN_v2)...)
9:     P = ∅
10:    X = ∅
11:    remove (u, O_u, postN_u) from values
12:    for each (v, O_v, postN_v) in values do
13:        if O_v > O_u then
14:            add v to P
15:        else
16:            add v to X
17:        end if
18:    end for
19:    B = constructpbam(P, X)
20:    TTT(P, X, {u}, B)
21: end function
```

Figure 34: Third job

a vertex u, its new order O_u, and the list of vertices in its adjacency list along with their new orders. Using this data the reducer partitions the vertices in its neighbor list into *pre* and *post* adjacency lists denoted as $preN_u$ and $postN_u$ respectively. All the vertices that have new order greater then O_u are added to the *post* adjacency list and all the other vertices are added to the *pre* adjacency list. The generated *pre* and *post* adjacency lists are forwarded to the next job.

The third job is the final job and it includes all the processing of data and enumeration of cliques. The job includes a mapper and a reducer(Figure 34). While the mapper is simple, the reducer performs the actual work of enumeration. Each input to the mapper consists of a vertex u, its new order O_u and its *pre* adjacency list $preN_u$ and *post* adjacency list $postN_u$. The mappers forwards the value $(u, O_u, postN_u)$ to all the vertices in the *pre* and *post* adjacency lists. Each input to the reducer then

Table 4: Degeneracy vs *k-degree*

dataset	degeneracy	*k-degree*
cit-Patents	64	77
wiki-Talk	131	340
copapers	336	336
web-Berkstan	201	201
soc-LiveJournal1	372	686

contains all the information required to construct *pbam* corresponding to a vertex u and perform enumeration. For each $(v, O_v, postN_v)$ in the value list, the vertex v is added to the candidate list P if $O_v > O_u$, otherwise it is added to the not list X. *pbam* is then constructed and used in the enumeration process. The maximal cliques generated can be output to a file.

3.6.2 ANALYSIS

We experiment using different vertex orderings in the *pbitMCE* algorithm. However, our main focus is the degree ordering. To aid with the analysis of *pbitMCE* using degree ordering, we introduce a parameter called *k-degree*. *k-degree* of a graph is defined as the smallest value k such that every vertex v of the graph has at most k neighbors that have degree greater than or equal to its degree. We denote *k-degree* by k. In degree ordering, since the vertices are ordered by non decreasing order of their degrees, for any vertex v in the graph, there can be no more than k neighbors that come later in the ordering. Therefore, in degree ordering the size of P is limited by k as in the degeneracy ordering the size of P is limited by d. Therefore d can be replaced by k in the analysis of the *pbitMCE* algorithm given in Section 3.3.6. This results in a time complexity of $O(kn3^{k/3})$. However, it is not clear how the two values, d and k, can be compared. So, we have done an empirical comparison of both the values. Table 4 shows the values of degeneracy and *k-degree* for the graphs described in Section 3.4. It can be seen that the $k - degree$ value is greater than or equal to the degeneracy for all the graphs.

3.6.3 EXPERIMENTAL RESULTS

We have performed our experiments using three different clusters. For the initial

Table 5: Time taken(in seconds) for enumeration using various orderings

dataset	degeneracy	degree	partial degeneracy	random	original
cit-Patents	129	126	125	211	153
wiki-Talk	133	130	117	-	-
copapers	351	330	383	515	500
web-Berkstan	55	48	481	16690	>17000

experiments to compare various orderings we have used a Hadoop cluster consisting of 8 heterogeneous computing nodes. All but one node have 4 cores and 4GB of RAM and the other node has 2 cores and 2 GB RAM. The total storage capacity of the cluster is 1TB. To compare the scalability of degree and degeneracy ordering we have used Amazon web services. We ran our experiments on a cluster with 64 standard EC2 compute units for slaves and 2 EC2 compute units for master. To experiment with very large graphs(soc-LiveJournal1) we have used an Amazon cluster with 512 EC2 compute units for slaves and 4 compute units for master. We ran our experiments using graphs from two different collections: Stanford large network data collection [40] and University of Florida sparse matrix collection [78]. The graphs *wiki-Talk*, *web-BerkStan*, *cit-Patents*, *soc-LiveJournal* are from the Stanford collection and the graph *coPapersDBLP*is from the University of Florida collection. The description of these graphs can be found in Section 3.4.

We have precomputed all the orderings i.e degeneracy, degree, partial degeneracy, random and original orderings and stored in map files. The degeneracy ordering is computed using the BZ algorithm explained in Chapter 2. Degree ordering is computed by sorting the vertices in non decreasing order of their degree. Partial degeneracy order is obtained by using the distributed k-core decomposition algorithm [36] and limiting the number of iterations to 10. We have used the shuffling algorithm in [84] to generate random ordering. The original ordering is the given ordering of the vertices.

Two files are given as input: the graph file containing one edge of the graph in each line, and a map file containing in each line a mapping of a vertex of the graph from the original order given to a new order. Given the input files, the approach uses three MapReduce jobs to generate the required output, i.e. maximal cliques. The first job and the second job are used to collect the adjacency list of each vertex and

the new order of the vertices in the adjacency list. These two jobs do not depend on the ordering used. These jobs take the same time irrespective of the order chosen. Since they do not contribute to the comparison of different orders, we ignore the timing results of the two jobs.

We have performed our initial experiments using a small cluster with 8 nodes. Table 5 shows the results of different ordering on some of the graphs. The results were obtained by using 8 reducers. The values in the table represent the wall clock time taken in seconds by the third job including the time for map, shuffle, sort and reduce. It can be seen from the table that the degree ordering performs comparable to the degeneracy ordering while random and original orderings perform significantly poorer compared to the other orderings. For the *wiki-Talk* graph, the original and random ordering ran out of disk space due to the enormous size of data that needs to be communicated. Partial degeneracy order performance is comparable to degeneracy ordering for some graphs but for other graphs it is slower by a large factor. For further experiments we have focused only on degree and degeneracy ordering. To compare the scalability of both the orderings we have performed our experiments on Amazon cluster with 64 EC2 compute units. To better compare the two ordering we measured the cumulative time taken by a reducer to execute the reduce function. Let t_i be the cumulative time spent by a reducer i in the reduce function and $maxTime = max\{t_i | 1 \leq i \leq n\}$ where n is the number of reducers. Figures 35 through 38 shows the results obtained on different graphs using the degeneracy and degree ordering. The time in seconds along y-axis represents the $maxTime$. Note that, this does not include the time taken for map, shuffle and sort phases. It can be seen from the plots that both degeneracy and degree orderings perform comparably for all the graphs except the *wiki-Talk* graph for which the degeneracy ordering performs slightly better.

To further evaluate the two orderings, we have experimented using the graph soc-LiveJournal1 which has 4.8 million nodes and 42.8 million edges. While experimenting we found that the graph has immensely dense subgraphs and these subgraphs contain enormous number of maximal cliques. We found that each such subgraph has more than a trillion maximal cliques. Exploring all the maximal cliques for such a graph requires very large amount of computing resources and takes a huge amount of time. Since our focus is to compare the two orderings, we have applied a rule to shorten the time taken for enumeration. During enumeration, if the clique size goes

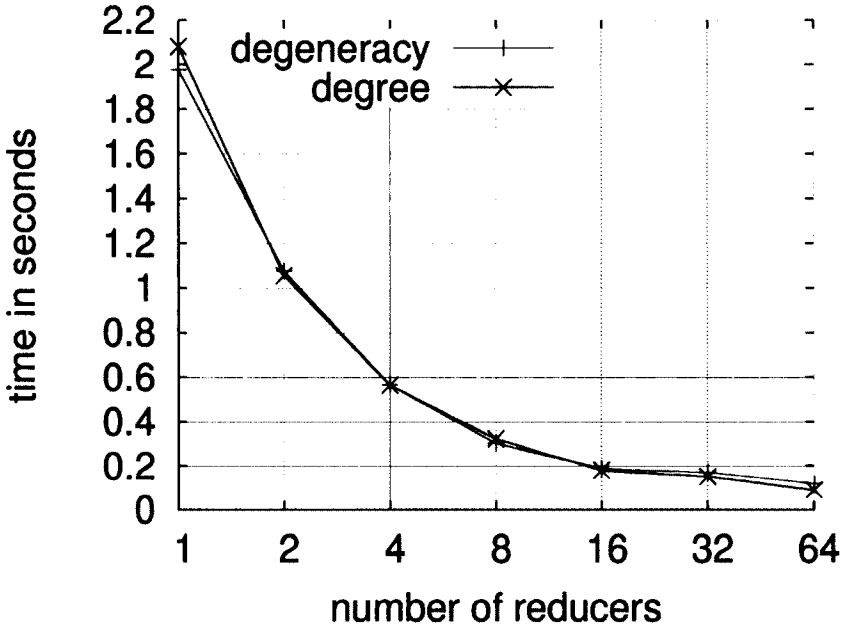


Figure 35: Comparison of time taken by *pbitMCE* using different orderings for *cit-Patents* graph

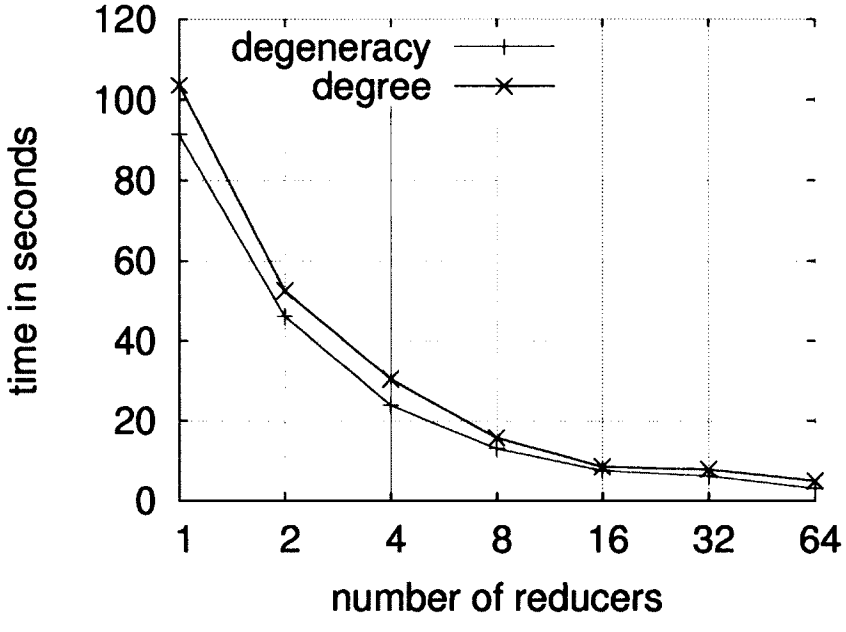


Figure 36: Comparison of time taken by *pbitMCE* using different orderings for *wiki-Talk* graph

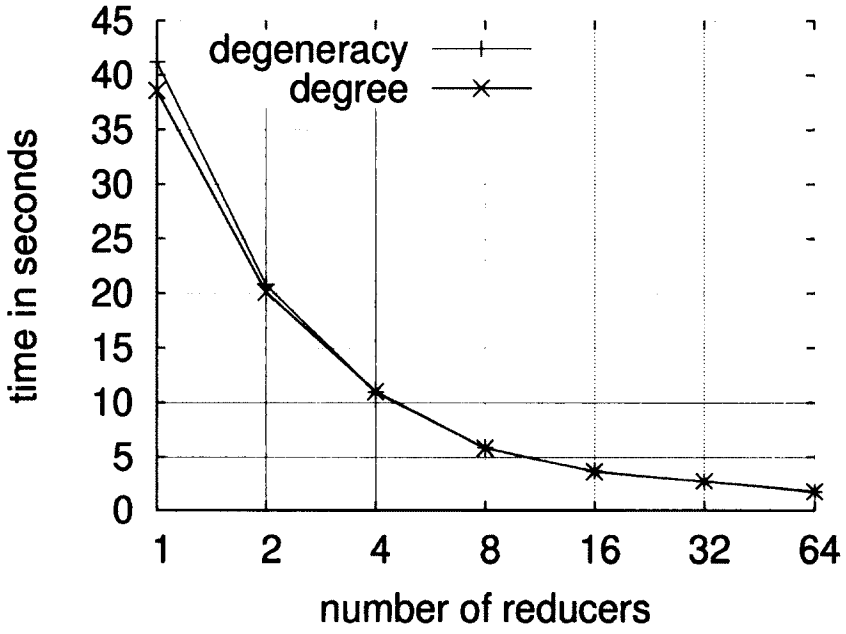


Figure 37: Comparison of time taken by *pbitMCE* using different orderings for *web-BerkStan* graph

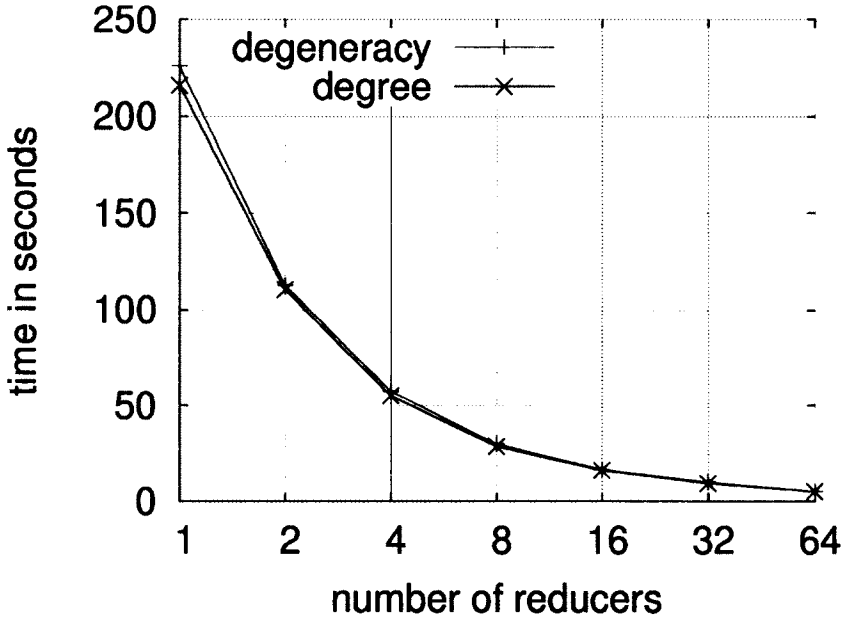


Figure 38: Comparison of time taken by *pbitMCE* using different orderings for *co-PapersDBLP* graph

beyond 100 i.e when $R > 100$ and if the remaining candidate vertices are less than 100 i.e $P < 100$, instead of further enumerating, we increment the clique count and backtrack. Note that this will not generate maximal cliques, but only a portion of each maximal clique. However, it will generate all the maximal cliques of size less than 100. Also, the degree and degeneracy orderings might produce different number of cliques. We ran the experiment using an Amazon cluster with 512 standard EC2 compute units with 512 reducers. The degeneracy ordering yielded 583.8 billion cliques and took 4.5 hours while the degree ordering generated 513.7 billion cliques and took 6.2 hours.

The experimental results obtained reflect the analysis in Section 3.6.2. For graphs with comparable degeneracy and $k - degree$ values, both the degree and degeneracy ordering result in similar performance, while for the graphs for which $k - degree$ is greater than degeneracy by a large factor, *soc-LiveJournal* for example, the degree ordering results in poorer performance compared to the degeneracy ordering.

3.7 SUMMARY

In this chapter, we define a *task-set* and discuss how the size of *task-set* can influence locality. We present a technique called *task-set reduction* which helps in improving the temporal locality of an algorithm by reducing the size of *task-set*. We demonstrate the effectiveness of the technique by using it to develop an algorithm for maximal clique enumeration.

Finding maximal cliques is a fundamental problem arising in many areas. In this chapter, we define the maximal clique enumeration problem and present some applications. We briefly discuss some MCE algorithms existing in the literature. The state-of-the-art algorithm for MCE referred to as ELS algorithm is described and its memory locality with respect to the *task-set* size is analysed. We propose a new algorithm, called *pbitMCE*, which uses a bit-based data structure to significantly reduce the *task-set* size. We have implemented the algorithm and we compare the results with the ELS algorithm and TTT algorithm. We have shown that our approach is faster than both the approaches for most graphs and slower only by a small factor for few graphs. We have implemented a parallel approach on a multicore machine and showed that it is scalable giving a speed-up of upto 29 times using 32 cores. We have also implemented the algorithm on distributed memory architecture and showed that *pbitMCE* scales upto 106 times using 128 processes.

CHAPTER 4

TRIANGLE LISTING

Memory Locality is a key aspect in the performance of an application. In the previous chapters, we have discussed two techniques for improving the memory locality in graph algorithms: *access transformation* and *task-set reduction*. We have also seen how the techniques are applied to the k-core decomposition and maximal clique enumeration problems. In this chapter, we show another example application, triangle listing, which uses both the techniques. Triangle listing algorithms have highly random memory access pattern and also large *task-set* sizes. Therefore, improving the memory locality can result in significant performance benefits. Many algorithms for the triangle listing problem exist in the literature. Out of those, the *edge-iterator* algorithm is the most widely used algorithm. The algorithm repeatedly accesses the adjacency lists of the vertices in random order resulting in poor memory locality. To apply the *access transformation* technique, the algorithm should have limited number of iterations. But the *edge-iterator* algorithm does not satisfy the property. The *task-set* of the *edge-iterator* algorithm is very large and also cannot be compressed using bit representation or other techniques. Therefore, the *task-set reduction* technique is not easily applicable to the *edge-iterator* algorithm.

We propose a new algorithm, called *window-iterator*, which is a modification of the *edge-iterator* algorithm. The *window-iterator* algorithms uses both the *access transformation* and *task-set reduction* techniques to improve locality. Unlike the *edge-iterator* algorithm, the *window-iterator* algorithm has limited number of iterations, each iteration working on a smaller *task-set*.

4.1 DEFINITION AND NOTATIONS

Given an undirected simple graph $G(V, E)$, the triangle listing problem is to find a set $T = \{(u, v, w) | u, v, w \in V and (u, v), (v, w), (w, u) \in E\}$. Note that the set T is a set of cliques of size 3. Triangle counting problem, a variation of triangle listing problem, is to find the number of triangles in the graph. The neighborhood of a vertex v is denoted by $N(v)$ and the degree of v is denoted by $deg(v)$. The number of triangles in a graph is denoted by Δ.

4.2 APPLICATIONS

The triangle counting/listing problems are of high interest in network analysis applications. They are used in finding a key statistical property of a graph called clustering coefficient [85]. Also they are used in finding transitivity coefficient [86][87][88], another key property of a graph. The triangle problems play an important role in bioinformatics in the study of motifs and protein-protein interaction networks [89][90]. Triangle listing is regarded as one of the fundamental graph mining problem. It is used for detecting sybil accounts and measuring content quality [91], detection of spamming activities, uncovering of hidden thematic relationships in web [91].

4.3 RELATED WORK

Triangle listing problem has been extensively studied and many algorithms exist in the literature [71][92][93][94][95][96][97][98]. In a recent study of listing algorithms by Mark et al. [92], it was shown that most listing algorithms have a common abstraction. They showed that the running time of nearly every triangle listing variant is in $O(a(G)m)$, where $a(G)$ is the arboricity of the graph and m is number of edges. It was shown that most triangle listing algorithms fall into one of the two categories: neighborhood intersection and adjacency testing.

Neighborhood intersection: The algorithms in this category iterate over all the edges. For each edge (u, v), the neighborhoods of u and v are intersected to get all the triangles that include the edge (u, v). The algorithms *edge-iterator* [93], *forward* [93] and *compact-forward* [94] belong to this category. To make the intersection efficient, the adjacency lists are first sorted. To further improve the efficiency of intersection, two other variants, *edge-iterator-hashed* [93] and *forward-hashed* [93] use hashing technique which required $O(m)$ extra space. Another variant, *new-vertex-listing* [94], performs the intersection using an extra $O(n)$ space. It uses a bit array to mark all the neighbors of a vertex u. The bit array is used in neighborhood intersections corresponding to all edges connected to u.

Adjacency testing: The algorithms in this category iterate over all the vertices. For each vertex u, every pair of vertices in its neighborhood are tested for adjacency. The algorithms *node-iterator* [93] and *node-iterator-core* [93] belong to this category. Both these algorithms use hashing to perform the adjacency testing in constant time

```
1: procedure edge-iterator(G(V, E))
2:     for each edge (u, w) in E do
3:         for each v in N(u) ∩ N(w) do
4:             output (u, v, w)
5:         end for
6:     end for
7: end procedure
```

Figure 39: The *edge-iterator* algorithm

and so require $O(m)$ additional space.

Ordering: The vertex ordering plays a key role in the performance of a triangle listing algorithm. Generally, the neighborhood of a vertex v, is divided into two parts: one containing the neighbors that come later than v in the ordering and the other containing the neighbors that come before v. The sizes of these neighborhoods depend on the vertex ordering. The most common orderings used by different algorithms include the degree ordering and degeneracy ordering. While some algorithms order the vertices by non-increasing order of degree, some algorithms use the reverse order. Mark et al. [92] presented a unified framework based on the ordering of vertices, the base algorithm(edge based or vertex based) and the amount of extra memory required. They have performed experiments using various graphs and the experiments revealed that their variant of neighborhood intersection algorithm that uses non-decreasing degree ordering along with an intersection strategy that requires $O(n)$ additional space outperformed all other algorithms.

As triangle listing algorithms are computationally expensive, many parallel algorithms have been proposed to tackle the massive volume of current graphs [99][100][101] [102][103][104] most of which use Hadoop framework.

4.3.1 *EDGE-ITERATOR* ALGORITHM

We have seen that many algorithms exist for triangle listing. The *edge-iterator* algorithm [93] has been widely used and the the studies [92] show that the *edge-iterator* and its variants outperform the other algorithms. Figure 39 shows the basic *edge-iterator* algorithm. It iterates over all edges and intersects the neighborhood of vertices connected by an edge. Different variants of the *edge-iterator* algorithm use different vertex ordering and different strategy for intersection. The variant we

```
1: procedure edge-iterator-deg(G(V, E))
2:     G'(V', E') = pre−process(G(V, E))
3:     for each vertex u in V' do
4:         ▷ Let {v_0, v_1, ..., v_{k−1}} = postN(u) where k = |postN(u)|
5:         for each vertex v_j in postN(u) do                    ▷ 0 ≤ j ≤ k − 1
6:             for each w in {v_{j+1}, v_{j+2}, ..., v_{k−1}} ∩ postN(v_j) do
7:                 output (u, v_j, w)
8:             end for
9:         end for
10:    end for
11: end procedure
```

Figure 40: The *edge-iterator* with degree ordering algorithm

use in this chapter, uses non-decreasing degree ordering i.e if $deg(u) < deg(v)$ then u comes before v in the ordering. We use a simple sort-merge approach which does not require any extra space for neighborhood intersection. We refer to this variant as *edge-iterator-deg*. This variant is equivalent to S1+1 variant discussed in [92]. The algorithm for *edge-iterator-deg* is presented in Figure 40.

Pre-processing

The original graph is preprocessed before starting the edge iteration. The degree ordering of the vertices is first computed. This can be done using count sort which can be performed in $O(n)$ time. Based on the ordering, the adjacency list(neighborhood) of each vertex v is partitioned into pre- and post-adjacency lists containing the neighbors that come before v in the ordering and that come after v respectively. We denote the adjacency list, pre- and post-adjacency lists of v by $N(v)$, $preN(v)$ and $postN(v)$ respectively. Let $\eta(v)$ denote the position of v in the degree ordering. A new graph $G'(V', E')$ is generated from the input graph $G(V, E)$, where $V' \subseteq V$ and $E' \subseteq E$, by replacing each vertex v by $\eta(v)$. Also, while generating the new graph we remove all the vertices(and their edges) with 0 and 1 degree as they do not belong to any triangle. We only use the post-adjacency lists during the edge iteration. Therefore we don't store the pre-adjacency lists. To make the intersection faster, the post-adjacency lists are sorted. We use Compressed Sparse Row(CSR) format to store the graph.

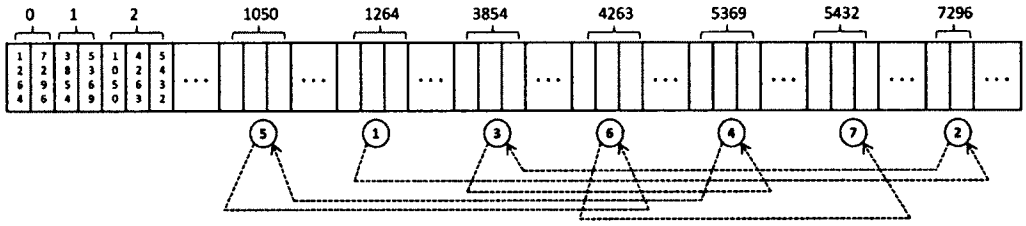


Figure 41: Memory access pattern of the *edge-iterator-deg* algorithm

Edge iteration

After pre-processing, the newly generated graph $G'(V', E')$ is used in edge iteration. We only use the post-adjacency lists during the edge iteration. The rationale behind using the non-decreasing degree order is to reduce the sizes of post-adjacency lists. In the non-decreasing degree vertex ordering, the large degree vertices are placed at the end and so their post-adjacency lists have smaller sizes. Note that this also facilitates in balancing the workload when using multiple processing units. For each vertex $u \in V'$, for each vertex v_j in its post-adjacency list, an intersection operation is performed between $postN(u)$ and $postN(v_j)$. Note that only the vertices that come after v_j in $postN(u)$ are included in the intersection operation.

4.3.2 ANALYSIS OF MEMORY LOCALITY

In this section, we analyze the memory locality of *edge-iterator-deg* algorithm focusing on access pattern and the size of *task-set*. In the rest of the chapter, we call the minimum numbered vertex in an edge as s-vertex and the other vertex in the edge as e-vertex. From the Figure 40, it can be seen that the algorithm processes all the edges corresponding to a s-vertex before moving to the next s-vertex. The memory access pattern for the *edge-iterator-deg* algorithm is depicted in Figure 41. In the example given in figure, the order in which the adjacency lists are accessed is as follows: $\{0, 1264, 0, 7296, 1, 3854, 1, 5369, 2, 1050, 2, 4263, 2, 5432\}$. As it can be seen, the access pattern of the s-vertices is sequential. However, the access pattern of the e-vertices is highly random. The task of edge iteration requires the adjacency lists of all the vertices and the lists are repeatedly accesses during the processing. Therefore,

the *task-set* for the *edge-iterator-deg* algorithm constitutes the entire graph. When the graph size is very large i.e significantly larger than the cache memory, the *edge-iterator* algorithm suffers from poor memory locality issues severely impacting the performance.

4.4 *WINDOW-ITERATOR* ALGORITHM

We have seen that the *edge-iterator-deg* algorithm have highly random access pattern and also large *task-set*. We propose a new algorithm, called *window-iterator*, which uses the access-transformation and task-reduction techniques to improve the memory locality. In the *edge-iterator-deg* algorithm, the e-vertices are accessed in a random order resulting in a large *task-set*. To reduce the *task-set*, in *window-iterator* algorithm, we limit the range of the e-vertices that a task accesses. We use multiple tasks, with a subset of consecutive e-vertices assigned to each task. We refer to this subset as a window. Also, the number of iterations is now limited to the number of windows, facilitating the use of *access transformation* technique. Note that, in the case of *edge-iterator-deg* algorithm, the number of iterations is $|V|$ which is too large for the *access transformation* technique.

Figure 42 shows the *window-iterator* algorithm. A window consists of vertices in the range $[wsv...wev]$ where wsv and wev stand for window start vertex and window end vertex. In each iteration, i.e for each window, all the vertices from 0 to wev are scanned and intersection is only performed when an e-vertex belongs to the current window. After processing all the edges with e-vertices in the current window, the window is moved to the next set of vertices.

The function *getWindowEndVertex* in Figure 42 returns the last vertex in the window. We measure the physical window size in terms of the number of bytes taken to store the adjacency lists of vertices in the window. The end vertex of the window is calculated such that the physical window size does not exceed a fixed value τ. The criteria for selecting the value of τ is discussed in Section 4.4.1.

4.4.1 MEMORY LOCALITY ANALYSIS

The memory locality of the *window-iterator* algorithm and the overall performance of the algorithm is governed by the physical size of the window chosen. When the value of τ is very large, i.e larger enough to fit the adjacency lists of all the vertices, then the *window-iterator* algorithm is equivalent to the *edge-iterator-deg*

```
1: procedure window-iterator(G(V, E))
2:     G'(V', E') = pre − process(G(V, E))
3:     wsv = 0
4:     wev = getWindowEndVertex(G', wsv)
5:     while wev < |V'| do
6:         for each vertex u in [0..wev] do
7:             ▷ Let {v_0, v_1, ..., v_{k−1}} = postN(u) where k = |postN(u)|
8:             for each vertex v_j in postN(u) and wsv ≤ v_j ≤ wev do
9:                 ▷ 0 ≤ j ≤ k − 1
10:                for each w in {v_{j+1}, v_{j+2}, ..., v_{k−1}} ∩ postN(v_j) do
11:                    output (u, v_j, w)
12:                end for
13:            end for
14:        end for
15:        wsv = wev + 1
16:        wev = getWindowEndVertex(G, wsv)
17:    end while
18: end procedure
19: function getWindowEndVertex(G', wsv)
20:    if wsv ≥ |V'| then
21:        return wsv
22:    end if
23:    wev = wsv + 1
24:    size = |postN(wsv)|
25:    while wev < |V'| and size + |postN(wev)| < τ do
26:        increment size by |postN(wev)|
27:        increment wev by 1
28:    end while
29:    return wev − 1
30: end function
```

Figure 42: The *window-iterator* algorithm

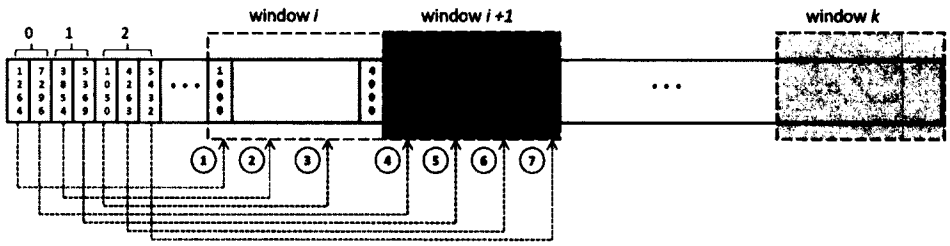


Figure 43: Memory access pattern of the *window-iterator* algorithm

algorithm. We have seen that the *edge-iterator* algorithm has poor memory locality for large graphs. On the other hand, if the value of τ is too small, then the number iterations will be large. Since in each iteration, all the vertices (that are $\leq wev$) need to be scanned, the time for scan operation can be dominating. The value of τ must be carefully chosen considering the cache memory size and the number of vertices in the graph. Based on experiments, we have chosen the maximum physical window size to be less than 16MB which is around 2/3 of the L3 cache memory size of the machine used for experiments.

Figure 43 shows the memory access pattern of the *window-iterator* algorithm. In the example given in figure, the memory access pattern is as follows: $\{postN(0)$, *window* $i, postN(1)$, *window* $i, postN(2)$, *window* $i, postN(0)$, *window* $i+1, postN(1)$, *window* $i+1, postN(2)$, *window* $i+1, postN(2)$, *window* $i+1\}$. As in *edge-iterator-deg* algorithm, the s-vertices are accessed sequentially. Since all the e-vertices processed in an iteration belong to the same window and the window size is chosen to be small enough to fit in cache, the order in which the e-vertices are accessed does not impact the performance.

4.4.2 IMPLEMENTATION

Efficient implementation is crucial for the performance of any algorithm. The data structures should be carefully chosen so that the memory latency is minimized. In the *window-iterator* algorithm, the scan operation can take considerable amount of time as it needs to access the adjacency list of each vertex involved in an iteration. We have used a special data structure to optimize the scan operation which resulted in significant performance benefits. In this section, we give the details of the implementation and the data structure.

In our implementation, we use compressed sparse row(CSR) format to store the graph. In this format, the adjacency lists of all the vertices are stored consecutively in one single array, i.e. all the neighbors of vertex 0, followed by all the neighbors of vertex 1 and so on. We refer to this array as *edgeArray*. Another array, referred by *vertexArray*, stores the starting indices of corresponding adjacency lists. For example, if the starting index of adjacency list of vertex v in *edgeArray* is i then $vertexArray[v] = i$. Note that, to access adjacency list of a vertex v, atleast two memory accesses are required, one to access $vertexArray[v]$ to get the starting index of the adjacency list and the other to access the adjacency lists in *edgeArray*. The access to *edgeArray* has high possibility of a cache miss as the *edgeArray* for large graphs is much larger than the cache.

In the *window-iterator* algorithm, the outer for-loop(line 6) corresponds to s-vertices and inner for-loop(line 8) corresponds to e-vertices. For each s-vertex the outer for-loop needs to be executed at-least once to verify if there is an e-vertex in the window. This requires access to the edgeArray. In most cases, only a small percentage of s-vertices have e-vertices in the window. This is especially true for the iterations at the end. To avoid the access to edgeArray, we use a structure $< minVertex, minVertexIdx >$ for each vertex in the graph. For a vertex v, the $minVertex$ stores the minimum vertex in its adjacency list that has not yet been processed(note that the adjacency lists are sorted). $minVertexIdx$ stores the index of the $minVertex$ in the *edgeArray*. Before executing the inner for-loop, we first check if *minVertex* belongs to the window. If it does not belong, then the access to *edgeArray* is not required. If it belongs then the *edgeArray* is accessed using the *minVertexIdx* and after the inner loop is executed, $< minVertex, minVertexIdx >$ is updated. Note that the structure is stored in an array and accessed sequentially. The use of the structure eliminates the unnecessary accesses to *edgeArray*.

There is also one minor optimization that needs to be mentioned as it is generally applicable to other graph problems which require performing intersection of adjacency lists. To perform intersection operation on adjacency lists of two vertices u and v, it is a general practice to use the degree information of the vertices which is most often stored in a separate structure. Procedure *intersectNeighbors1* in Figure 44 shows an example code that uses degree information. Accessing the degree information results in increased *task-set* size and also in most cases requires random access to the degree array. We propose an optimization which is shown in procedure

```
1: procedure intersectNeighbors1(u, v)
2:     i = 0 j = 0
3:     while i < degree[u] and j < degree[v] do
4:         ▷ uList and vList are adjacency lists of u and v respectively
5:         if uList[i] = vList[j] then
6:             ▷ some code
7:         end if
8:         ▷ some code
9:     end while
10: end procedure
11: procedure intersectNeighbors2(u, v)
12:     i = 0 j = 0
13:     ▷ uList and vList are adjacency lists of u and v respectively
14:     while uList[i] ≠ λ and vList[j] ≠ λ do
15:         if uList[i] = vList[j] then
16:             ▷ some code
17:         end if
18:         ▷ some code
19:     end while
20: end procedure
```

Figure 44: A simple optimization

intersectNeighbors2. We mark the end of an adjacency list using a special value λ. This eliminates the need to access degree information.

4.5 EXPERIMENTAL RESULTS

All the results presented in this section are obtained using a four socket 2.27GHz Xeon X7560(Nehalem-EX) with 256 GB shared memory and running 64-bit Ubuntu 12.04. Each socket consists of 8 cores(can run 16 threads with hyper-threading). Each core has a private 32 KB L1 cache and 256 KB L2 cache. A 24 MB L3 cache is shared by all the cores in a socket. All the implementation is done using C++ programming language and compiled using g++ compiler with -O3 optimization flag.

We have used graphs from the Stanford large network collection [40] and a synthetic graph generated using GTGraph tool [41]. The description of all the graphs is given in Section 2.5 of Chapter 2. The timing results comparing the *edge-iterator-deg* algorithm and *window-iterator* algorithm are given in Table 6. Note that the time for pre-processing is not included in the timing results for both the algorithm.

Table 6: Comparison of *edge-iterator-deg* and *window-iterator* algorithms. n, m and T refer to number of vertices, edges and triangles(all in millions) and time(in seconds)

graph	n	m	T	$edge-iterator$ (in seconds)	$window-iterator$ (in seconds)
cit-Patents	3.8	16.5	7.5	1.87	1.73
soc-Pokec	1.6	30.6	32.5	6.33	5.13
soc-LiveJournal1	4.8	69.0	285.7	13.27	10.77
com-Orkut	3.1	117.2	627.6	89.30	68.9
rmat-32-512	32.0	512.0	2.6	326.27	247.33
com-Friendster	65.6	1806.0	4173.7	2645.20	1851.50

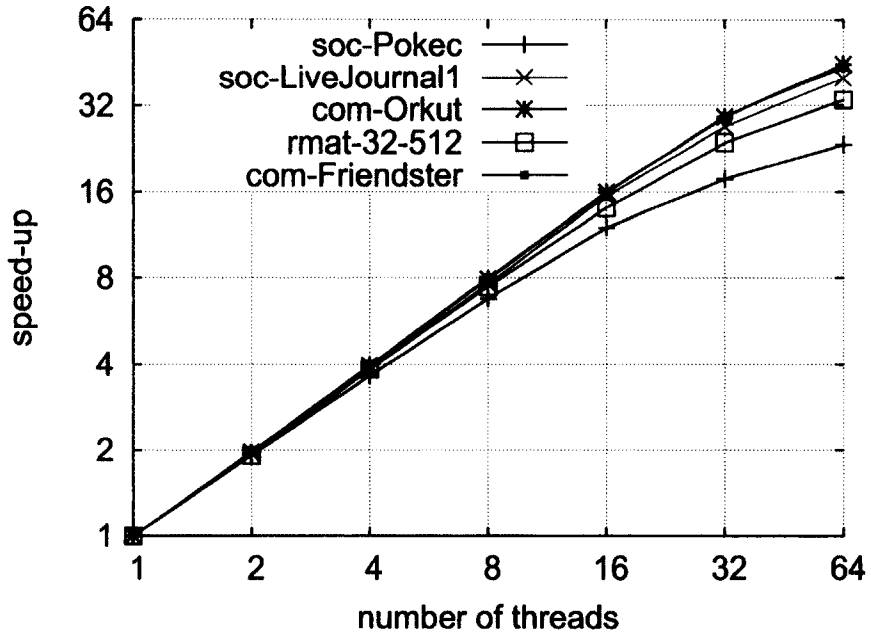


Figure 45: Scalability of *window-iterator* algorithm

Also, for experimental purpose we only output the number of triangles, we don't output all the triangles. It can be clearly seen that the *window-iterator* outperforms *edge-iterator-deg* for all the graph and the gap widens as the graph size increases. We have also experimented using smaller graphs. In case of smaller graphs both the algorithms resulted in similar performance.

The parallel implementation is done using OpenMP. *window-iterator* is amicable to parallelization and it is straightforward to parallelize the algorithm. The workload of each iteration is distributed to all the available threads. After each iteration, before entering the next iteration all the threads are synchronized using a barrier. One major advantage of *window-iterator* algorithm is that, all the threads share the same *task-set*, i.e. the chunk of *edgeArray* corresponding to the window. There are only read accesses to the *task-set* eliminating the overhead of the cache coherency protocol. Also, the shared cache memory is efficiently used as all the threads in the core access the same window. The scalability results of the *window-iterator* algorithm can be seen in the Figure 4.4.2. The algorithm scales more than 29 times using 32 threads and more than 44 times using 64 threads(note that there are only 32 physical cores in the machine).

4.6 SUMMARY

In this chapter, we define the triangle listing problem and present some applications. Different existing triangle listing algorithms are briefly described. A variant of *edge-iterator* algorithm that uses degree ordering of vertices is discussed in detail and its memory locality is analysed. We propose a new algorithm, called *window-iterator*, that combines the *access transformation* and *task-set reduction* techniques discussed in previous chapters to improve the memory locality. To reduce the *task-set* size, it uses an idea similar to the blocking technique [17] used for regular applications. The *task-set* is limited to a window which consists of adjacency lists of a smaller set of vertices. The size of the window is chosen such that the window fits in one of the caches(L3 cache in our experiments). By limiting the *task-set* size, the impact of random memory access pattern is greatly reduced. The *window-iterator* algorithm is compared with a variant of *edge-iterator* algorithm and is shown to outperform for large graphs. *window-iterator* algorithm is amicable to parallelization. we have implemented the algorithm for multicore architecture. We present the scalability results showing that the algorithm scales well, more than 29 times using 32 threads.

CHAPTER 5

CONCLUSION

One of the major differences between the regular applications like linear algebra applications and graphs applications is the memory access pattern. The current commodity processors are dominated by multicore systems. Multicore processors have multiple processing units called cores. Each core has a private L1 cache(and possible more cache levels). Each core also has access to a shared cache and also main memory. Caches are typically orders or magnitude faster and smaller than the main memory. When the access pattern is sequential the cache memory is better utilized reducing the need to access the main memory. The hardware prefetcher can analyze the access pattern and make better predictions to bring the data into the cache before it is accessed. In the case of graph algorithms, the memory access pattern is highly random. It is generally not possible for the hardware prefetcher to predict what data is going to be accessed. When the graph size is small such that most data can fit in the cache, the access pattern does not pose a major problem. However, for the current large sizes of the graphs, the impact of the access pattern can be substantial. Since most data accesses result in cache misses requiring access to the main memory which has high latency, the performance of graphs algorithms is dominated by the memory access time. Locality also impacts the degree of parallelism due to the effect of cache coherency protocols. Therefore improving the locality in graph applications is crucial to the performance. The main motivation of the thesis is to show the importance of locality in graph algorithms and present techniques to improve locality that can result in significant performance benefits.

Improving locality in graph algorithms is highly challenging. Achieving good locality requires careful analysis of the data structures and the access pattern. An algorithm can be implemented in different ways using different data structures. The data structures chosen play a crucial role in the performance. For example, a graph can be represented using an adjacency matrix, adjacency list, adjacency array or in compressed row format(CSR) and the choice of representation influences the performance.

In this thesis, we present two techniques to improve locality. The first technique, *access transformation* focuses on the access pattern while the second technique focuses on the size of the data. The *access transformation* technique is applicable to iterative approaches in which a subset of vertices/edges are processed in each iteration. The idea of this technique is to scan all the vertices to extract the order in which they are processed. This technique adds $O(kn)$ to the complexity of the algorithm where k is the number of iterations and n is the number of vertices. However, since the access pattern of scan operation in sequential, it is not much overhead when the number of iterations is limited. Moreover, the scan operation is embarrassingly parallel. We apply the technique to the k-core decomposition and triangle listing problems.

The *task-set reduction* technique is a more general technique. It refers to reducing the size of the data that a task repeatedly accesses. The reduction can be achieved using different methods. Once method, that we used in the maximal clique enumeration problem, is called compression. In compression, the *task-set* is not modified but the memory required to store the *task-set* is reduced by using a different format like bit representation. Another method, that is used in the triangle listing problem, is based on the blocking technique. In this method, the task is divided into sub-tasks each sub-task working on a smaller *task-set*. Another method, used in k-core decomposition problem, is called elimination. This requires eliminating the use of some data structures by modifying the tasks such that the data in those data structures is generated from other sources when required.

The two techniques, *access transformation* and *task-set reduction* have been applied to three graphs problems, k-core decomposition, maximal clique enumeration and triangle listing. The applicability of these techniques requires a thorough understanding of the algorithms, the data structures used for implementing and analysis of the access pattern. Intuitively, the algorithms that have scope for temporal locality like the enumeration of vertex covers, enumeration of spanning trees and enumeration of matchings are candidates for *task-set reduction* technique. And the graph problems which access the vertices/edges only once(or constant number of times) but in random order like the minimum spanning trees, approximate vertex cover, single source shortest path are candidates for *access transformation* technique. The memory locality of other graph algorithms needs to be analyzed and improved using the proposed techniques.

REFERENCES

[1] A Comprehensive List of Big Data Statistics [Online]. Available: `http://wikibon.org/blog/big-data-statistics/`

[2] How Much Data is Created Every Minute? [Online]. Available: `http://www.domo.com/blog/2012/06/how-much-data-is-created-every-minute/`

[3] Big Data: What is it and why it matters [Online]. Available: `http://www.sas.com/big-data/`

[4] M. J. Keeling and K. T. D. Eames. Networks and epidemic models. *J. R. Soc. Interface*, 2:295–307, 2005.

[5] O. Mason and M. H. A. Verwoerd. Graph theory and networks in Biology. *Systems Biology, IET*, 1(2):89–119, 2007.

[6] E. Mohyedinbonab, M. Jamshidi, and Y. F. Jin. A review on applications of graph theory in network analysis of biological processes. *International Journal of Intelligent Computing in Medical Sciences & Image Processing*, 6(1):27–43, 2014.

[7] D. Kempe, J. Kleinberg, and E. Tardos. Maximizing the spread of influence through a social network. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '03, pages 137–146, 2003.

[8] M. Cha, A. Mislove, and K. P. Gummadi. A measurement-driven analysis of information propagation in the flickr social network. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, pages 721–730, 2009.

[9] W.-S. Yang, J.-B. Dia, H.-C. Cheng, and H.-T. Lin. Mining social networks for targeted advertising. In *HICSS*. IEEE Computer Society, 2006.

[10] J. He. *A Social Network-based Recommender System.* PhD thesis, 2010.

[11] S. Fortunato. Community detection in graphs. *Physics Reports*, 486(3-5):75 – 174, 2010.

[12] V. Krebs. Uncloaking Terrorist Networks. *First Monday*, 7(4), Apr. 2002.

[13] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[14] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, 2010.

[15] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, 2010.

[16] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, July 2010.

[17] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 30–44, 1991.

[18] V. Batagelj and M. Zaversnik. An O(m) Algorithm for Cores Decomposition of Networks. *CoRR*, arXiv.org/cs.DS/0310049, 2003.

[19] S. Hong, T. Oguntebi, and K. Olukotun. Efficient parallel graph exploration on multi-core cpu and gpu. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 78–88, 2011.

[20] S. B. Seidman. Network structure and minimum degree. *Social Networks*, 5(3):269 – 287, 1983.

[21] S. Wasserman and K. Faust. *Social network analysis: Methods and applications*, volume 8. Cambridge university press, 1994.

[22] J. I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani. K-core decomposition of internet graphs: hierarchies, self-similarity and measurement biases. *NHM*, 3(2):371–393, 2008.

[23] S. Carmi, S. Havlin, S. Kirkpatrick, and E. Shir. Medusa - new model of internet topology using k-shell decomposition. *PNAS*, pages 11–150, 2007.

[24] L. Gallos, S. Havlin, M. Kitsak, F. Liljeros, H. Makse, L. Muchnik, and H. Stanley. Identification of influential spreaders in complex networks. *Nature Physics*, 6(11):888–893, Aug. 2010.

[25] M. Pellegrini, F. Geraci, and M. Baglioni. Detecting dense communities in large social and information networks with the core & peel algorithm. *CoRR*, abs/1210.3266, 2012.

[26] S. Papadopoulos, Y. Kompatsiaris, A. Vakali, and P. Spyridonos. Community detection in social media. *Data Mining and Knowledge Discovery*, 24(3):515–554, 2012.

[27] V. Batagelj, A. Mrvar, and M. Zaversnik. Partitioning approach to visualization of large graphs. In *GD '99: Proceedings of the 7th International Symposium on Graph Drawing*, pages 90–97. Springer-Verlag, 1999.

[28] J. I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani. k-core decomposition: a tool for the visualization of large scale networks. *CoRR*, abs/cs/0504107, 2005.

[29] J. I. Alvarez-hamelin, A. Barrat, and A. Vespignani. Large scale networks fingerprinting and visualization using the k-core decomposition. In *Advances in Neural Information Processing Systems 18*, pages 41–50. MIT Press, 2006.

[30] G. D. Bader and C. W. Hogue. Analyzing yeast protein-protein interaction data obtained from different sources. *Nature biotechnology*, 20(10):991–997, 2002.

[31] Y. Cheng, C. Lu, and N. Wang. Local k-core clustering for gene networks. In *IEEE International Conference on Bioinformatics and Biomedicine*, pages 9–15, 2013.

[32] D. Eppstein, M. Löffler, and D. Strash. Listing all maximal cliques in sparse graphs in near-optimal time. In *ISAAC*, volume 6506 of *Lecture Notes in Computer Science*, pages 403–414. 2010.

[33] R. A. Rossi, D. F. Gleich, A. H. Gebremedhin, and M. M. A. Patwary. A fast parallel maximum clique algorithm for large sparse graphs and temporal strong components. *CoRR*, abs/1302.6256, 2013.

[34] G. Csardi and T. Nepusz. The igraph software package for complex network research. *InterJournal*, Complex Systems:1695, 2006.

[35] C. Staudt, A. Sazonovs, and H. Meyerhenke. Networkit: An interactive tool suite for high-performance network analysis. *CoRR*, abs/1403.3005, 2014.

[36] A. Montresor, F. D. Pellegrini, and D. Miorandi. Distributed k-core decomposition. *IEEE Transactions on Parallel and Distributed Systems*, 24(2):288–300, 2013.

[37] A. E. Saríyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and U. V. Çatalyürek. Streaming algorithms for k-core decomposition. *Proc. VLDB Endow.*, 6(6):433–444, Apr. 2013.

[38] R. Li, J. X. Yu, and R. Mao. Efficient core maintenance in large dynamic graphs. *IEEE Trans. Knowl. Data Eng.*, 26(10):2453–2465, 2014.

[39] D. Miorandi and F. D. Pellegrini. K-shell decomposition for dynamic complex networks. In *WiOpt*, pages 488–496, 2010.

[40] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. [Online]. Available: `http://snap.stanford.edu/data/`

[41] D. A. Bader and K. Madduri. Gtgraph: A synthetic graph generator suite, 2006.

[42] J. Seward, N. Nethercote, and J. Weidendorfer. *Valgrind 3.3 - Advanced Debugging and Profiling for GNU/Linux Applications.* Network Theory Ltd., 2008.

[43] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader. Scalable graph exploration on multicore processors. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, 2010

[44] Harshvardhan, A. Fidel, N. M. Amato, and L. Rauchwerger. Kla: A new algorithmic paradigm for parallel graph computations. In *Proceedings of the 23rd*

International Conference on Parallel Architectures and Compilation, PACT '14, pages 27–38, 2014.

[45] D. Eppstein and D. Strash. Listing all maximal cliques in large sparse real-world graphs. In *Proceedings of the 10th international conference on Experimental algorithms*, SEA'11, pages 364–375, 2011.

[46] N. Du, B. Wu, X. Pei, B. Wang, and L. Xu. Community detection in large-scale social networks. In *WebKDD/SNA-KDD '07: Proceedings of the 9th WebKDD and 1st SNA-KDD 2007 workshop on Web mining and social network analysis*, pages 16–25, 2007.

[47] Z. Chen, K. A. Wilson, Y. Jin, W. Hendrix, and N. F. Samatova. Detecting and tracking community dynamics in evolutionary networks. In *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*, ICDMW '10, pages 318–327, 2010.

[48] R. Rowe, G. Creamer, S. Hershkop, and S. J. Stolfo. Automated social hierarchy detection through email network analysis. In *Proceedings of the 9th WebKDD and 1st SNA-KDD 2007 Workshop on Web Mining and Social Network Analysis*, WebKDD/SNA-KDD '07, pages 109–117, 2007.

[49] R. Horaud and T. Skordas. Stereo correspondence through feature grouping and maximal cliques. *IEEE Trans. Pattern Anal. Mach. Intell.*, 11(11):1168–1180, Nov. 1989.

[50] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In *3rd Intl. Conf. on Knowledge Discovery and Data Mining*, pages 283–286, 1997.

[51] B. Zhang, B.-H. Park, T. V. Karpinets, and N. F. Samatova. From pull-down data to protein interaction networks and complexes with biological relevance. *Bioinformatics*, 24(7):979–986, 2008.

[52] K. L. Jensen, M. P. Styczynski, I. Rigoutsos, and G. Stephanopoulos. A generic motif discovery algorithm for sequential data. *Bioinformatics*, 22(1):21–28, 2006.

[53] H. M. Grindley, P. J. Artymiuk, D. W. Rice, and P. Willett. Identification of tertiary structure resemblance in proteins using a maximal common subgraph isomorphism algorithm. *J Mol Biol*, 229(3):707–721, Feb. 1993.

[54] Y. Chen and G. M. Crippen. A novel approach to structural alignment using realistic structural and environmental information. *Protein Science*, 14(12):2935–2946, 2005.

[55] I. Koch, T. Lengauer, and E. Wanke. An algorithm for finding maximal common subtopologies in a set of protein structures. *Journal of Computational Biology*, 3(2):289–306, 1996.

[56] R. Samudrala, J. Moult, and C. Biology. A graph-theoretic algorithm for comparative modeling of protein structure. *J Mol Biol*, 279(1):287–302, 1998.

[57] E. J. Gardiner, P. Willett, and P. J. Artymiuk. Graph-theoretic techniques for macromolecular docking. *Journal of Chemical Information and Computer Sciences*, 40(2):273–279, 2000.

[58] J. Moon and L. Moser. On cliques in graphs. *Israel Journal of Mathematics*, 3(1):23–28, 1965.

[59] F. Harary and I. C. Ross. A procedure for clique detection using the group matrix. *Sociometry*, 20:205–215, 1957.

[60] C. Bron and J. Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Commun. ACM*, 16(9):575–577, Sept. 1973.

[61] E. Tomita, A. Tanaka, and H. Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theor. Comput. Sci.*, 363(1):28–42, Oct. 2006.

[62] P. M. Pardalos and J. Xue. The maximum clique problem. *Journal of Global Optimization*, 4(3):301–328, April 1994.

[63] R. E. Bonner. On some clustering techniques. *IBM J. Res. Dev.*, 8(1):22–32, Jan 1964.

[64] J. G. Augustson and J. Minker. An analysis of some graph theoretical cluster techniques. *J. ACM*, 17(4):571–588, Oct. 1970.

[65] E. A. Akkoyunlu. The enumeration of maximal cliques of large graphs. *SIAM J. Comput.*, 2(1):1–6, 1973.

[66] H. C. Johnston. Cliques of a graph-variations on the Bron-Kerbosch algorithm. *International Journal of Parallel Programming*, 5(3):209–238, Sept. 1976.

[67] I. Koch. Enumerating all connected maximal common subgraphs in two graphs. *Theor. Comput. Sci.*, 250(1-2):1–30, Jan. 2001.

[68] F. Cazals and C. Karande. Note: A note on the problem of reporting maximal cliques. *Theor. Comput. Sci.*, 407(1-3):564–568, Nov. 2008.

[69] S. Tsukiyama, M. Ide, H. Ariyoshi, and I. Shirakawa. A new algorithm for generating all the maximal independent sets. *SIAM J. Comput.*, 6(3):505–517, 1977.

[70] E. L. Lawler, J. K. Lenstra, and A. H. G. R. Kan. Generating all maximal independent sets: NP-hardness and polynomial-time algorithms. *SIAM J. Comput.*, 9(3):558–565, 1980.

[71] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM J. Comput.*, 14(1):210–223, Feb. 1985.

[72] K. Makino and T. Uno. New algorithms for enumerating all maximal cliques. In *SWAT*, pages 260–272, 2004.

[73] Y. Zhang, F. N. Abu-Khzam, N. E. Baldwin, E. J. Chesler, M. A. Langston, and N. F. Samatova. Genome-scale computational approaches to memory-intensive applications in systems biology. In *SC*, page 12, 2005.

[74] F. Kose, W. a. Weckwerth, T. Linke, and O. Fiehn. Visualizing plant metabolomic correlation networks using clique-metabolite matrices. *Bioinformatics*, 17(12):1198–1208, 2001.

[75] N. Du, B. Wu, L. Xu, B. Wang, and P. Xin. Parallel algorithm for enumerating maximal cliques in complex network. In *Mining Complex Data*, pages 207–221. 2009.

[76] M. C. Schmidt, N. F. Samatova, K. Thomas, and B.-H. Park. A scalable, parallel algorithm for maximal clique enumeration. *Journal of Parallel and Distributed Computing*, 69(4):417 – 428, 2009.

[77] L. Lu, Y. Gu, and R. L. Grossman. dmaximalcliques: A distributed algorithm for enumerating all maximal cliques and maximal clique distribution. In *ICDM Workshops*, pages 1320–1327, 2010.

[78] T. A. Davis and Y. Hu. The university of Florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, Dec. 2011.

[79] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, editors. *Graph Partitioning and Graph Clustering - 10th DIMACS Implementation Challenge Workshop, Georgia Institute of Technology, Atlanta, GA, USA, February 13-14, 2012. Proceedings*, volume 588 of *Contemporary Mathematics*. American Mathematical Society, 2013.

[80] D. J. Johnson and M. A. Trick, editors. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, Workshop, October 11-13, 1993.* American Mathematical Society, Boston, MA, USA, 1996.

[81] D. Strash. Quick cliques: A package to efficiently compute all maximal cliques in sparse graphs. [Online]. Available: `http://www.dcs.gla.ac.uk/~pat/jchoco/clique/enumeration/quick-cliques/doc/index.html`

[82] B. Wu, S. Yang, H. Zhao, and B. Wang. A distributed algorithm to enumerate all maximal cliques in mapreduce. In *Proceedings of the 2009 Fourth International Conference on Frontier of Computer Science and Technology*, FCST '09, pages 45–51, 2009.

[83] B. Elser and A. Montresor. An evaluation study of bigdata frameworks for graph processing. In *Big Data, 2013 IEEE International Conference on*, pages 60–67, Oct 2013.

[84] R. Durstenfeld. Algorithm 235: Random permutation. *Commun. ACM*, 7(7):420, July 1964.

[85] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):409–10, 1998.

[86] Harary, Frank and Paper, Herbert H. Toward a general calculus of phonemic distribution. *Language*, 33(2):143–169, Apr. 1957.

[87] F. Harary and H. J. Kommel. Matrix measures for transitivity and balance. *The Journal of Mathematical Sociology*, 6(2):199–210, 1979.

[88] Z. Bar-Yossef, R. Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *SODA*, pages 623–632, 2002.

[89] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: simple building blocks of complex networks. *Science*, 298(5594):824–827, October 2002.

[90] E. Yeger-Lotem, S. Sattath, N. Kashtan, S. Itzkovitz, R. Milo, R. Y. Pinter, U. Alon, and H. Margalit. Network motifs in integrated cellular networks of transcription-regulation and protein-protein interaction. *Proceedings of the National academy of Sciences of the United States of America*, 101(16):5934–5939, 2004.

[91] Z. Yang, C. Wilson, X. Wang, T. Gao, B. Y. Zhao, and Y. Dai. Uncovering social network sybils in the wild. In *Internet Measurement Conference*, pages 259–268, 2011.

[92] M. Ortmann and U. Brandes. Triangle listing algorithms: Back from the diversion. In *2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments, ALENEX*, pages 1–8, 2014.

[93] T. Schank and D. Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *Proceedings of the 4th International Conference on Experimental and Efficient Algorithms*, WEA'05, pages 606–609, 2005.

[94] M. Latapy. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical Computer Science*, 407(13):458 - 473, 2008.

[95] S. Chu and J. Cheng. Triangle listing in massive networks and its applications. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, pages 672–680, 2011.

[96] A. Itai and M. Rodeh. Finding a minimum circuit in a graph. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, STOC '77, pages 1–10, 1977.

[97] A. Itai and M. Rodeh. Finding a minimum circuit in a graph. *SIAM Journal on Computing*, 7(4):413–423, 1978.

[98] V. Batagelj and A. Mrvar. A subquadratic triad census algorithm for large sparse networks with small maximum degree. *Social Networks*, 23(3):237 - 243, 2001.

[99] H.-M. Park, F. Silvestri, U. Kang, and R. Pagh. Mapreduce triangle enumeration with guarantees. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, CIKM '14, pages 1739–1748, 2014.

[100] H.-M. Park and C.-W. Chung. An efficient mapreduce algorithm for counting triangles in a very large graph. In *Proceedings of the 22nd ACM International Conference on Conference on Information and Knowledge Management*, CIKM '13, pages 539–548, 2013.

[101] F. N. Afrati, A. D. Sarma, S. Salihoglu, and J. D. Ullman. Upper and lower bounds on the cost of a map-reduce computation. *CoRR*, abs/1206.4377, 2012.

[102] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, pages 607–614, 2011.

[103] M. Sevenich, S. Hong, A. Welc, and H. Chafi. Fast in-memory triangle listing for large real-world graphs. In *Proceedings of the 8th Workshop on Social Network Mining and Analysis*, SNAKDD'14, pages 2:1–2:9, 2014.

[104] J. Shun and K. Tangwongsan. Multicore triangle computations without tuning. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 149–160, 2015.

VITA

Naga Shailaja Dasari
Department of Computer Science
Old Dominion University
Norfolk, VA 23529

Shailaja received her undergraduate degree in 2003 from Kakatiya University, India and Masters degree in 2006 from Indian Institute of Technology Kanpur, India. During the masters program she worked as a teaching assistant and did her thesis in Biometrics as part of the program.

After obtaining M.Tech degree she joined the Virtualization team at Microsoft India Development Center in 2006 as a software design engineer. While working at Microsoft she contributed to the products Virtual PC 2007, Virtual Server 2005 R2 SP1, Hyper-v and Windows Virtual PC.

Shailaja joined the PhD program at Old Dominion University in Fall 2009. Her research interests include Bioinformatics, algorithms, data structures and high performance computing. In 2011, she received the Outstanding Computer Science Research Assistant Award given by the Department of Computer Science.

Typeset using LaTeX.