MULTICORE PARALLELIZATION OF AN INDEXER IN QUESTION ANSWERING SYSTEM AND PAGE RANK ALGORITHM

A DISSERTATION

Submitted in partial fulfillment of the requirements for the award of the degree

of

MASTER OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

By

TARUN KUMAR

DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY ROORKEE
ROORKEE -247 667 (INDIA)
JUNE, 2009

[illegible]

Candidate's Declaration

I hereby declare that the work being presented in the dissertation report titled "**Multicore Parallelization of an Indexer in Question Answering System and PageRank Algorithm**" in partial fulfillment of the requirement for the award of the degree of Master of Technology in Computer Science and Engineering, submitted in the Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee, is an authentic record of my own work carried out under the guidance of Dr. Ankush Mittal, Associate Professor in Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee. I have not submitted the matter embodied in this dissertation report for the award of any other degree.

Dated: 10-06-09

Place: IIT Roorkee

Tarun Kumar

(Tarun Kumar)

Certificate

This is to certify that above statements made by the candidate are correct to the best of my knowledge and belief.

Dated: 10-06-09

Place: IIT Roorkee.

[illegible]

Dr. Ankush Mittal,
Associate Professor,
Department of Electronics
and Computer Engineering,
IIT Roorkee, Roorkee,
247667 (India).

ACKNOWLEDGEMENTS

I am thankful to Indian Institute of Technology Roorkee for giving me this opportunity. It is my privilege to express thanks and my profound gratitude to my supervisor Dr. Ankush Mittal, Associate Professor for his invaluable guidance and constant encouragement throughout the dissertation. I was able to complete this dissertation in time due to the constant motivation and support received from him.

I am also grateful to Mr. Parikshit Sondhi, Ph. D Scholar in Department of Computer Science, University of Illinois at Urbana Champaign for helping me understand some basic and important concepts explored in the dissertation work. His valuable help and constant support proved immensely beneficial for my work so did his ability to motivate me. I am grateful to Mr. Salil Shirish Sahasrabudhe, Mr. Khalil Sawant, Mr. Kshitiz Gupta, Mr. Nityam Parakh and Mr. Payas Goyal, my colleagues, for being excellent peers and creating a congenial environment for work. I am also thankful to all my friends who helped me directly and indirectly in completing this dissertation.

Most importantly, I would like to extend my deepest appreciation to my family for their love, encouragement and moral support.

(Tarun Kumar)

ABSTRACT

Explosive growth of information over internet and increasing number of users of WWW are throwing major challenges to the web applications. In order to deal with this growth, web applications are utilizing increased processing hardware. The need of hardware is currently served by connecting thousands of computers in cluster. But faster and less complex alternatives to this system can be found as a multi-core processor. A recent breakthrough with introduction of the STI Cell Processor and GPU multiprocessors has provided a new alternative for the researchers to port computationally intensive applications on them.

A question answering system is an information retrieval application which allows users to directly obtain appropriate answers to a question. Over the time, in order to provide more accurate and relevant answer, processing stages in question answering systems have increased many times. Tasks like indexing a huge document set and retrieving answer to the user query are highly computational intensive and consume significant processing time. As a part of this dissertation we identify major issues involved in porting a general question answering framework on Cell processor and their possible solutions. The work is evaluated by porting the indexing algorithm of a biomedical question answering system, INDOC (Internet Doctor) on Cell processor.

In order to provide most relevant results to a search query, search engine Google implemented a ranking technique (called PageRank algorithm) for assigning ranks to all web pages. Page rank of a particular web page is determined by page rank of all those web pages which are pointing to this web page. Besides this, PageRank algorithm works upon a large number of web pages. Thus the PageRank calculation is computational intensive. In this dissertation we identify major issues involved in porting PageRank algorithm on Cell BE Processor and CUDA, and their possible solutions. The work is evaluated by taking three input graphs of different size ranging from 0.35 million nodes to 1.3 million and comparing results with previous implementation of PageRank on Cell BE.

Table of Contents

List of Figures

List of Tables

CHAPTER 1

INTRODUCTION

1.1 Question Answering System

Question Answering (QA) systems represent the next step in information retrieval applications as they allow users to directly obtain answers to questions rather than following the search engine style approach of returning a list of documents for queries. QA system has to deal with wide variety of question like fact, list, definition, how, why etc. There are two types of QA systems: Closed domain QA systems which deal with questions under a specific domain such as medicines and Open domain QA systems which deal with questions about everything [1]. A general QA system framework usually involves steps such as document preprocessing, parsing, indexing, question classification, question keyword weighing, document ranking and answer extraction. Thus, there is often a deeper level of document and question processing involved both in the indexing and retrieval stages. While such an extended pipeline of NLP operations greatly helps in improving accuracy, it also greatly increases the time and processing power required for indexing and retrieval operations. This makes it infeasible to run sophisticated information extraction systems over very large corpora where these operations are really required.

A rapid increase in size of document set and information has led to a huge growth in WWW in recent years. If we talk about a particular domain, say biomedical literature where there are currently an estimated 17 million citations in PUBMED [2], the current breed of search engines have been proven to be grossly inadequate [3] as they lack the knowledge of biomedical terminology [4]. As a solution to these problems, [5] suggests a biomedical question answering system– INDOC , which is designed and developed at IIT Roorkee. It is based on the novel idea of indexing, document ranking and extracting the answer to the question posed. The system achieves an accuracy of 76% over first five documents and increases up to 83% for 50 documents retrieved. However, the drawback of this algorithm is that it is slow to be used because of the large document set to be processed. Beside this, the number of searches on biomedical domains has been increased rapidly to nearly 120 million searches yearly on PUBMED database alone [6]. In order to reduce the response time to these many queries, QA system should be fast at query processing and answer extraction. Besides this, as the document set increases rapidly in size, indexing of these document set need to be done more frequently in order to produce more accurate results. In this dissertation, we focus on a faster implementation of indexing module of INDOC.

1.2 PageRank Algorithm

The most used search engine, Google produces high precision results. The main reason of its better results is the use of link structure of web to calculate a quality ranking for each web page. This ranking technique is called PageRank [7]. PageRank assigns a relative importance called rank of the page, to each web page. Rank of a particular web page depends upon the rank of the web pages which are linked to this page. Higher the page rank more important is the page. PageRank approach, introduced in [8] has been the most successful ranking technique for determining the relative importance of web pages. PageRank algorithm itself is computational intensive and it has to work upon billions of web pages. It takes time in order of days to solve the PageRank algorithm [9]. Web pages are updated, added, removed to and from WWW continually, therefore the frequent computation of rank of pages is required. Besides this, some applications of PageRank like topic sensitive search and personalized web search require large number of page rank scores recomputed to reflect the user preferences [10]. Thus, some new ways to calculate rank of web pages in minimum possible time are always sought. Different possibilities and ways have been devised to reduce the time to solve PageRank algorithm like reducing I/O time of disk, improving convergence rate of algorithm, and calculating PageRank of web pages in parallel on a cluster of computers. An alternative approach can be thought in the form of multicore processors like Cell Processor and GPU. We have explored this approach in our dissertation.

1.3 Multicore Processor

A multicore processor is an integrated circuit to which two or more processors have been attached for enhanced performance, reduced power consumption, and more efficient simultaneous processing of multiple tasks [11]. Multicore processors can be used for the type of problems which are computationally intensive. They can provide a significant amount of performance gain as compared to the uniprocessor. Different multicore processors differ to each other in terms of memory organization, communication between different cores, processing units, type of parallelization etc. The Cell Broadband Engine (Cell BE) processor is a multicore processor. It is designed with the computationally intensive applications in mind and is often used to achieve real time processing and reduce the execution time considerably for various applications. GPU (Graphics Processing Unit) is also a multicore processor. It is a highly parallel, multi-threaded processor with tremendous computing power. GPUs were designed to work on images and graphics oriented applications. CUDA (Compute Unified Device Architecture) provides a programming environment which facilitates programmer to design general purpose applications on GPU. We will explore these two multicore architectures in the present work.

1.4 Problem Statement

In this dissertation work we

1. Identify major issues involved in porting a general Question Answering framework on a Cell BE processor and propose potential solutions to these issues. The evaluation of these solutions is done by porting indexing algorithm of a biomedical QA system, INDOC on Cell BE processor.
2. Identify the issues and their possible solution of porting PageRank algorithm on Cell BE Processor followed by the implementation of PageRank algorithm on Cell BE. We also provide an implementation of PageRank algorithm on CUDA.

1.5 Organization of Report

Chapter 2 discusses the hardware architecture of the multicore processor STI Cell BE. It also provides a brief introduction about GPU and CUDA.

Chapter 3 discusses the background details of a Question Answering system and PageRank algorithm. It also points out the issues of implementing a general QA system and PageRank algorithm on Cell BE processor, and then provides solutions to those issues.

Chapter 4 discusses the indexing algorithm of a Biomedical Question Answering system-INDOC and then describes implementation of this indexing algorithm on Cell BE Processor. It also shows the performance of Cell BE processor on indexing algorithm.

Chapter 5 discusses the PageRank algorithm, link structure of web and describes design and implementation of PageRank algorithm on Cell BE processor. It also shows the performance of Cell BE processor on PageRank algorithm.

Chapter 6 provides the design and implementation details of PageRank algorithm on CUDA and compare it with results on Cell BE Processor.

Chapter 7 concludes the dissertation work and gives suggestions for future work.

CHAPTER 2

MULTICORE PROCESSORS

A multicore processor is a processing system composed of two or more independent cores (or CPUs). The cores are typically integrated onto a single integrated circuit die. In this chapter we are going to describe two multicore architectures- STI Cell BE and CUDA.

2.1 STI Cell Broadband Engine

Cell Broadband Engine (Cell BE) is a joint venture of Sony, Toshiba and IBM Corporation formed in 2001. This collaboration of three companies is known as STI. The Cell BE processor is the first implementation of a new family of multiprocessors conforming to the Cell Broadband Engine architecture which extends 64 bit Power PC Architecture.

2.1.1 Cell BE Architecture

The Cell BE [12] is a heterogeneous multicore chip that is significantly different from conventional multiprocessors. Architecture of Cell BE is shown in figure 2.1. It consists of a central microprocessor called the Power processing element (PPE), eight SIMD co-processing units called synergistic processor elements (SPE), a high speed memory controller, and a high bandwidth bus interface, all integrated on a single chip. The Cell BE operates on the fundamentals of increasing concurrency through the use of multiple processing cores and increasing specialization in execution through non-homogeneous parallelization. It employs 8 SPEs onto which threads of an application can be mapped parallely and controlled by PPE. PPE and SPEs communicate through a common internal high-speed Element Interconnect Bus (EIB). The SPE offers a high bandwidth interface to a direct memory access (DMA) that can transfer 32 GB/sec to and from the 256 KB local store memory. The Cell BE has clock speed of 3.2 GHz.

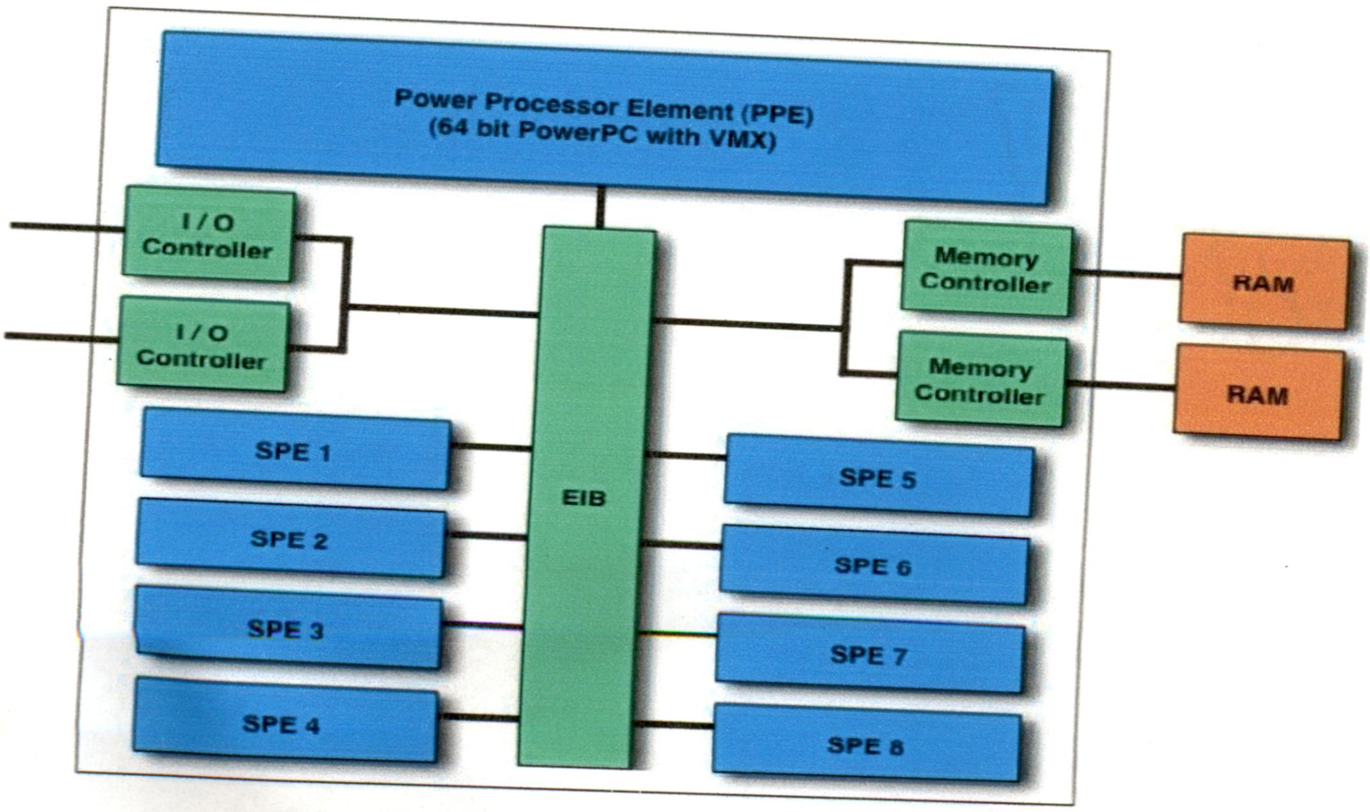

Figure 2.1: Architecture of Cell BE Processor [13]

Power Processor Element (PPE)

PPE is responsible for overall control of Cell BE. It runs an operating system, manages system resources, and is intended primarily for control processing, including the allocation and management of SPE threads. It has 32KB L1 instruction cache, 32KB L1 data caches and 512 KB L2 instruction and data cache. The instruction set for PPE is an extension of the PowerPC instruction set. It also includes a vector multimedia extension unit, called Single Instruction, Multiple Data (SIMD), so that it can do multiple operations simultaneously with a single instruction.

Synergistic Processor Element (SPE)

Eight homogeneous SPEs are Single Instruction Multiple Data (SIMD) processor elements that are optimized for data-rich operations allocated to them by PPE. It consists of two main units, the Synergistic Processor Unit (SPU) and the Memory Flow Controller (MFC). The SPE deals with instruction control and execution. It includes mainly a single register file with 128 registers (each one 128 bits wide), a unified (instructions and data) 256-KB local store (LS), an instruction-control unit, a load and store unit, and DMA interface. The SPE implements a new SIMD instruction set, the SPE Instruction Set Architecture. The MFC contains a DMA controller that supports DMA transfers. Programs running on the SPE use the MFC's DMA transfers to move instructions and data between the SPE's LS and main storage. To support DMA transfers, the MFC maintains and processes queues of DMA commands. After a DMA command has been queued to the MFC, the SPE can continue to execute instructions while the MFC processes the DMA command autonomously and asynchronously.

Element Interconnect Bus (EIB)

The EIB is a communication bus, internal to the Cell BE processor which connects various on-chip system elements: the PPE processor, the memory controller (MIC), eight SPE processors, and two off-chip I/O interfaces, for a total of 12 participants.

2.1.2 Level of parallelism in Cell BE

Cell BE offers several level of parallelism to achieve high performance such as.

- SIMD processing
- Multithreading
- Double Buffering
- Multiple execution units with heterogeneous architectures

2.1.3 Cell BE Programming

Programming on Cell BE involves developing two separate set of codes, one that is executed on PPE and another for SPE. Execution of program starts with PPE code. PPE creates threads and load the SPE code onto SPEs for execution. Different SPEs may run same program or they may have separate programs. PPE manages data, input/output for all SPEs. Main programming components of Cell BE involve SIMD instructions, DMA transfer and mailbox.

SIMD

Both the PPE and SPE support parallel processing of SIMD (Single Instruction Multiple Data). They can execute four single precision floating-point operations in one cycle. Loop unrolling is also preferred on both PPE and SPEs. One SPE has 128 registers of 128 bit which can provide deeper level of loop unrolling and support for SIMD operations. Branches on Cell BE are expensive and each branch mis-prediction results in loss of 18 cycles. This also forces a programmer to use loop unrolling sothat number of branches in loop can be reduced.

DMA

DMA is an operation which is used to transfer data between main memory and local store of SPE. SPEs can use this to directly access the main memory in parallel to its execution. The instruction set for DMA operations provides a lot of flexibility to double buffer the memory operations and program execution. DMA transfer can be initiated either from PPE or SPE. PPE initiated DMA transfer have more latency than the SPE initiated DMA transfer. DMA transfer can take place between main memory and local store or between local-store of one SPE to another.

Mailbox

The communication between any two SPEs or SPE and PPE can be done with the help of mailbox. Mailboxes are considered from the point of view of SPEs. Each SPE has three mailboxes. Two mailboxes are used for sending data from SPE to PPE or other SPE, and third is used for sending data from PPE to SPE.

2.2 GPU and CUDA

GPU (Graphics Processing Unit) is a highly parallel, multi-threaded and many-core processor with tremendous computing power. Over the past few years the capabilities of GPU have increased drastically and performance has been increasing many folds compared to CPUs. Figure 2.2 shows a comparison of floating point operations per second on CPU and GPU [14].

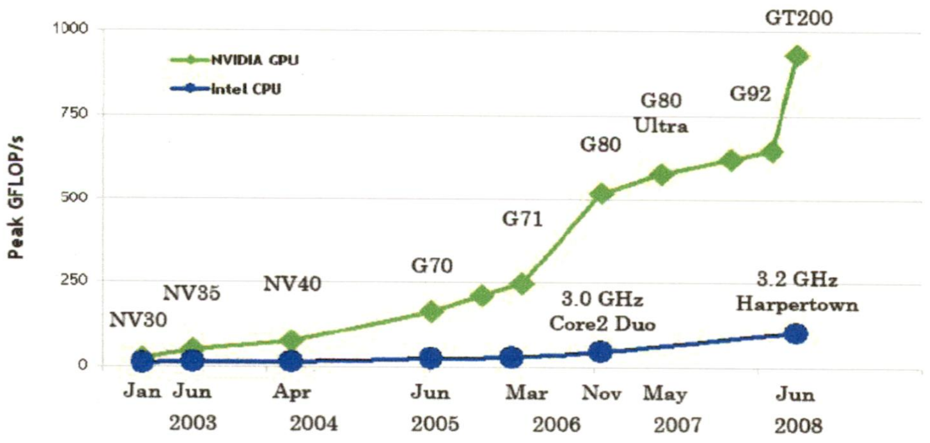

Figure2.2: Comparison of floating point operations per seconds on GPU and CPU [14]

The reason behind the better capability of GPU over CPU is that GPU is specially designed for computation in such a way that more transistors are devoted to computation rather than flow control and caching. Figure 2.3 shows that transistors for ALU (in green color) are more in GPU than CPU. Besides this, GPU is specially designed to address the problems which require data parallel computation.

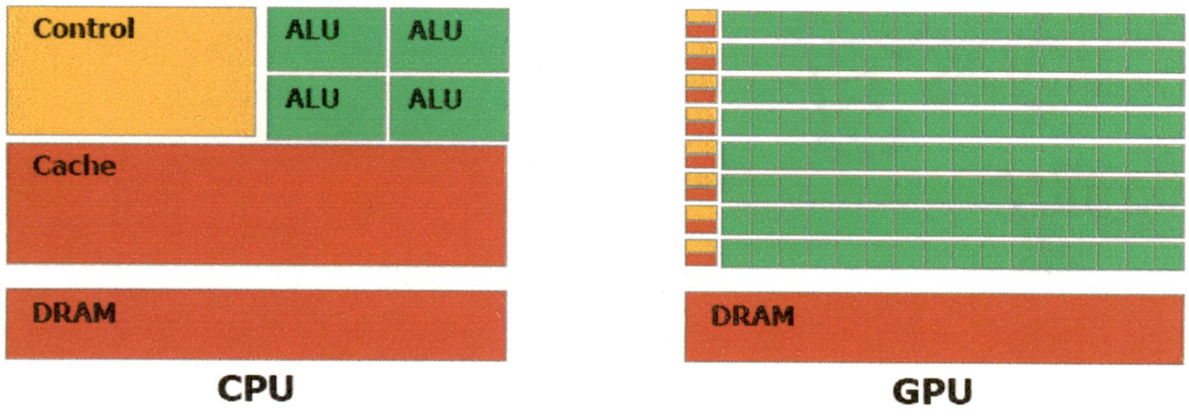

Figure2.3: Comparison of number of transistor devoted to CPU and GPU [14]

GPU has evolved from special purpose processor to a programmable processor. Until recently a graphics API was needed to code on GPUs which made coding for non graphics oriented calculations tough. NVIDIA removed this limitation of programming and introduced a new programming environment known as CUDA (Compute Unified

Device Architecture). CUDA provides access to the native instruction set and memory of the parallel computational elements in GPUs [14].

2.2.1 CUDA Programming Environment

NVIDIA developed a programming environment for CUDA that uses an extension to C language. CUDA programming model emphasizes on mainly two goals. It extends the C/C++ language to ease the programming, and it is designed for writing scalable code which can run simultaneously on tens of thousands of threads running in parallel.

A CUDA program is organized into a host program having a main thread of execution running on CPU, and kernel programs which run on GPU device and invoked by main thread of host. A kernel program is executed by a set of parallel threads. These threads are spawns by the host program at the time of kernel invocation. The threads, thus spawned are organized into a grid of blocks. Each block can contain maximum 512 numbers of threads. This organization is shown in the figure 2.4.

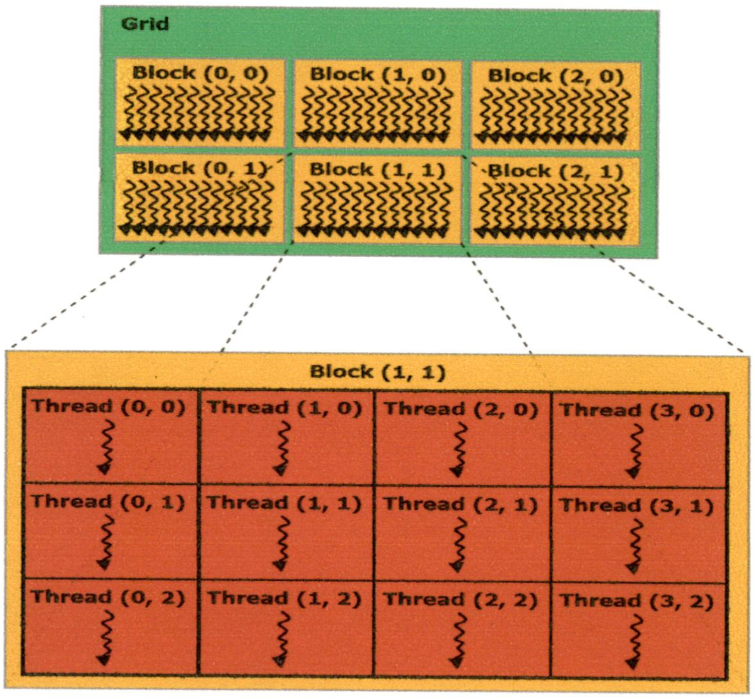

Figure 2.4: A grid of thread blocks and block of threads [14]

A GPU consists of some number of multiprocessors in it. Each multiprocessor consists of eight scalar processors. Host program invokes kernel and spawns threads. Threads are organized at the time of kernel invocation into a number of blocks such that each block contains same number of threads. All the bocks generated are enumerated and distributed to the multiprocessor. Block is further divided into groups of 32 threads. These groups are known as warp. Threads of a warp execute concurrently on a multiprocessor. As soon as execution of one block finishes and multiprocessor become vacant new thread block is launched on vacated multiprocessor.

There are three memory spaces which are accessed by threads of GPU. Figure 2.5 shows the memory hierarchy of the GPU. Each thread has its own memory called local memory. Threads in a block can share a memory which is called shared memory. Threads of one block cannot have access to the shared memory of other block. The third type of memory called global memory. All the threads running on GPU have access to the global memory. Memory bandwidth of these memories is different [14].

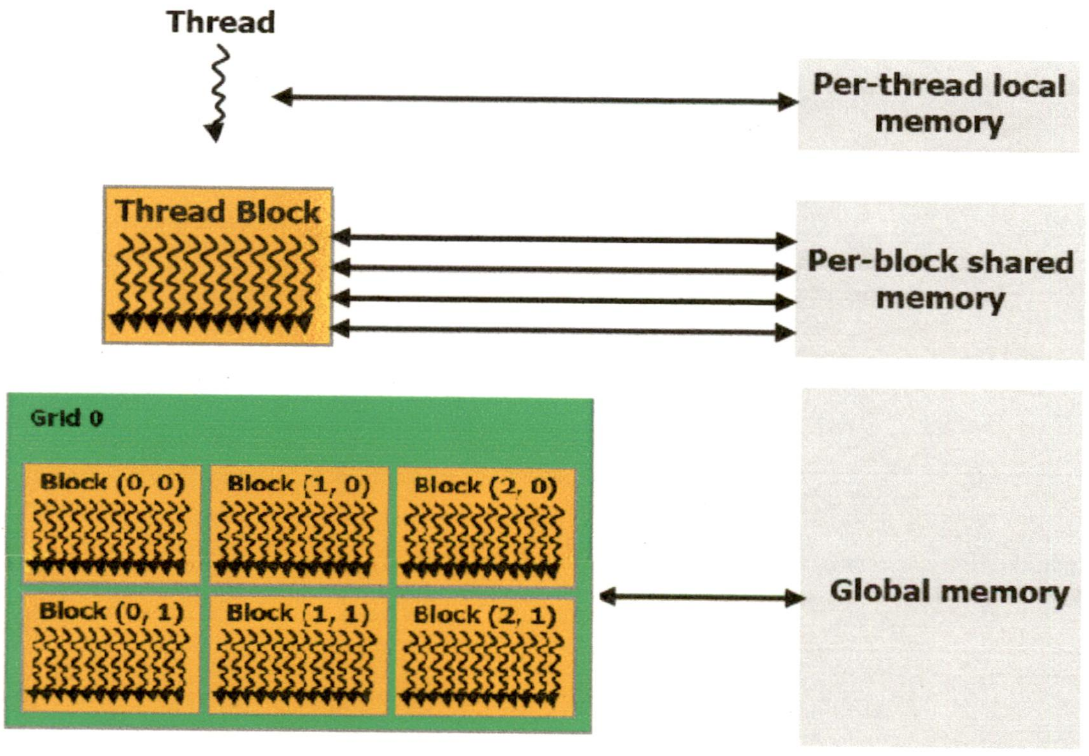

Figure 2.5: Memory hierarchy which can be accessed by threads [14]

2.2.2 Program Execution on CUDA

A CUDA Program is divided into a host program and several kernel programs. Host program is run by CPU; kernel programs run on GPU. These kernels are invoked by host program as and when required. Although the host program uses the legacy C/C++ constructs but kernel program uses extended C/C++ language function. The overall program flow on CUDA is shown in figure 2.6. The data to be used by kernels is copied to the GPU memory space from main memory. As soon as CPU invokes a kernel, all the threads specified in invocation are created and start execution of the kernel code parallely. After the calculation of all kernels is finished result is copied back to the main memory [15].

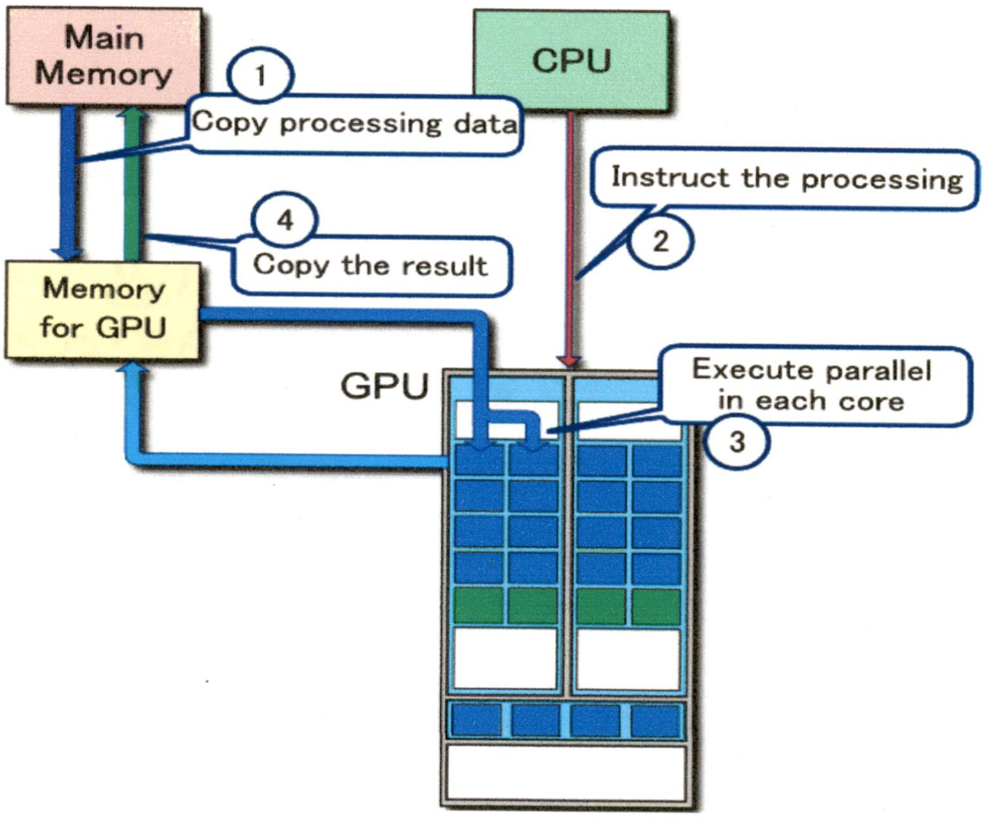

Figure 2.6: Overall program flow in CUDA [15]

CHAPTER 3

QUESTION-ANSWERING SYSTEM AND PAGERANK ALGORITHM

3.1 Question Answering System

A Question Answering (QA) system is considered a next step to the search engine for retrieving information. Unlike to the search engine which produce information in the form of links to documents, a QA system generate a precise answer to the question posed. A QA system is considered more complex than a search engine because of extra steps such as document preprocessing, parsing, indexing, question classification, question keyword weighing, document ranking and answer extraction. Since, effectiveness of a QA system highly depends upon the size of corpus therefore, domain specific QA systems are more likely to be efficient and popular [16].

There has been going a research on QA system for a long time but accuracy and relevance of answer generated to the question is always a matter of interest. A biomedical QA system – INDOC [5], which is designed and developed at IIT Roorkee and is based on the novel ideas of indexing, document ranking and extracting the answer to the question posed. The system achieves an accuracy of 76% over first five documents and increases up to 83% for 50 documents retrieved.

Though the results produced by this QA system are satisfactory in terms of relevance and accuracy, yet it suffers from problem of high response time. In order to reduce the response time, answer retrieval processing should be fast. Besides this, QA systems may have to face another challenge. They need to index a large document set and documents are parsed line by line, therefore, time of processing is very high. Incremental nature of document set produces the need of frequent indexing of document set.

One possible solution to reduce the time of processing is to make a cluster of thousands of computers and run QA system on that [17]. Though the scheme is used widely and considered successful yet it is proved a very complex and costly approach. Beside this, computers in a cluster are connected through a high speed network, which causes a delay and hence, a bottleneck in performance. The technology of cluster can be replaced with a Multicore processor. Cell BE and GPU as discussed in previous chapter have proved very effective in high computational algorithms.

In chapter 4, we provide the implementation of INDOC indexing algorithm on Cell BE Processor.

3.2 Issues of porting a QA system on Cell BE processor

We now enumerate various issues that arise during porting of a QA system on Cell processor.

1. Each SPE of Cell BE has 256KB local store. This local store is shared by code segment and data segment. This limited memory may not allow the entire document to fit on SPE store.
2. The SPEs cannot directly read or write a file. This means that the PPE needs to read files for all SPEs, the SPEs then perform the task of indexing and send back the output to the PPE. Thus, it can potentially become a bottleneck if all these operations are not performed efficiently enough.
3. The NLP toolkits such as the MMTX server [18] are not implemented for the Cell BE processor. Porting them is non-trivial task. Thus these toolkits need to run on the PPE which makes it a bottleneck and puts severe limitations on the amount of gain that could be achieved. Moreover, the APIs provided by MMTX are in Java which cannot be accessed through C/C++.
4. Unlike the multimedia or scientific computing domains where the Cell BE has been largely successful, information retrieval applications tend to involve a lot more of string processing over variable sized strings rather than mathematical calculations over mostly fixed sized matrices or arrays. We thus need to come up with efficient ways leveraging the unique capabilities of Cell BE such as vector processing to manipulate strings.

5. The sizes of the documents involved may also vary considerably. This issue needs to be taken into consideration while designing the overall approach. Otherwise, it may lead to severe load imbalances across SPEs.
6. Work allocation by the PPE has to be done in a way that vouches to keep SPEs equally busy for maximum amount of time.
7. To send a document in parts from PPE to SPE synchronization is required between them.
8. It should be noted that the task at hand has a lot of file processing. This type of data needs to be read sequentially causing a bottleneck in the performance.
9. At the time of retrieving answer to a question, it can be difficult to figure out a global strategy to rank relevance of documents across all SPEs.

3.3 Potential solutions to the issues

This section discusses the solution to the issues arising in indexing the document set.

1. Since limited memory of SPE may not allow entire document to fit at once, the document should be worked upon in such a way that only a part of document is required at a time for indexing purpose. Since SPE cannot directly read the document, PPE is required to read the document and send its contents in parts to the SPEs as and when required.
2. To deal with programming language issues of MMTX toolkit, we can interface the kit programmatically so that they receive input and generate output which is then used for indexing purpose. Thus, Cell processor need not worry about Java APIs. In that case SPE is required to have a networking support.
3. A solution to the problem of variable sized documents is to let PPE read many documents and built a sort of pool of read documents. Whenever an SPE finishes off with its current document, it simply requests the PPE for the next. One may not pre-assign a set number of documents to SPEs. Instead, one can just let them request the documents whenever they need it
4. A large number of DMA operations take place, so instead of sequentially reading line and making DMA, these operations are overlapped. This compensates the time of transferring the content from PPE to SPE.

5. Since many files are needed to be opened on PPE for serving many SPE's simultaneously, many threads can be spawned on PPE so that I/O overhead of opening, closing and reading can be minimized.
6. Synchronization between SPE and PPE can be done through mailboxes.
7. In order to deal with the string processing efficiently, the strings are needed to be converted into vectors and then SIMD operations can be applied to these vectors.

3.4 PageRank algorithm

World Wide Web is a vast collection of extremely diverging WebPages ranging from sports, fun to the journals for information retrieval. Besides this, more than 150 million pages add on to the web in less than a year [8]. In addition to these challenges, web search engine also contend with inexperienced users. Overcoming these challenges, the most used search engine, Google produces a high precision search results. The main reason of its better results is the use of link structure of web to calculate a quality ranking for each web page [7]. This ranking of web pages helps search engines to make sense of the heterogeneity of World Wide Web.

The PageRank algorithm determines the relative importance of web pages. It has become the most important technique used by search engines. The PageRank takes as input a matrix which represents the link structure of web, which runs in size of order in Gigabytes. Thus PageRank calculation is time consuming [9].

There has been a sincere effort to reduce the time of computation of PageRank algorithm. Major stress is given to reduce the IO time of disk access, technique of PageRank calculation is also tried to be improved so that convergence could come in less number of iterations. Besides this, parallel implementation of PageRank on a cluster of computers has also been done.

Chen et. al. [19] has proposed some I/O efficient technique to reduce the disk reads and writes. They analyzed the link structure of the web in detail and perform the preprocessing of the web graph and propose IO efficient algorithm. Their approach shows significant benefits over original PageRank algorithm when main memory of the system to be worked upon is very small of the order of MBs. But in real scenario main memory size has been increased very much therefore their approach becomes of no use.

Another technique for solving rank of web pages which exploits block structure of web was presented by Kamvar et. al. [20]. Web graph has majority of hyperlinks which link pages on a host to other pages on the same host, many of those that do not link pages to within the same domain. They exploited this structure of web by a 3 stage algorithm whereby (1) the local ranks of pages for each host are computed independently using the link structure of that host, (2) these local ranks are then weighted by the importance of the corresponding host, and (3) the standard PageRank algorithm is then run using as its starting vector the weighted aggregate of local PageRanks. They achieved a speedup of 2 times with this approach.

Manaskasemsak et. al. [21] presented a parallel PageRank Computation on a Gigabit PC cluster. They conducted this experiment on a large web graph of over 1.5 billion links and their implementation took only 15 seconds for one iteration. They addressed the issues of porting PageRank on cluster and communication required between PCs. Again Manaskasemsak et. al. [22] presented a comparison study on the bases of I/O cost, memory usages and synchronization cost with other two techniques.

PageRank is a highly computational intensive and Cell BE Processor is also designed for computational intensive algorithm. With this idea, Brehrer et. al. [23] implemented PageRank algorithm on Cell BE. But, because of large number of random memory writes, and data transfer between PPE and SPE required by PageRank algorithm, implementation took more time than on single processor Xeon. They also presented a comparison of time taken by different processors to calculate ranks of pages for a particular graph and found that their implementation on Cell BE is 22 times slow in comparison to Xeon processor.

There may be a scope of improving the performance on Cell BE by reducing the data transfer between SPEs and PPE. Besides this, if vector operations of Cell BE are properly applied during calculation, time of calculation can be reduced drastically. In the chapter 5 we are going to implement PageRank on Cell BE processor by a new technique which reduces the data transfer between SPEs and PPE and utilizes SIMD operations.

3.5 Implementation issues of PageRank on Cell BE Processor

1. PageRank operate on a huge amount of data. To achieve a better performance gain all calculations should be done on SPEs. Since SPE's local store is small (256 KB) and data to be worked upon is large and available at PPE, therefore a large number of DMA transfer need to be done between PPE and SPE producing a bottleneck in performance.
2. PageRank requires copying an array of output ranks to the array of input ranks and both array are present at PPE, therefore this operation cannot be done on SPE. This means that only some part of PageRank algorithm can be parallelized on Cell BE.
3. Rank of a particular node depends upon any number of nodes in the complete range of nodes. That means data to be worked upon is not continuous (rather scattered in memory arbitrarily). So on the direct input, data level parallelism is not possible. To achieve data level parallelism some modification are required.
4. Since DMA is done on sequential data while requirement in PageRank is of any random node. Thus many DMA may be required for small data.
5. PageRank implementation requires random writes and reads but Cell BE Processor is not good at this point.

3.6 Possible solutions to the issues

1. Since the complete algorithm cannot be ported on SPEs, therefore some parallelization technique like SIMD can be applied on that part which is done at PPE. In case of copying with a large array, loop unrolling can also be applied.
2. In order to provide data level parallelism and reducing the unnecessary DMAs, data is first arranged on PPE in such a way that the division of data corresponding to SPEs always lead in such partition that each partition contains nodes whose PageRank

depends upon only nodes of that groups. This remedy may also require insertion of some redundant data.

3. Data to be fetched by SPE's must be arranged in continuous memory locations for reducing the DMA overhead.

Thus the study of both a question answering system and PageRank reveals that both problems are computationally heavy and have a scope of parallelization. In the following chapters these two problems will be implemented on Cell BE processor.

CHAPTER 4

INDEXING ON CELL BE PROCESSOR

4.1 INDOC- Introduction

INDOC [5] is a biomedical Question Answering system based on idea of indexing and extracting the answer to the question posed. Major tasks in it are indexing, question processing, document ranking, clustering, and display results. Complete architecture of INDOC is shown in figure 4.1.

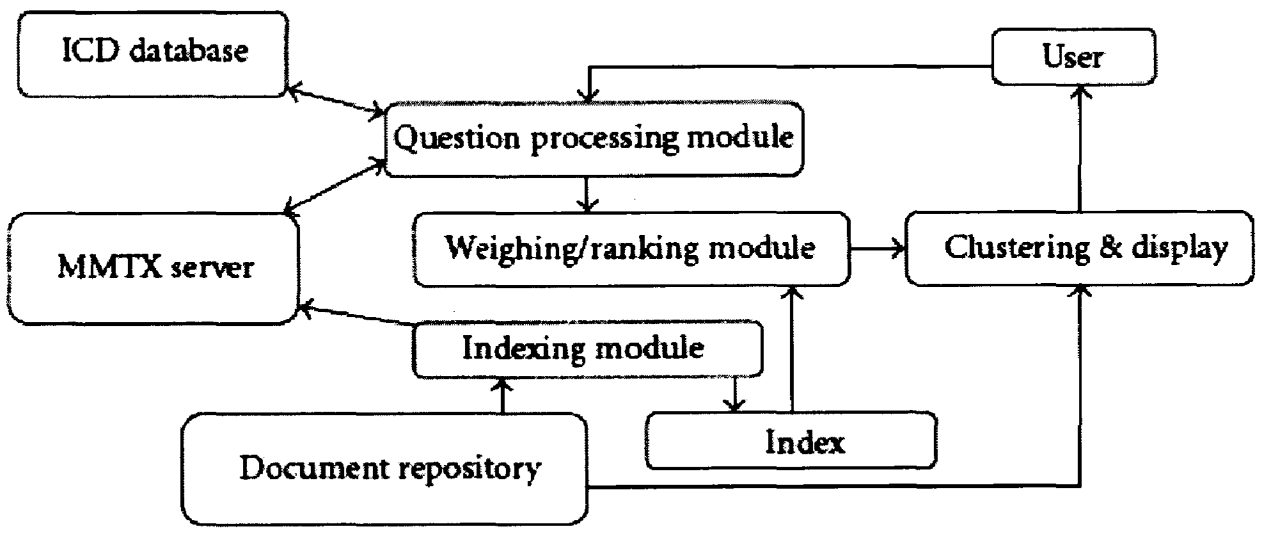

Figure 4.1: Complete Architecture of the INDOC [5]

4.1.1 MMTX Server

MMTX server [18] is a program which maps the free text received by it into the UMLS concepts [24]. This program is used by indexing module to make an indexed database from a document repository and Question processing module to find concepts of the question.

4.1.2 Indexing Module

Indexing module takes input a document repository. An indexed database is prepared with the help of MMTX Server. Indexing here is not just to select important keywords, rather documents are represented in the form of sections and these sections are actually indexed on the basis of concepts present in a particular section. At the time of document retrieval, a document is considered useful if some or all question concepts are present in one section.

4.1.3 Question Processing Module

As soon as the query arrives, concepts present in the question are extracted with the help of MMTX server. Since all the concepts are not of equal importance therefore a relative weight is assigned to each concept. These weights become useful in the determination of document Ranking. Further this, the concept with highest weight is sent to ICD database [25] in order to find the related concepts. Answer retrieval is made on all these concepts.

4.1.4 Document Ranking

On the bases of concepts found in question by the question processing module, for each document all those sections which contain at least one concept are of interest. All such sections are assigned weights. Weight of a section is equal to the sum of weight of concepts of the sentence present in it. Weight to the document is equal to the weight of the best section and number of lines in the best section of document. Documents are then ranked in decreasing order of their weights.

Thus, first the entire document set is processed by the indexing module and an indexed database is prepared. At run time, as soon as the system receives a query, the question processing module recognizes the keywords of question and finds the UMLS concepts corresponding to these keywords with the help of MMTX server. The ranking module searches the indexed database for retrieving the documents and assigning them a rank on the basis of their relevance to the question concepts. Finally the display module displays the documents in decreasing order of their weights.

4.2 INDOC – Indexing

Indexing module for preparing indexed database not only selects the important keywords and concepts from document; but also represents the entire document in the form of sections. Each section has a section heading and number of sentences in it. Section heading consists of concepts that represent the section. One sentence can belong to only one section and a section can contain only consecutive sentences [5].

4.2.1 Algorithm

The algorithm to perform the task of indexing is shown in figure 4.2.

Algorithm in Figure 4.2 obtains the concepts of title and stores them in file. Beside this, sections are formed on the basis of concept present in the sentences. A new sentence is added to the current section till intersection of concepts of the current section and sentence to be added is not empty. But there are two restrictions on the size of section.

If size of the current section < M (a Const.), the sentence is added to the section and section concept will be intersection of concepts of the current section and sentence to be added.

If size of the current section > M, the sentence is added to the current section only if sentence's concept are a subset of concepts of the current section.

If size of the current section < L (minimum number of sentence in a section), then the current section is merged with previous section.

An indexed file containing the concepts of titles of all the documents in the document set is also prepared. This file is used by the ranking module at the time of retrieval of documents corresponding to a question submitted by the user.

```
1. Obtain the concepts of the title and store them.
2. Initialize i =1 and j=1 and set all Xi, SCj, XCi to be empty where
        Sj : jth sentence in the document.
        Xi : ith section.
        SCj : set of concepts in jth sentence (concepts in an individual sentence).
        XCi : set of concepts in ith section.
        L : min number of sentences necessary in a section.
        M : minimum number of sentences in a section so that merging is not
        necessary.
3. Formation of Sections
        Set XCi to concepts in the first sentence.
        Define | S | as the number of elements in set S.
        For each sentence Sj left in the document to process
        {
                If(|Xi|==0)
                {
                        Add Sj to Xi
                        Add SCj to XCi
                }
                else
                {
                        if( (|Xi|<M && |Xci∩SCj|>0 ) || XCi==SCj )
                        {
                                Add Sj to Xi
                                Set XCi= XCi∩SCj
                        }
                        else
                        {
                                i=i+1
                                Add Sj to the new section Xi
                                Add SCj to XC
                        }
                }
        }
4. Final Section merging step
for each section Xi
{       If( i>1 && ( |Xi|<L| || XCi is a subset of XCi-1) )
        {       Merge Xi with Xi-1
        }
}
```

Figure 4.2: Indexing Algorithm of INDOC [5]

4.3 Design and Implementation of INDOC Indexing algorithm on Cell BE

This section presents the implementation of some of the solutions provided in chapter 3 by applying them while porting indexing algorithm of biomedical QA system, INDOC on Cell BE processors.

Cell BE offers a number of ways to achieve parallelism viz. SIMD processing, multithreading, shared memory multiprocessing, multiple execution units with heterogeneous architectures. Of all these, we have selected to use multithreaded and double buffering approaches to achieve data level parallelism. The data is partitioned such that entire document set is divided into eight subsets and one thread at PPE corresponding to one SPE, is responsible for assigning one such document subset to that SPE. We have done so because I/O operations in opening, reading and closing files by multiple threads may be overlapped with other computations. The logic used on PPE and SPEs is as follows:

4.3.1 PPE

The main task of PPE is to read files for the SPEs. PPE creates one thread each for one SPE and has one file allocated per SPE. Once this specific file is indexed completely, PPE picks up the next file for that particular SPE. PPE reads the file character by character until it reads one complete line in a temporary buffer. Later the corresponding SPE is directed to pick up this line of text by using SPE read inbound mailbox. Status variable of mailbox represents the number of free entries in mailbox. Initially its value is 4. A write operation on mailbox by PPE decreases the status value by 1 and read operation by SPE increases the value by 1.

Value of status is checked repeatedly. If the value is 4 then temporary buffer is copied to original buffer whose address is available to the SPE. Status variable is updated to 3 to indicate that buffer is ready to be read by SPE. Status variable is updated with a write operation to the mailbox value. This value is used to indicate the SPE about end of file. SPE reads the line with DMA operation and perform the task of indexing on this line of text. During this time PPE is busy reading the file, constantly generating raw data for the 8 SPEs. Working of PPE is shown in figure 4.3.

Value of the status variable of mailbox is 4 when it is reset by the SPE after completing DMA. As soon as PPE updates the buffer, it sets the status to 3 to indicate SPE about update of buffer. In this way tasks occur simultaneously both on the PPE and the SPE.

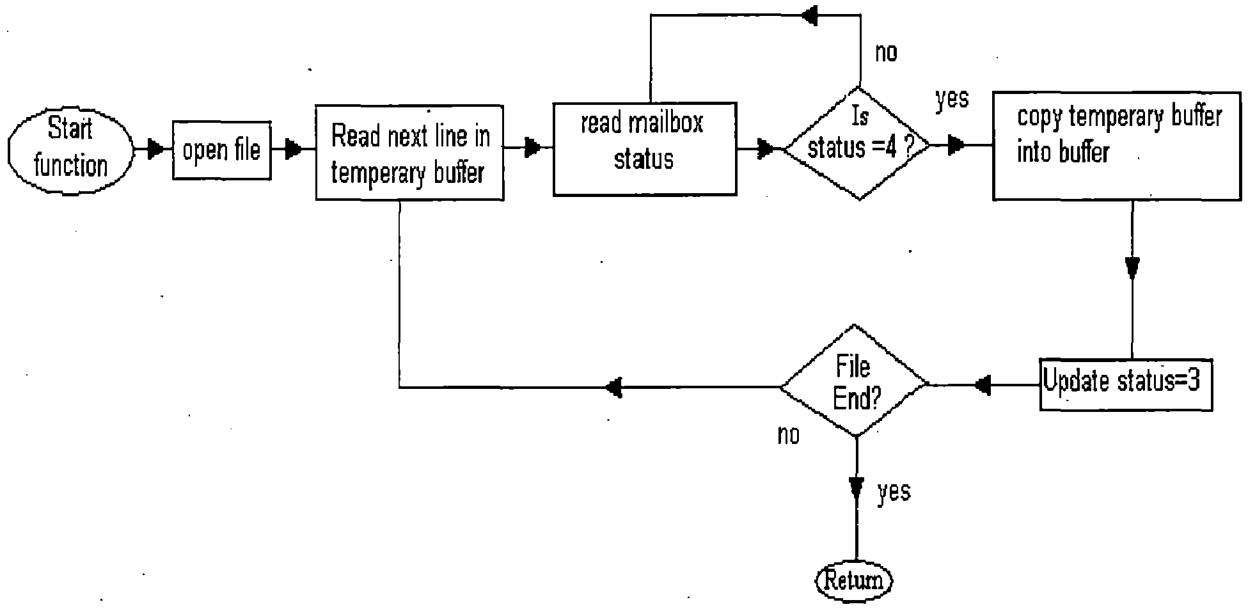

Figure 4.3: Working of a PPE Pthread

4.3.2 SPE

SPE has the task of creating sections of the input lines that were sent to it from the PPE. To receive the data from main memory (PPE) to the SPE we use DMA operations. This also allows us to make use of double buffering.

Initially value of status of mailbox is 0 therefore SPE waits to perform DMA until status becomes 1. As soon as a DMA operation of taking one line from PPE to SPE is over, the mailbox which was earlier used by PPE to signal to SPE to indicate the presence of a new line of data is now used by the SPE to ask for next line of data. Here value of status variable represents number of occupied entries in the mailbox. Initially it is 0. A write operation by PPE increases the value of status by 1 and read operation by SPE decreases the value by 1. The value read from mailbox is helpful in deciding whether the document/ file has finished or not. Working of a particular SPE is as shown in figure 4.4.

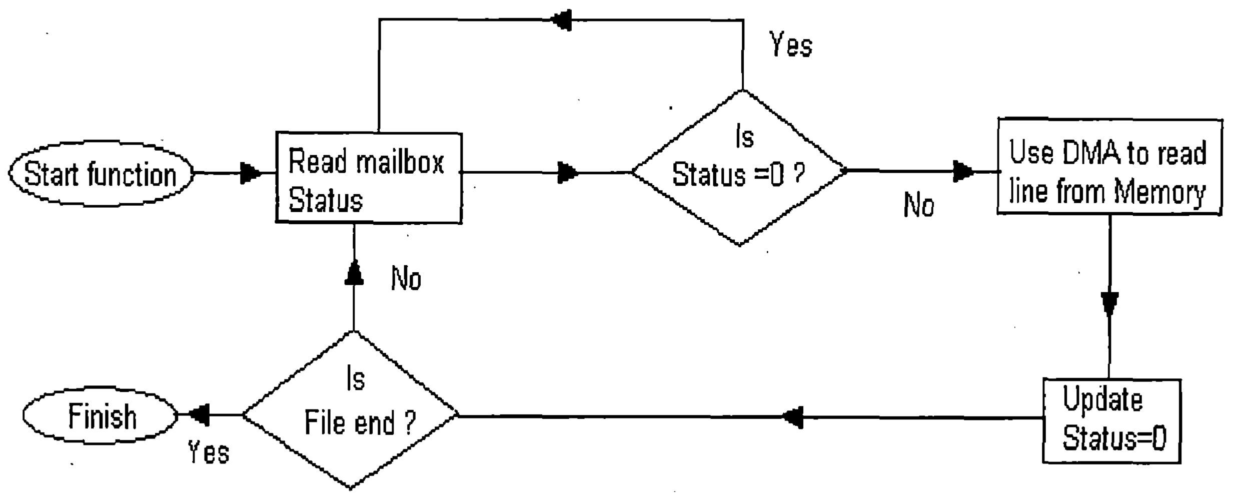

Figure 4.4: File read operation by SPE and signaling to PPE for next read

Status variable of the mailbox in the SPE remains 0 till the buffer is not updated by PPE. As soon as SPE finds the values of status non zero, it starts DMA to read the buffer. After the DMA operation is over, it resets the status to 0 by reading the mailbox. On the basis of read value from mailbox it is checked whether file ended or not. The line just read is used for making sections. Since DMA is a non blocking call, task of processing current line on SPE overlaps the next DMA operation.

As soon as the line is received, a counter which keeps track number of sentences is incremented. If the sentence is first sentence then first section is initialized with the sentence. Current sentence is added to the current section if either of the following two conditions is satisfied: (1) whether the length of intersection of concepts of current section and current sentence is greater than 0 and length of current section is less than a const and (2) whether the number of concepts in current section and current sentence are equal. If neither of the condition is satisfied then a new section is made and initialized with current sentence. Here the operations performed are mainly subset finding, string comparison, string matching therefore, vector operations were difficult to be applied.

4.4 General observations

During the implementation of indexing algorithm on Cell BE Processor, some important observations worth pointing out are as follows:

1. The size of the documents was varying greatly. This causes an imbalance of work among the SPE's. For maximum speed up we would ideally want all the SPEs to stay busy for same amount of time. For example, in video compression, the sizes of all video frames are same as against our case.
2. In some of the cases, it was observed that synchronization between the PPE and SPE resulted in some periods of time when either of the machines was idle. For example if we have a long line followed by a small line, in such a case the DMA transfer of the long line shall take considerable more time keeping the PPE idle for some time.
3. While implementing the indexing system for INDOC, it was observed that the issues encountered were likely to be fairly general to a lot of other QA systems.

4.5 Results

Performance of the code built for Cell BE is evaluated on Georgia Tech Cell Buzz [26]. Number of documents used for measuring performance has been varied from 8 – 40. Three samples were taken for each observation and an average time was calculated. Results show a comparison with Intel Xeon dual core (2.0 Ghz) processors with 2 GB RAM. Table 4.1 shows the observed comparison. It is found that speedup in execution times on Cell Buzz is about 22+ times against Intel Xeon.

From table 4.1, It can be observed that there is variation of speedup for different number of documents. The reason behind this is variation in size of documents. Size of documents vary from 1 KB to 95 KB , which actually hinders the speedup because of an imbalance of work distribution among the SPEs.

Table 4.1: Comparison of execution times

	8 docs (µs)	16 docs (µs)	24 docs (µs)	32 docs (µs)	40 docs (µs)
8 SPE	2194 4270 3703	6769 7749 7043	9210 7229 8614	9771 8738 9274	14901 10814 10233
Average(t1)	3389	7187	8351	9261	11982
Intel Xeon (2.0 Ghz)	86440 87072 87030	161749 164603 163615	216708 218688 215714	259634 259771 258467	310906 314959 312666
Average(t2)	86847	163322	217036	259290	312843
Speed up (t2/t1)	25.6	22.7	25.9	27.9	26.1

On the other hand if we take documents of same size for measuring performance, results are consistent and better because of equal amount of work for all SPEs. The results are shown in table 4.2.

Table 4.2: Table showing low variation in speedup when all documents are of same size.

	8 docs (μs)	16 docs (μs)	24 docs (μs)	32 docs (μs)	40 docs (μs)
8 SPE	780 731 791	1542 1403 1591	2232 2214 2209	3212 3020 3149	4067 3518 4220
Average(t1)	767	1512	2218	3127	3935
Intel Xeon (2.0 Ghz)	30239 29955 30362	59741 60050 60219	90173 90094 88071	120283 120352 117889	150184 150136 150782
Average(t2)	30065	60003	89446	119508	150367
Speed up (t2/t1)	39.1	39.7	40.32	38.2	38.2

Figure 4.5 shows the comparison of timing observed keeing number of documents same but increasing the number of SPUs. It was observed that as we increase the number of SPUs the time of computation also decreases.

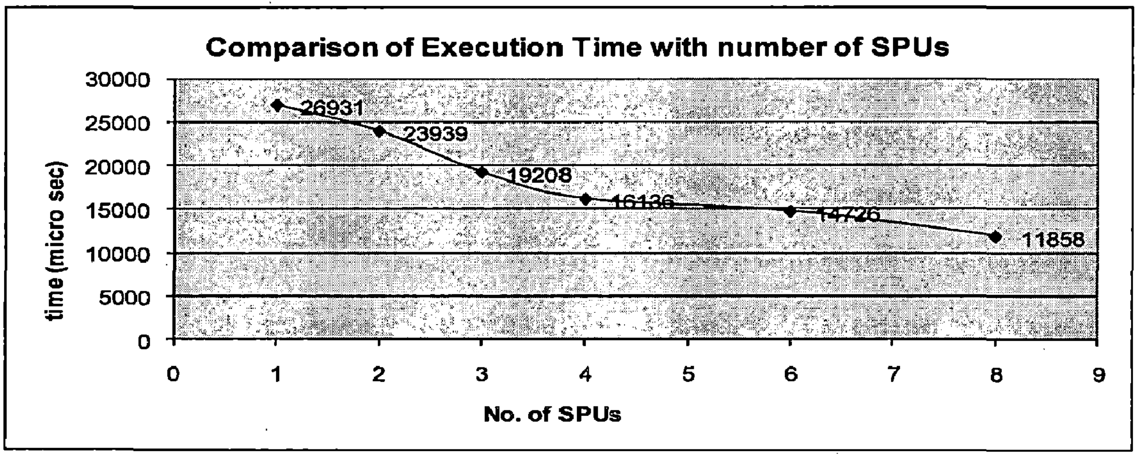

Figure 4.5: Comparison of time with number of SPUs

CHAPTER 5

PAGERANK ON CELL BE PROCESSOR

5.1 PageRank

PageRank is an algorithm to determine the relative ranking of web pages among them. The concept of PageRank is based on an idea that if a page v of interest has many other pages u with pointing to , then the pages u are implicitly conferring some importance to page v. Let C(u) be the number of links which page u points out, and let PR(u) be the rank of page u, then hyperlink $u \to v$ confers $PR(u)/C(u)$ units of rank to page v.

Mathematically, page rank of a web page at any iteration can be defined as follows:

$$PR_i(A) = (1-d) + d * \left(\frac{PR_{i-1}(T_1)}{C(T_1)} + \text{--------} + \frac{PR_{i-1}(T_n)}{C(T_n)} \right) \text{-------------------------} \ 5.1$$

Where,

PR_i (A) is the PageRank of page A, calculated in ith iteration.

PR_{i-1}(Ti) is the PageRank of pages Ti which link to page A, calculated in i-1th iteration.

C(Ti) is the number of outbound links on page Ti and

d is a damping factor which can be set between 0 and 1.

So, first of all, we see that PageRank does not rank web sites as a whole, but is determined for each page individually. Further, the rank of page A is recursively defined by the rank of those pages which link to page A. Rank of a page T which links to a page A, does not influence rank of A uniformly as the rank of a page T is always weighted by the number of outbound links C(T) on page T. This means that more outbound links a page T has, less will page A get benefit from a link to it on page T. Figure 5.1 shows how PageRank from one page to another page are passed.

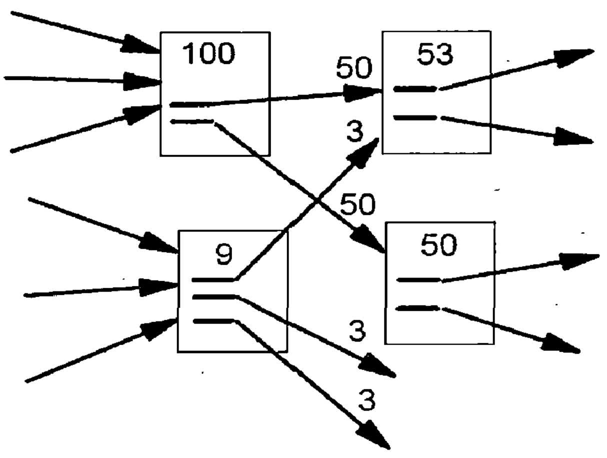

Figure 5.1: Rank contribution from one page to another[8]

The PageRank algorithm is based on the idea of Random Surfer Model [8]. The probability that a random surfer clicks on a link is given by the number of links on that page. Therefore, the probability for the random surfer reaching one page is the sum of probabilities for the random surfer following links to this page. Now, this probability is reduced by the damping factor d. The justification within the random surfer model, therefore, is that the surfer does not click on an infinite number of links, but gets bored sometimes and jumps to another page at random. Damping factor d is the probability for the random surfer not stopping to click on links. Since the surfer jumps to another page at random after he stopped clicking links, the probability therefore is implemented as a constant (1-d) into the algorithm. Regardless of inbound links, the probability for the random surfer jumping to a page is always (1-d), so a page has always a minimum PageRank [8].

5.2 Link Structure of Web

World Wide Web can be considered as a directed graph where each web page is treated as a node of graph and hyperlinks as edges of graph which is known as web graph. Every node of web graph has some number of forward links and backward links. A web graph is the input to the PageRank algorithm. Standard web graphs are available on WWW in compressed form which can be accessed through [27]. The web graphs which are used for experiments are: EU-2005, IN-2004, and CNR-2000. These are prepared with ubi-Crawler[28]. These compressed graphs are extracted into a text file with the help of a tool called WebGraph [29]. WebGraph is a framework to study the compressed web graph. It provides APIs in JAVA, Matlab and C++ to explore the web graph. It provides simple ways to manage very large graphs by exploiting modern compression techniques and can be used to generate the web graph in a text file [30].

The output text file containing the web graph has three columns in it. First column consists of one integer referring to the destination webpage. Second column refers to the in-degree of the web pages referred in column one. The last column refers to the source web pages to the web page referred in column one. Structure of Text file containing the web graph is shown in figure 5.2

(dest_id)	(in_degree)	(Source_nodes)
1	2	3 7
2	4	4 5 7 9
3	3	2 7 9
4	1	1
-	-	-------

Figure 5.2: Structure of text file containing web graph

5.3 Algorithm

```
1. Two arrays V1[] , V2 [] for having ranks of nodes in ith and (i+1)th iteration.
2. T is total number of nodes.
3. d is damping factor.
4. $C_i$ is number of out links of node i.
5. for: all u = 1 to T
6.      V1[u] := 1;
7  for: number of iterations do{ // loop1
8.      for: all u = 1 to T
9.           V2[u] := 0;
10.     for: all nodes x in the web graph do { // loop2
11.        for all source nodes y of node x do{ // loop3
12.            V2[x] = V2[x] + V1[y]/$C_y$
13.        }
14.        V2[x] = (1- d) + d* V2[x];
15.     }
16.     V1= V2;
17.}
```

Figure 5.3: Algorithm of PageRank calculation

5.4 Implementation of PageRank on Cell BE Processor

5.4.1 Design 1

First of all, we make two arrays V1[] and V2[] to keep rank for (i-1)th iteration and ith iteration at PPE. Another array GRAPH[] is made to keep nodes and their source nodes.

For one iteration,

1. Divide all nodes of the web graph equally among all SPEs.
2. Calculate rank of those nodes (on a SPE) using array V1[] and GRAPH[], and then send back the ranks to PPE.

3. Receive ranks at PPE in a vector V2.
4. Update V1 by V2.

Since the number of nodes dedicated to an SPE is very large, no SPE can accommodate all dedicated nodes at same time to calculate rank. So the SPE brings nodes in, using multiple DMA, few nodes at a time. In one time it brings as many nodes it can accommodate. Since the rank of any particular node (e.g. node x on SPE) depends upon any number of source nodes (which point to node x) and these source nodes can be present in any section of the input rank array V1, so we need complete rank vector V1 in SPE (for calculation of any given node x). SPE cannot accommodate V1 into it, so it will be brought in parts (for calculation of each given node x). It may also happen that for each node to calculate its rank complete v1 is brought. Thus number of DMA operations will be large. Therefore the approach seems of no benefit.

5.4.2 Design 2

Following the idea of approach 1, and making some changes in data structures, DMA operations can be decreased by significant amount of times.

The data structures design is as follows:

1. The web graph is read on PPE and stored into two arrays such that,
   a. Array1 (referred as Node array) contains
      i. First node followed by its in-degree, followed by the source nodes, then
      ii. Second node followed by its in-degree, followed by its source nodes then
      iii. Third node and so on.

      n1| deg1| s1|s2|s3.....|n2|deg2|s4|s5|s6|.....

      here n1, n2 are nodes.

      deg1, deg2 are in-degree of n1, n2 respectively and s1, s2 represent the source node to node n1.
   b. Array2 (referred as Degree array) contains the out-degree of nodes in the indices corresponding to source nodes in Array1

n1|deg1|d1|d2|d3|....|n2|deg2|d4|d5|d6......

here n1 and n2 are same as in array1, deg1 and deg2 are same as in array1 but di represents the out-degree of node si (present in array1)

2. Two arrays V1[] and V2[] are maintained to keep rank of nodes at ith and (i+1)th. V1[] is used as a reference array containing the page ranks as calculated from the previous iterations and used in calculation of V2[].
3. Size of V1[] is equal to the size of array1[] and array2[] while size of V2[] is equal to number of nodes. Here thing to be noted is that V1[] contains rank of all nodes in the sequence same to the sequence of nodes of array1. That means there is redundancy of rank value of a particular node several times in V1[]. This is because, a particular node may be the source node of multiple nodes and hence present multiple times in array1.

Algorithm proceeds in the following way,

1. Array V1 is initialized to 1.
2. For each iteration,
   i. Array2 (array of degree) and V1 are equally divided among number of SPEs.
   ii. Since number of nodes dedicated to an SPE is large so SPE reads array2 and V1 in parts. In one time SPE reads as many elements of arry2 and V1 as it can accommodate in its local store. Since a particular node, its in-degree, source nodes and their out-degree all are present in array2 and V1 so rank of node can be calculated easily, and same section of V1 need not be read again for calculations of two nodes. As soon as the rank of nodes is calculated in one time it is sent back to the PPE where it is stored in V2.
   iii. As soon as all SPEs calculate the rank of all nodes dedicated to them and V2 is updated at PPE, V1 is updated from V2.

5.4.3 Implementation details

This section presents the implementation details of the PageRank algorithm. The approaches used to achieve data level parallelism at PPU are multithreading and loop unrolling while at SPU are double buffering and SIMD. The logic used on PPE and SPEs is as follows:

PPE

Processing starts with PPE by reading the web graph and preparing data structures. As soon as data structures are prepared, PPE spawns pthreads equal in number to SPEs. Each pthread spawns a SPE thread. As soon as SPE thread is created, it starts calculating rank of nodes assigned to it. During this time PPE's pthread goes into a blocking wait giving control to other ptheads of PPE while waiting for a signal by SPE. Figure 5.4 shows the overall working of PPE and SPEs for one iteration of PageRank algorithm. Pthreads update their data structures from the shared memory which is updated by corresponding SPE. Shared memory synchronization is required between pthread and SPE.

SPE

As soon as the SPE thread is created by PPE, SPE starts reading two arrays of degree and rank. Since the task is equally divided among all SPEs so each SPE reads from a particular array location which is determined by the number of SPE. Each SPE starts reading at (SIZE/N)*i location where size of arrays is given by SIZE, N represents total number of SPEs and i is between 0 to N-1 for different SPEs. SPE reads data from memory through DMA operation. Since DMA transfers are limited by 16KB per transfer, therefore only 4096 integer elements can be brought at one time. Thus 4096 elements of degree array and 4096 elements of rank array are brought by two successive DMA transfer. The calculation of PageRank is done with SIMD operation. New rank of nodes present in these 4096 elements is calculated and sent back to PPE by writing into the shared memory. Before writing into shared memory SPE waits for a signal by PPE. After writing into the shared memory SPE sends a signal to PPE about the update of memory and start reading next data from input arrays.

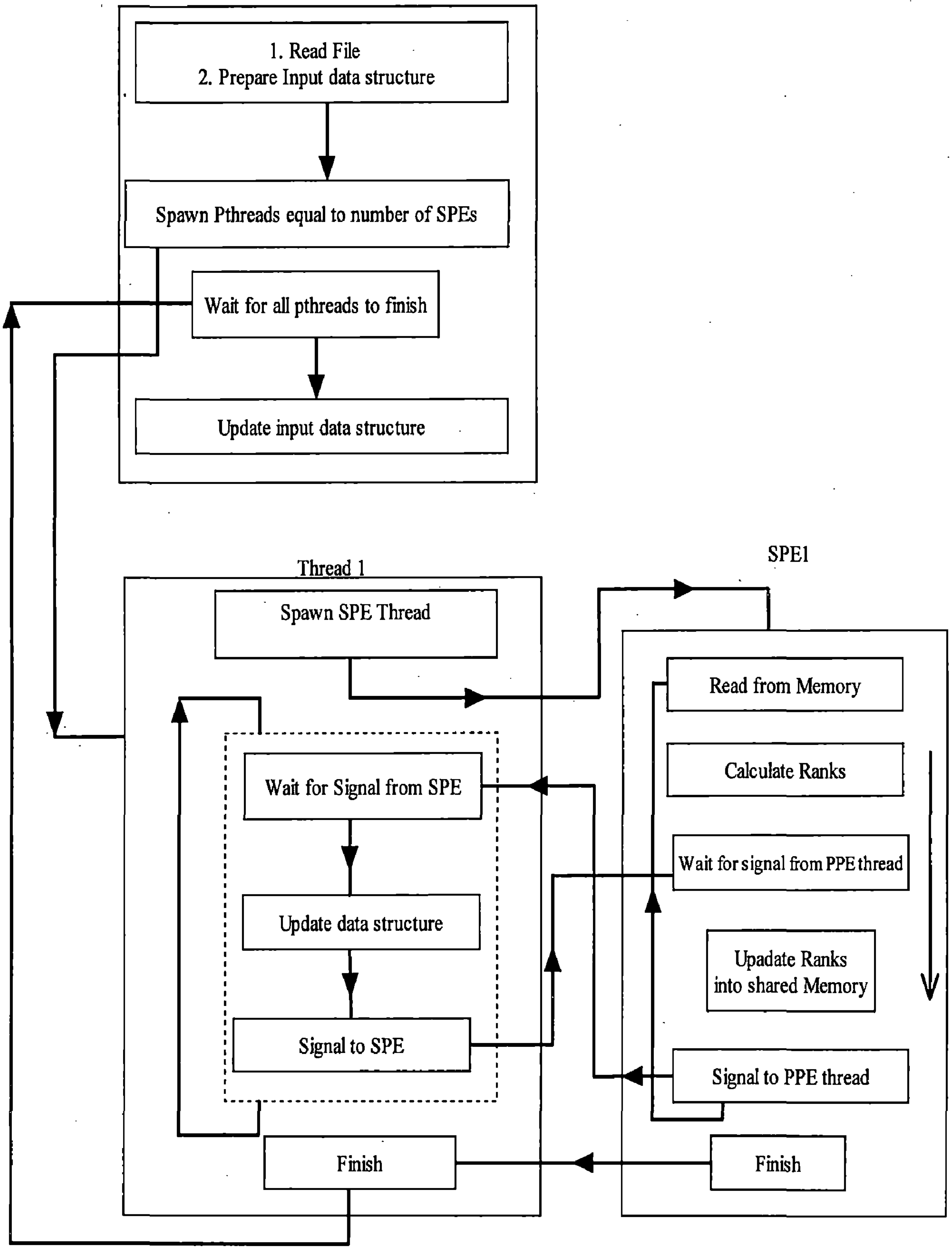
1. Read File
2. Prepare Input data structure
Spawn Pthreads equal to number of SPEs
Wait for all pthreads to finish
Update input data structure
Thread 1
Spawn SPE Thread
Wait for Signal from SPE
Update data structure
Signal to SPE
Finish
SPE1
Read from Memory
Calculate Ranks
Wait for signal from PPE thread
Upadate Ranks into shared Memory
Signal to PPE thread
Finish

Figure 5.4: Overall working of Cell BE processor for PageRank Algorithm

Synchronization

Communication and shared memory synchronization between PPE thread and SPE is achieved with mailbox. The mailbox used is SPE write outbound mailbox. The SPE informs PPE each time after updating shared memory. The status of mailbox at PPE is 1 when mailbox is full and 0 when mailbox has been read by PPE while at SPE its value is 1 when there is no data in it and 0 when SPE writes data in mailbox. The communication synchronization between PPE and SPE is shown is figure 5.5.

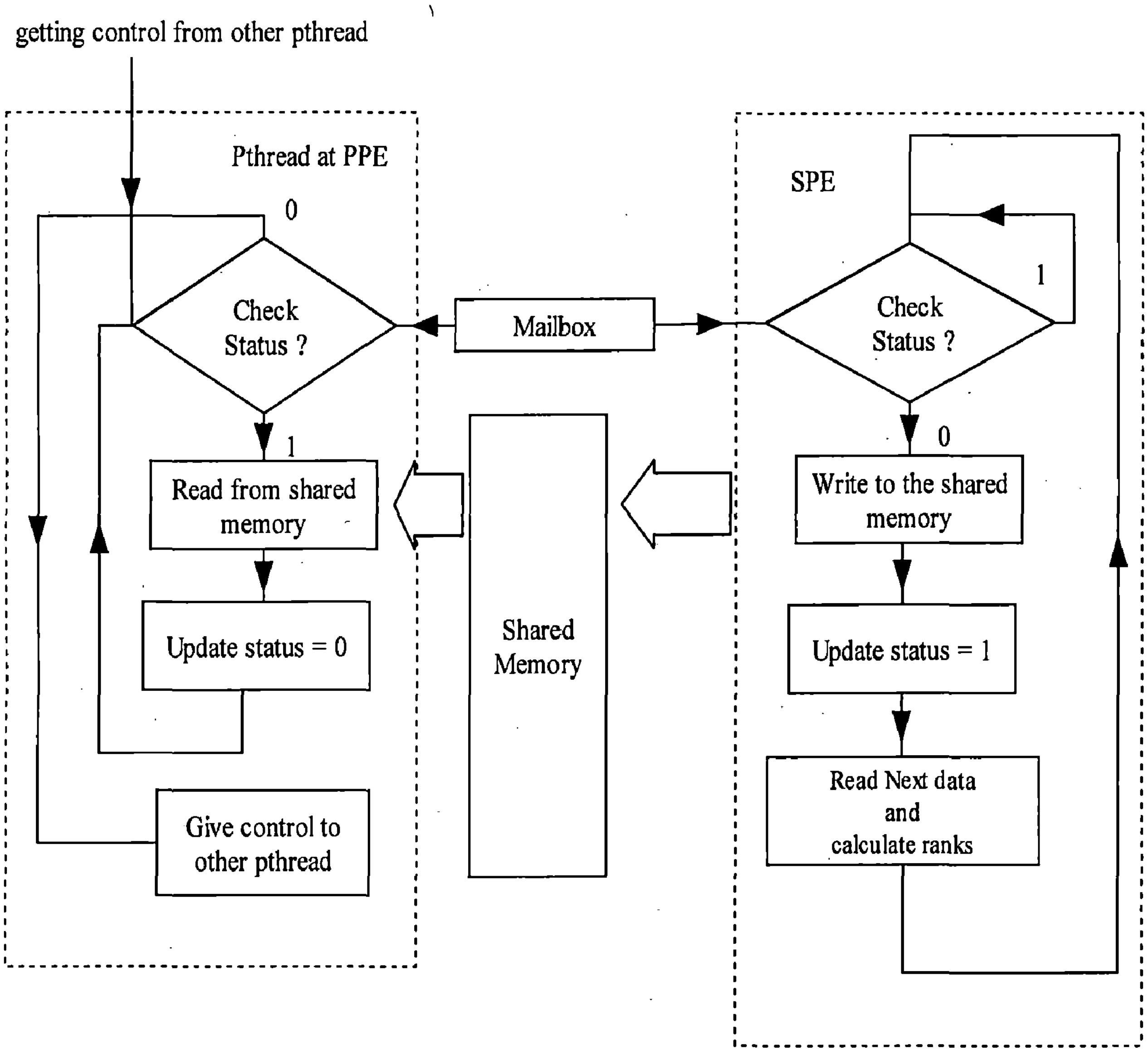

Figure 5.5: Synchronization between PPE thread and SPE

Data Flow in Memory

Web graph is read from disk into main memory and maintained by PPE. Graph is actually kept into two arrays one for holding degrees of nodes and other for holding rank. SPE are directed to read these arrays equally from different locations. The complete flow of data in memory is shown in figure 5.6. Data read by SPEs from main memory is shown by path1 <1>. SPEs calculate PageRank and update the shared memory between PPE and particular SPE though path 2 <2>. Threads at PPE receive data from shared memory through path3 <3>and finally update the input array of ranks through path4 <4> for the next iteration.

5.5 Results

The PageRank algorithm works upon a large web graph in practice. Therefore, the web graphs used for experiments are EU – 2005 and In-2004. EU-2005 graph contains 862664 nodes and 19235140 links. Graph In-2004 contains 1382908 nodes and 18534900 links. The Cell BE processor used for execution and testing results is Cellbuzz provided by Georgia Tech University [26]. Comparison of Cell BE implementation is done with Xeon dual core 2.0 Ghz. The measurement of time is done for 1, 2, 4, 8, 16, 32 number of iterations of PageRank calculations.

Table 5.1 shows results obtained for both Xeon and Cell BE processor for graph EU-2005. It shows a constant increase in time with increase in number of iterations. Also, the ratio of time between Xeon and Cell BE processor was nearly constant for all iterations. The Cell BE processor shows a speed up of nearly 5% over Xeon. The speed up obtained over Xeon does not show a marked improvement but when compared to Brehrer et. al. [23]'s implementation of PageRank algorithm on Cell BE, our implementation of algorithm is 22 times faster.

Figure 5.7 shows the comparison of execution time between Xeon and Cell BE for graphs EU-2005 and In-2004. These timings are for 32 iterations.

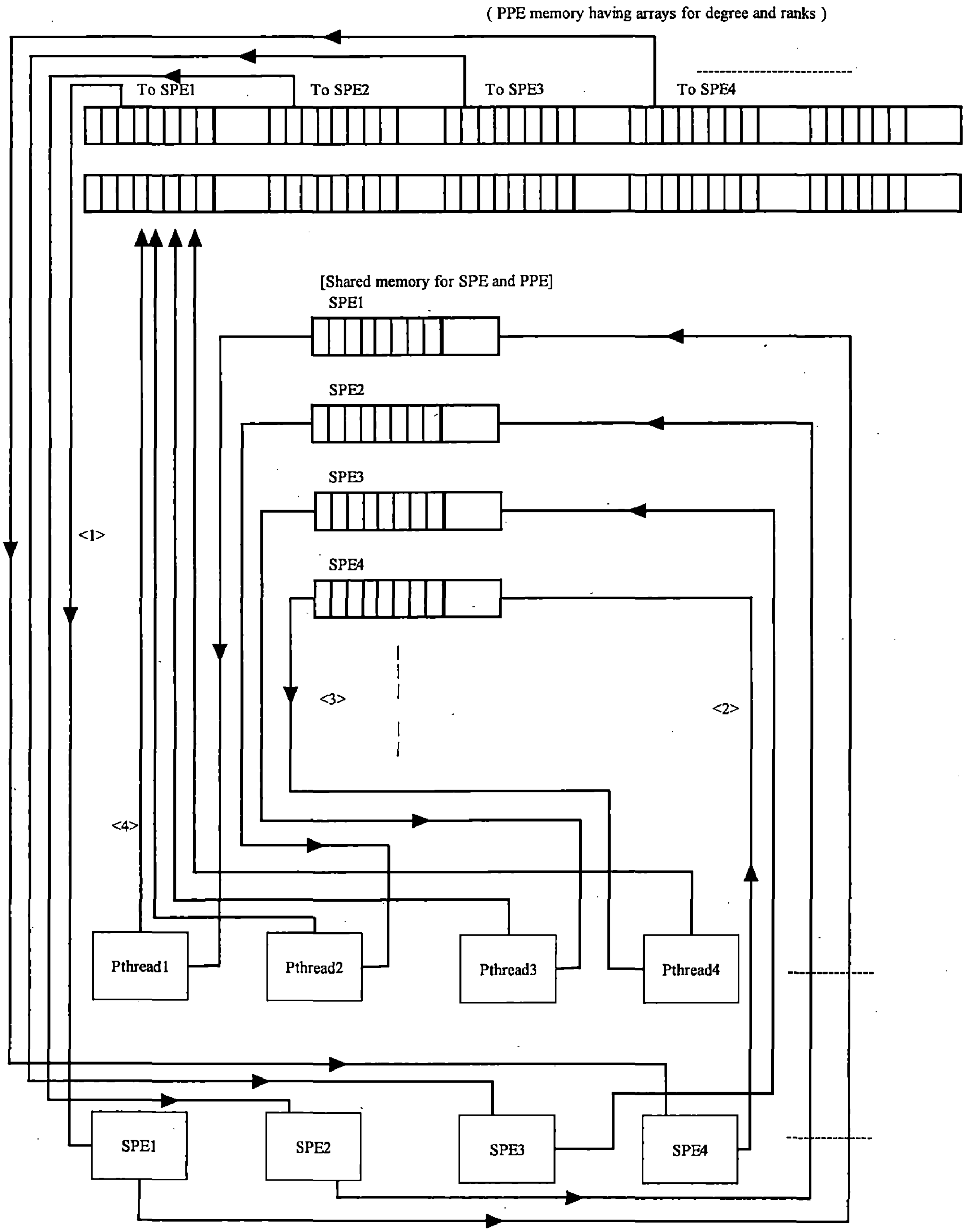

Figure5.6: Data flow in memory between PPE and SPE. <1> <2> <3> <4> are paths of flow

Table 5.1: comparison of timing of PageRank implementation on Xeon and Cell BE processor

Iteration number	Time(seconds) on Xeon	Time (seconds) on Cell BE Processor
1	.390146	.38
2	.823685	.691557
4	1.551139	1.379249
8	3.22564	3.03875
16	6.314917	5.80895
32	12.43	11.61175

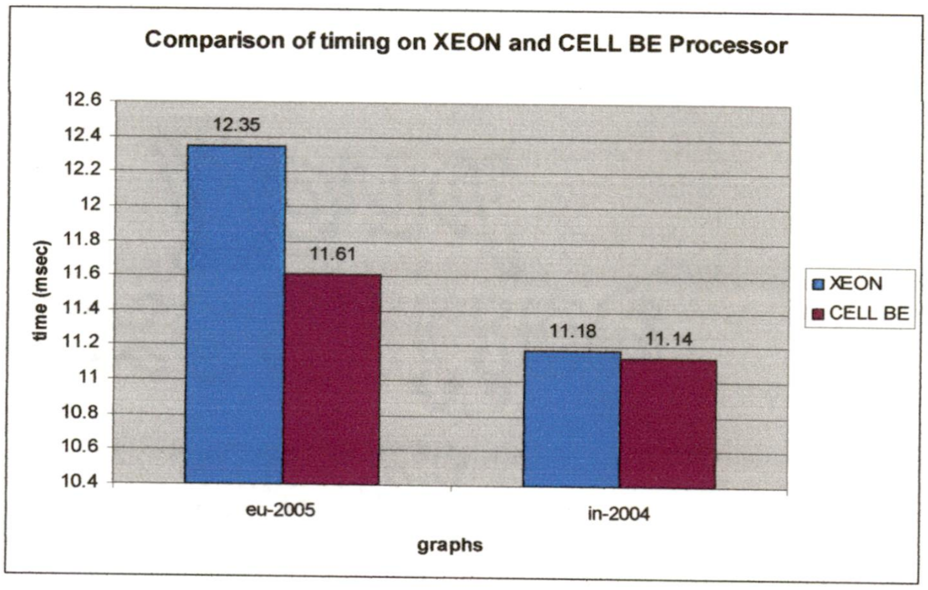

Figure 5.7: Comparison of time between XEON and Cell BE processor

CHAPTER 6

PAGERANK ON CUDA

GPU (Graphics Processing Unit) is a highly parallel, multi-threaded and many-core processor with tremendous computing power. CUDA provides a programming environment to the programmer to utilize GPU in a simpler fashion. Execution of program starts with CPU. GPU is used by the CPU to execute parallel threads. In this chapter we present implementation of PageRank Algorithm on CUDA.

6.1 Issues of implementing PageRank on CUDA

1. Architecture of CUDA requires threads of same code path to be running in parallel on a multiprocessor. Execution on CUDA takes place in form of warps. Warps are 32 thread units that are executed on a multiprocessor. CUDA stops all divergent threads within a warp. So if any branch statement is encountered the amount of parallelization gets reduced as divergent threads are no longer running in parallel. Real world scenario web graphs have varying number of in-degrees for nodes. Now processing in the PageRank algorithm works on every node. We have initiated one thread for every node. Since these nodes have varying number of in-degrees the amount of iterations performed by every thread is different which creates a number of divergent threads.
2. Memory synchronization constructs for global memory are not available in CUDA.
3. CUDA does not provide atomic statements for floating point values while PageRank works totally on floating point values.
4. PageRank calculation performs read operations from a wide range of memory area with very less localization of reference. This generates a lot of page faults.

6.2 Design and Implementation on CUDA

CUDA provides the facility of generating tens of thousands of threads at a time with no generation time. These threads can run on multiprocessors of GPU in parallel. CUDA threads run on different multiprocessors simultaneously, therefore there should not be any data dependency among threads.

6.2.1 Design 1

PageRank algorithm calculates rank of nodes such that rank of a particular node depends on rank of other nodes which have a link with that node. There is no data dependency present between any two nodes of web graph. We create threads equal to the number of nodes. CUDA's limitations disallow this direct approach of solving PageRank algorithm efficiently. The limitations of CUDA with respect to PageRank are as follows:

1. Number of nodes in a web graph is very large (of the order of billion) and such a large number of threads cannot be generated on GPU (limited by hardware).
2. Rank of a node may depend upon any number of nodes; therefore a loop to calculate rank of different nodes runs for different number of iterations and hence causing different code paths for threads. This means a large number of threads diverge and they cannot run in parallel.

The above stated problems were further eliminated in the following manner.

1. Threads corresponding to all nodes should not be spawned at the same time; therefore threads are created in multiple passes in a loop.
2. To avoid the problem of divergence an extra level of parallelism is added. Instead of calculating rank of a node on a single thread, rank of one node is calculated by as many threads as the in degree of node. We create threads equal to the total in-degree of all the nodes. For each node, threads equal to its in-degree calculate parallely their respective shares of rank and add that share to the rank of node (which is kept 0 initially). This causes threads to have equal amount of work to be done and hence the code path is same for each thread. This approach requires the rank of a node modified by several threads running in parallel which causes the problem of synchronization among the threads. CUDA does not provide synchronization tools for global memory. Though it provides atomic operations (means once a thread is using a particular memory location no other thread can use that location) for integers only but PageRank requires floating point values. Thus this approach could not be used.

6.2.2 Design 2

The main problem with design 1 is that it hinders the performance because of the variable loop length of each thread. In order to avoid this problem, we run a fixed length loop (say N) on GPU for all threads. Value of N depends upon the web graph to be worked upon. Ideally we want most of the computations to be performed on GPU. GPU prefers threads of similar amount of computation. We select N in such a way that more GPU threads are similar in computation. The idea is to run GPU and CPU parallely such that while GPU is running loop of length N for all threads, CPU calculates partial rank of those nodes which have in-degree more than N by running a loop from N to in-degree of the node.

Figure 6.1 shows an example of small web graph. Value of N is kept 4. Rank of all nodes is calculated with 4 (or less) source nodes at GPU. Host at the same time calculates partial rank of nodes 4, 5, 7, 8, 11 with those source nodes participating that have index more than N (= 4) . For example for node 4 partial ranks with source nodes 2, 5, 6, 7 is calculated on GPU and partial rank with source nodes 8, 9, 12 is calculated at Host.

		GPU				CPU						
Node	In-deg	Source nodes			N = 4							
1	2	6	7									
2	3	2	4	9								
3	1	1										
4	7	2	5	6	7	8	9	12				
5	11	1	2	3	4	5	6	7	9	11	14	16
6	3	6	9	12								
7	5	1	4	7	9	11						
8	6	2	5	6	8	11	12					
9	5	3	7	9	11	12						
⋮												

Figure 6.1: An example showing the division of work between GPU and CPU

The work of PageRank calculation is divided onto GPU and Host in such a way that when GPU is calculating partial rank of all nodes with the help of N (or less) source nodes, Host at same time calculates partial rank of all nodes with remaining source nodes (other than N nodes if any). GPU calculates the share of N source nodes which point to a destination node by running loop N times for each thread. Host calculates share of rest of the nodes by running a loop from N to their corresponding in degree. In this way host and GPU calculate partial rank of nodes. These partial ranks are then added and used for next iteration.

6.2.3 Implementation Details

This section presents implementation details of design 2 described in previous section. Program for calculating PageRank consists of mainly two parts, host and kernel. Figure 6.2 shows the overall control flow of PageRank calculation on CUDA. Execution of PageRank algorithm starts with host program. Host program reads the web graph and copies it to the GPU's memory. Host invokes the kernel to be run on GPU for calculating partial rank with N source nodes. GPU starts processing. Since kernel calls are non-blocking for host, therefore Host also starts rank calculation of nodes having in-degree more than N. As soon as calculation on both GPU and Host are finished, partial rank of all nodes from Host is brought into GPU memory and added with partial rank calculated at GPU. Thus the new rank of all nodes is calculated and input vectors for next iteration is updated both at GPU and Host.

6.3 Results

The approach used assigns only that much amount of task on GPU that reduces thread divergence and rest of the task is performed on the host at the same time when kernel is being executed on GPU. Though the approach ensures better performance yet there is always an issue of exact amount of work to be divided between GPU and CPU. Speed up obtained will not be same for all the web graphs. Experiments are performed for two graphs. Graph EU- 2005 contains 862664 nodes and 19235140 links and graph CNR-2000 contains 325557 nodes and 3216152 links. GPU used for experiments is GTX 280.

This GPU consists of 1 GB of memory and 30 multiprocessors. Each multiprocessor has 8 scalar processors.

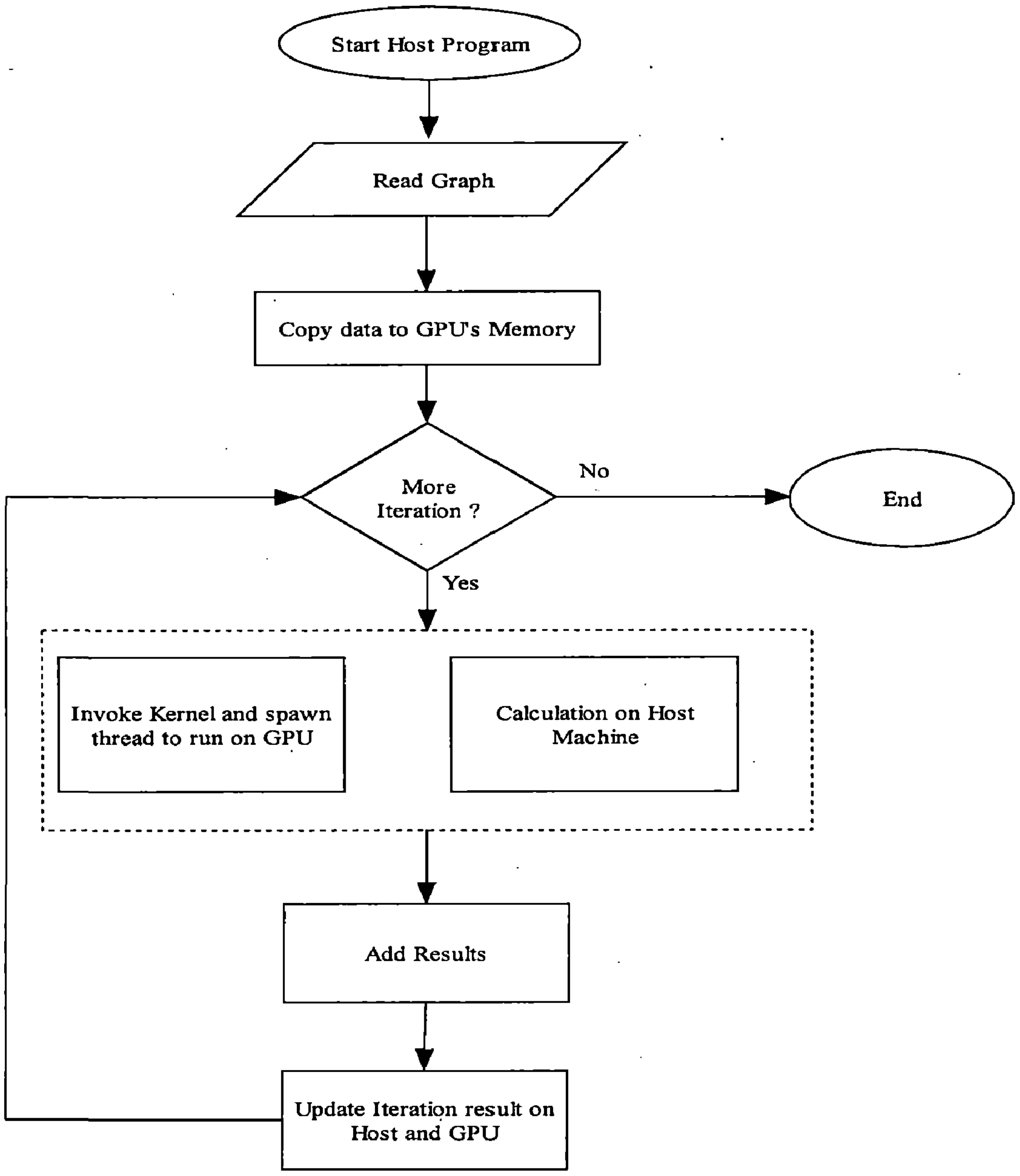
Start Host Program
Read Graph
Copy data to GPU's Memory
More Iteration ?
No
End
Yes
Invoke Kernel and spawn thread to run on GPU
Calculation on Host Machine
Add Results
Update Iteration result on Host and GPU

Figure 6.2: Program flow of PageRank calculation on CUDA

Figure 6.3 shows a comparison of timing on Xeon dual core 3.0 Ghz and CUDA. It can be observed that for graph EU-2005 a speed up of nearly 2.8 times is obtained on CUDA over Xeon dual core 3.0 while for CNR-2000 speed up is nearly 2.2.

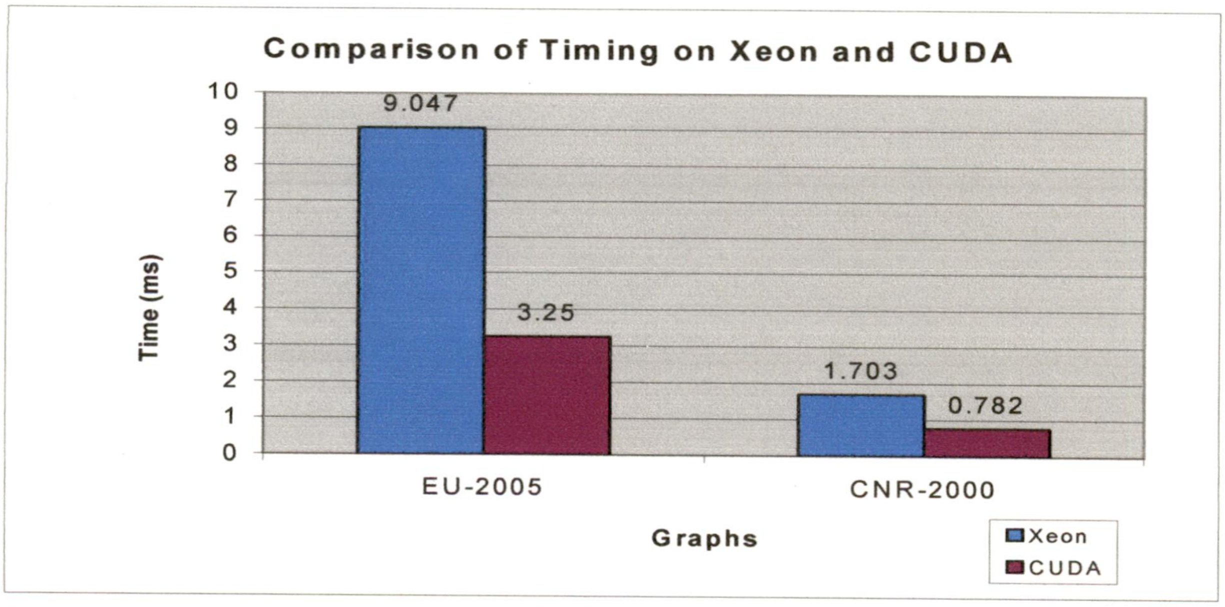

Figure 6.3: Comparison of timing on Xeon and CUDA

We compare the implementation on CUDA with Cell BE for graph EU-2005. It is found that CUDA performs much better over Cell BE. Figure 6.4 shows a comparison of timing on Xeon dual core 2.0 Ghz, Cell BE and CUDA. It shows that implementation on CUDA is 2.6 times faster than implementation on Cell BE.

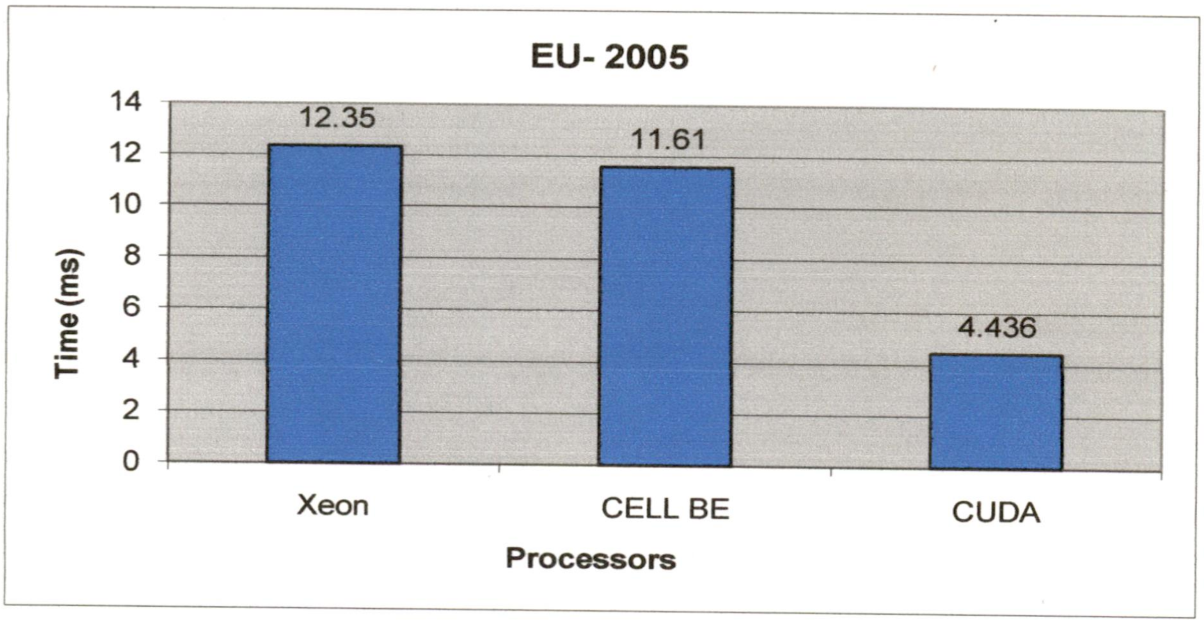

Figure 6.4: Comparison of timing on Xeon and CUDA and Cell BE processors

CHAPTER 7

CONCLUSION AND FUTURE WORK

In this dissertation, we have implemented two algorithms on multicore processor. Firstly, we identified the major issues which may encounter during implementation of a general question answering system on Cell BE processor and then proposed solutions to them. We then implemented the indexing component of a domain specific QA system on Cell BE processor. While there were a number of major challenges involved and some of them were only partially dealt with, we still managed to obtain reasonable speedup. This suggests that Cell BE processor holds considerable amount of potential for information retrieval applications.

Secondly, in PageRank we indentified the major issues of porting PageRank algorithm on Cell BE Processor. Possible solutions to these issues were drawn. Previous implementation of PageRank on Cell BE resulted in poor performance because of the high data transfer operation between PPE and SPE. A new approach is implemented which reduces the data transfer between PPE and SPE drastically and leads to a better performance. We also presented issues of porting PageRank algorithm on CUDA followed by its implementation on CUDA. It was found that implementation of PageRank on CUDA is performing much better than on Cell BE.

In future, one can try implementing NLP toolkits like MMTX on the Cell BE processor so that they will lead better compatibility while used with application designed on Cell BE processor. Information retrieval applications can also be implemented so that a complete Question Answering system can be ported on Cell BE. We showed in our results in chapter 4 that if documents to be indexed are of same size then speed up is high and also no fluctuation appears in results. This high speed up is obtained because of equal workload on all SPEs. In real scenario where documents will vary greatly, performance will deteriorate, thus this opens a scope for further improvement in proposed work. Some new approach for balancing the work load between the SPEs can be devised.

As far as future work for PageRank algorithm is concerned, a better performance in current implementation can be found by dividing the graph in small blocks and then determine the value of N (number of iterations to be run on GPU for a thread) for each block. Performance of PageRank algorithm on multicore processor can be improved by analyzing the web graph in detail and preprocess it according to the restrictions and features of multicore architecture like sorting the web graph on the basis of in-degree of nodes.

REFERENCES

[1] Question Answering System: http://en.wikipedia.org/wiki/Question_answering
[2] National Library of Medicine, http://www.nlm.nih.gov/news/pubmed_15_mill.html
[3] P. Jacquemart and P. Zweigenbaum, “Towards a medical question-answering system: a feasibility study,” in Proceeding of Medical Informatics Europe (MIE ’03), P. L. Beux and R. Baud, Eds., vol. 95 of Studies in Health Technology and Informatics, IOS Press, San Palo, Calif, USA, 2003, pages: 463–468
[4] S. Schultz, M. Honeck, and H. Hahn, “Biomedical text retrieval in languages with complexmorphology,” in Proceedings of the Workshop on Natural Language Processing in the Biomedical domain, Philadelphia, Pa, USA, July 2002, pages: 61–68.
[5] Parikshit Sondhi , Purushottam Raj , V. Vinod Kumar, and Ankush Mittal, “Question processing and clustering in INDOC: a biomedical question answering system,” EURASIP Journal on Bioinformatics and Systems Biology, v.2007 n.3, July 2007, pages: 1-7.
[6] MEDLINE to PubMed and Beyond, http://www.nlm.nih.gov/bsd/historypresentation.html
[7] S. Brin and L. Page, “The Anatomy of a Large-Scale Hypertextual Web Search Engine,” in Proceddings of the 7th International world wide web conference, Brisbane, Australia, April 1998, pages: 107-117
[8] L. Page, S. Brin, R. Motwani, and T. Winograd. “The PageRank citation ranking: Bringing order to the web,” in Stanford Digital Library Working Paper,1998.
[9] PageRank Google's Original Sin: http://www.google-watch.org/pagerank.html
[10] T. H. Haveliwala, “Topic Sensitive PageRank,” IEEE Transactions on Knowledge and Data Engineering,Volume 15, Issue 4, July-Aug. 2003 pages: 784 - 796
[11] Multicore processor: http://searchdatacenter.techtarget.com/sDefinition/0,,sid80_gci 1015740,00.html
[12] IBM alphaWorks Cell BE SDK: http://www.alphaworks.ibm.com/topics/cell
[13] Cell Broadband Engine – An introduction. Cell Programming Workshop, IBM System and Technology Group, April 14-18, 2007.

[14] Cuda Programming Guide 2.0,
http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf, July 6, 2008

[15]CUDA: http://en.wikipedia.org/wiki/CUDA

[16] Y. Niu, G. Hirst, G. McArthur, and P. Rodriguez-Gianolli, "Answering clinical questions with role identification," in Proceedings of 41st annual meeting of the Association for Computational Linguistics, Workshop on Natural Language Processing in Biomedicine, Sapporo Japan, 2003 pages: 73-80

[17] L. A. Barroso, J. Dean, and U. Hölzle, "Web Search for a Planet: The Google Cluster Architecture," IEEE Micro, vol. 23, number 2, Mar/Apr, 2003, pages: 22-28.

[18] MetaMap Portal: http://mmtx.nlm.nih.gov/

[19] Y.Y. Chen , Q. Gan, and T. Suel, "I/O-efficient techniques for computing PageRank," in Proceedings of the eleventh international conference on Information and knowledge management, McLean, Virginia, USA 2002 Pages: 549 - 557

[20] S. D. Kamvar, T.H. Haveliwala, C.D. Manning, and G.H. Golub, "Exploiting the block Structure of the Web for Computing PageRank," Technical Report CSSM-03-02, Computer Science Department, Stanford University, 2003.

[21] B. Manaskasemsak and A. Rungsawang, "Parallel PageRank Computation on gigabit PC Cluster," in Proceedings of 18th International Conference on Advanced Information Networking and Applications AINA, Fukuoka Japan, March 2004, Vol. 1 pages: 273-277.

[22] B. Manaskasemsak and A. Rungsawang, "An efficient partition-based parallel PageRank algorithm," in the proceedings of 11th International Conference on Parallel and Distributed Systems, Fuduoka, Japan, July 2005, Vol. 1, pages: 257-263

[23] G. Buehrer, S. Parthasarathy, and M. Goyder, "Data mining on the Cell broadband engine," in the Proceedings of the 22nd annual international conference on Supercomputing, Island of Kos, Greece, June 2008, pages: 26-35.

[24] National Library of Medicines, Unified Medical Language System: http://www.nlm.nih.gov/research/umls

[25] International Classification of Diseases: http://www.cdc.gov/nchs/about/otheract/icd9/abticd9.htm

PUBLICATIONS

1 T. Kumar, P. Sondhi, and A. Mittal, "Parallelization Issues of a Domain Specific Question Answering System on Cell BE Processor," in International Conference on Contemporary Computing, August 2009. [Accepted in May 2009, to be published in the series of Communications in Computer and Information Science, Springer]