A Project Report on

PARALLELIZATION OF SUFFIX TREE CONSTRUCTION ON IBM CELL BE AND ITS APPLICATIONS

For partial fulfillment of Bachelor of Technology in Computer Science and Engineering

Under Guidance of:
Dr. Ankush Mittal
Associate Professor,
Department of Electronics & Computer Engineering
Indian Institute of Technology Roorkee

Submitted By:
Piyush Behre (Enrollment No.040818)
Rahul Goyal (Enrollment No.040821)
Vivek Kumar Chaurasiya (Enrollment No.040832)

B.Tech. (Final Year)
(Computer Science and Engineering)
Department of Electronics & Computer Engineering
Indian Institute of Technology Roorkee

May, 2008

`

CANDIDATE'S DECLARATION

We hereby certify that the work being presented in this project entitled, "Parallelization of Suffix Tree Construction on IBM Cell BE and its applications", for the award of the degree of Bachelor 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 our original work carried under the guidance of Dr. Ankush Mittal.

We have not submitted the matter embodied in this project for the award of any other degree.

(Piyush Behre)
Enrolment no. 040818

(Rahul Goyal)
Enrolment no. 040821

(Vivek Kumar Chaurasiya)
Enrolment no. 040832

`

CERTIFICATE

This is to certify that Piyush Behre (Enrollment No.040818), Rahul Goyal (Enrollment No.040821), Vivek Kumar Chaurasiya (Enrollment No.040832), students of B.Tech IV year, Computer Science and Engineering, Department of Electronics & Computer Engineering, Indian Institute of Technology, Roorkee have successfully worked on a research project entitled "**Parallelization of Suffix Tree Construction on IBM Cell BE and its applications**" as a part of the final year B.Tech project. This project is a result of an original and sincere effort made by the above candidates. All the statements made by the candidates are correct to the best of our knowledge.

Date: May 2008

……..
Dr. Ankush Mittal
Associate Professor,
Department of Electronics & Computer Engineering,
Indian Institute of Technology Roorkee

`

ACKNOWLEDGEMENT

We express our heartfelt gratitude to Dr. Ankush Mittal, for giving us such a novel idea and opportunity to work on this project. Without his able guidance and encouragement, this project would not have been possible. His valuable feedbacks, comments and suggestions were helpful in maintaining a positive approach throughout the entire project. We thank Dr. Ravi Gupta, who helped us understand the Cell Design & how it can be exploited in our project. Special thanks to Dr. Manoj Mishra for his lectures on Advanced Computer Architecture.

Piyush Behre
Rahul Goyal
Vivek Kumar Chaurasiya

B.Tech. (IV Year, Computer Science and Engineering)
Department of Electronics & Computer Engineering
Indian Institute of Technology Roorkee

`

ABSTRACT

A well-known challenge in Bioinformatics is to process large strings of nucleotides and searching for matching patterns. The size of the DNA is huge and a brute force implementation can take up a lot of time and resources. This problem can be tackled by using multi-processor architecture like Cell BE and by use of novice data structure like suffix trees which improves upon time and space complexity.

In this project, we have implemented some improvements to the design of construction of suffix trees to make use of parallel processing power of the synergistic processor elements of the cell processor. We have suggested and implemented a dual parallel construction mode to make use of all eight processors simultaneously to make maximum utilization of the processors even in case of bioinformatics applications where for one construction only four SPEs are generally active. Then we have developed all major applications namely string search, largest common substring, largest repeated substring and largest palindrome search, some of which use the generalized suffix tree version of the data structure. At the end of the project, we have done some extensive time analysis for these applications and shown the various speedups achieved.

We have implemented all the classes and functions in C/C++. For testing and performance analysis, we have used the IBM cell simulator on linux environment. IBM cell simulator has its own programming models, libraries and communication mechanisms which have been used aptly throughout the project.

`

TABLE OF CONTENTS

`

`

List of Figures

`

List of Tables

`

Chapter 1

INTRODUCTION

1.1 Problem Statement

One of the most important problems in bioinformatics is that of pattern matching. The sequences of DNA or proteins can be very large in size and searching for subsequences becomes a problem if normal matching algorithms are used. This has a worst case complexity of O(nm). Significant improvements in efficiency are observed if one uses a suffix tree where matching can be performed with complexity O(m). Further improvements may be obtained if we use the parallel processing power of the Synergistic Processor Elements (SPEs) in Cell Processor. Our project aims at a successful implementation to obtain such speed-up.

1.2 Bioinformatics

Bioinformatics involves the manipulation, searching, and data mining of DNA sequence data. The development of techniques to store and search DNA sequences have led to widely-applied advances in computer science, especially string searching algorithms, machine learning and database theory. String searching or matching algorithms, which find an occurrence of a sequence of letters inside a larger sequence of letters, were developed to search for specific sequences of nucleotides. In other applications such as text editors, even simple algorithms for this problem usually suffice, but DNA sequences cause these algorithms to exhibit near-worst-case behavior due to their small number of distinct characters. The related problem of sequence alignment aims to identify homologous sequences and locate the specific mutations that make them distinct. These techniques, especially multiple sequence alignment, are used in studying phylogenetic relationships and protein function.

Data sets representing entire genomes' worth of DNA sequences, such as those produced by the Human Genome Project, are difficult to use without annotations, which label the locations of genes and regulatory elements on each chromosome. Regions of DNA

`

sequence that have the characteristic patterns associated with protein- or RNA-coding genes can be identified by gene finding algorithms, which allow researchers to predict the presence of particular gene products in an organism even before they have been isolated experimentally.

1.3 Motivation

Any string of length m can be degenerated into m suffixes, and these suffixes can be stored in a suffix tree. Creating this structure requires time O(m) and searching for a pattern in it requires time O(n), where n is the length of the pattern. These two properties make the suffix tree an appealing structure for a diverse range of bioinformatics applications including: multiple genome; selection of signature oligonucleotides for DNA arrays and identification of sequence repeats. There can be significant improvements if we could identify the steps which can be parallelized and port the whole algorithm on a multiprocessor. This is the main motivation behind the project.

In IBM Cell Broadband Engine there is one central processor (PPE) and eight secondary processors (SPEs). PPE manages the load distribution among the SPEs which are waiting for instructions from the PPE. In a typical case, each SPE can handle one branch of the Suffix Tree and after construction step it can wait for search or match type queries. PPE can provide the interface for clients to put up queries.

1.4 Objectives

The primary objectives of the project are as follows:

- Study of the cell broadband engine architecture.
- Implementation of suffix tree construction algorithm on uniprocessor with linear time complexity.
- Port the linear time algorithm into Cell BE to gain speed-up using parallel processing power of the SPEs.
- Implement pipelined string search application.
- Parallel double suffix tree construction for biotech applications where strings only contain 4 characters: A, G, C and T.

`

- Modify algorithm for generalized suffix tree Construction on Cell BE.
- Develop longest palindrome search, longest common substring search and longest repeated substring search application.
- Test the applications and calculate the speed-ups achieved.

1.5 Organization of the Report

The report is organized into 5 chapters apart from introduction. The contents of the various chapters are:

Cell Processor Architecture: This chapter briefly discusses the structural components of the PPE and the SPE. It also gives a brief overview of communication mechanisms available and the cell simulator.

Data Structures and Algorithms: This chapter discusses the classical definition of suffix trees along with construction algorithm with $O(n^2)$ time complexity. It them discusses methods to improve the complexity first to linear time and then improving further by porting it on Cell BE.

Implementation Details: This chapter describes the main classes involved in the implementation and enlists the problems faced in the project.

Applications and Testing: In this chapter, we cover the four main applications along with the speedup results achieved. A brief overview of generalized suffix trees is also given as it is central to many applications.

Conclusion and Results: This chapter enlists all the results of the project and also highlights the area where further work needs to be done.

`

Chapter 2

CELL BE ARCHITECTURE

2.1 Why CELL BE?

Although the cell broadband engine is initially intended for application in game consoles and media-rich consumer-electronics devices such as high-definition televisions, the architecture and the cell broadband engine implementation have been designed to enable fundamental advances in processor performance.

Cell processor inherits its characteristics from the popular 64 bit Power PC architecture, the result of collaboration between Sony, Toshiba, and IBM.

2.2 Benefits – Challenge the limits. Walls

The Cell broadband engine overcomes three important limiters of contemporary microprocessor performance—**power use**, **memory use**, and **processor frequency**.

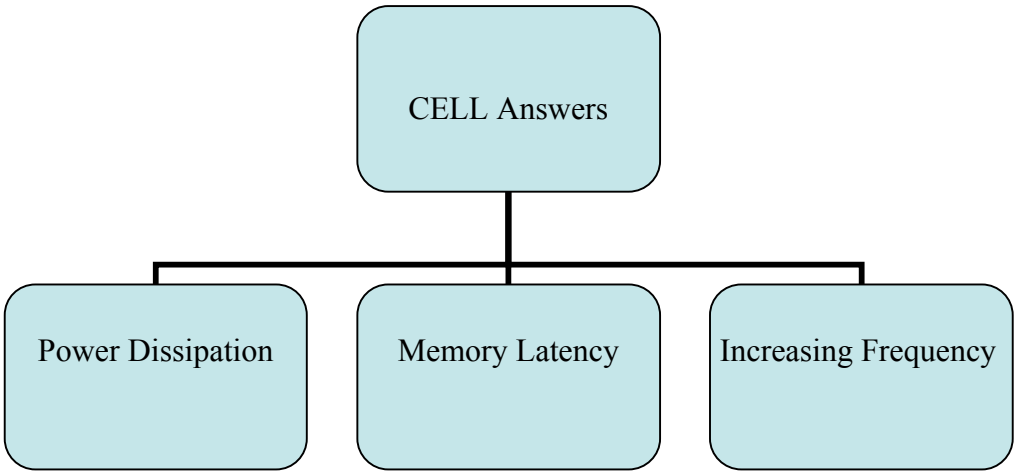


Figure 1 Three Walls

`

2.3 Comparison of Cell with Conventional Microprocessors:

2.3.1 Power Wall

Microprocessor performance is limited by achievable power dissipation rather than by the number of available integrated-circuit resources (transistors and wires). Thus, the only way to significantly increase the performance of microprocessors is to improve power efficiency at about the same rate as the performance increase.

One way to increase power efficiency is to differentiate between control-intensive codes and compute intensive code. Cell makes use of this concept by using two types of Processors:
PPE (PowerPC Processor Element) – Control Intensive applications
SPE (Synergistic Processor Element) - Compute Intensive applications

2.3.2 Memory Wall

On multi-gigahertz symmetric multiprocessors—even those with integrated memory controllers latency to RAM is currently approaching 1,000 cycles. So, program performance is badly affected by the high latency.

The Cell Broadband Engine's SPEs use **two mechanisms** to deal with long main-memory latencies:

1. A 3-level memory structure
    a. Main storage
    b. Local Stores (in each SPE)
    c. Large register files (in each SPE)

2. Asynchronous DMA transfers between main storage and local stores.

`

2.3.3 Frequency Wall

Conventional processors require increasingly deeper instruction pipelines to achieve higher operating frequencies. This technique has reached a point of diminishing returns—and even negative returns if power is taken into account.

By specializing the PPE and the SPEs for control and compute-intensive tasks, respectively, the Cell Broadband Engine Architecture, on which allows both the PPE and the SPEs to be designed for high frequency without excessive overhead.

2.4 Cell Architecture

The Cell BE comprises of a PPE (POWER Processing Element) optimized for control intensive applications and 8 SPEs (Synergistic Processing Elements) optimized for computationally intensive applications. PPE is the main dual-threaded processor containing a 64-bit PowerPC Architecture RISC core. It runs the top level control thread of the application.

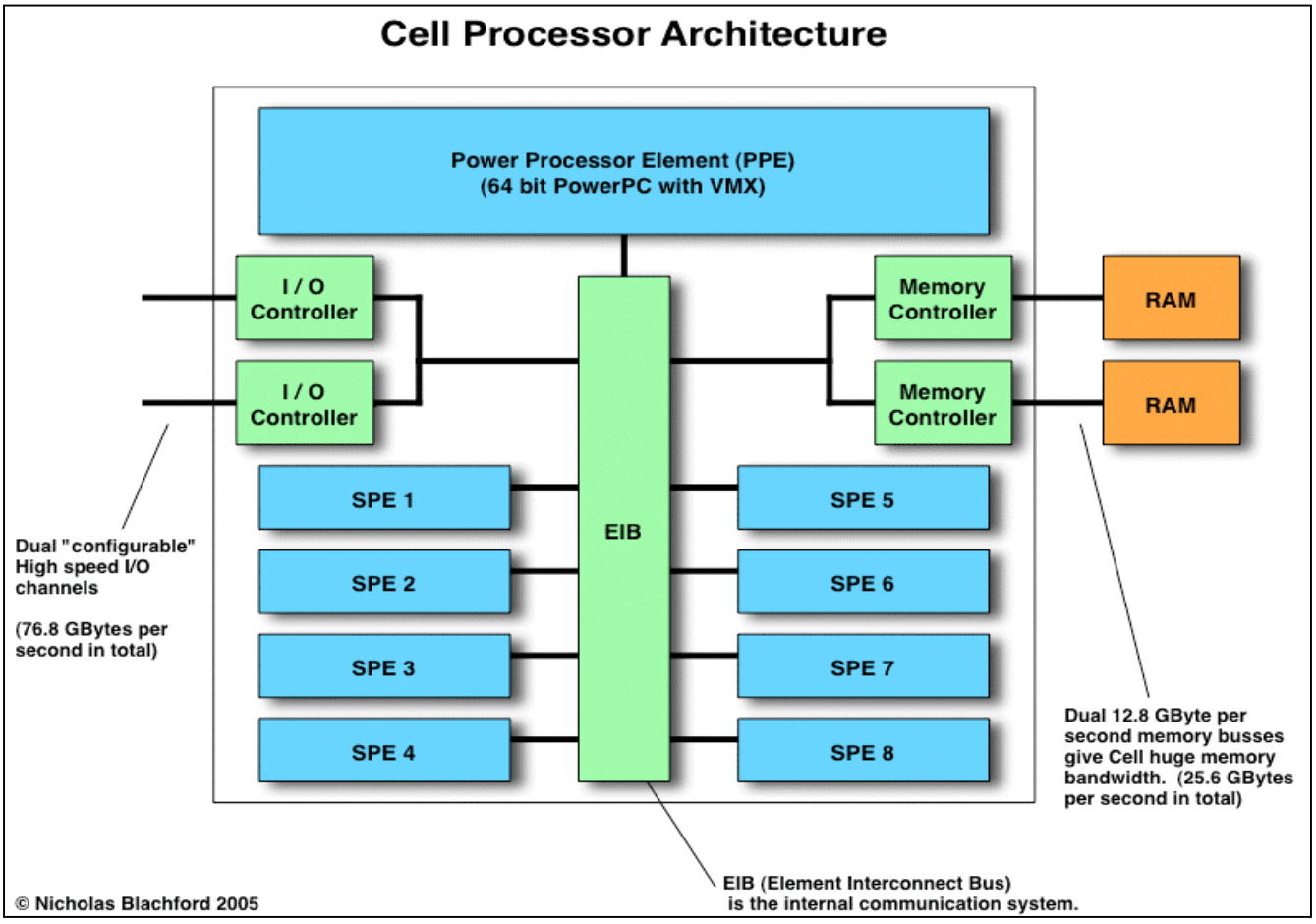


Figure 2 Cell BE Architecture [3]

`

The Cell Broadband Engine consists of **nine** processors on a **single chip**, all connected to each other and to external devices by a high-bandwidth, memory-coherent bus. *Figure above* shows a block diagram of the Cell Broadband Engine. The main blocks include:

• ***PowerPC Processor Element (PPE)*** **—** The PPE is the main processor. It contains a 64-bit PowerPC Architecture reduced instruction set computer (RISC) core with a traditional virtual memory subsystem. It runs an operating system, manages system resources, and is intended primarily for control processing, including the allocation and management of SPE threads. It can run legacy PowerPC Architecture software and performs well executing system- control code. It supports both the PowerPC instruction set and the Vector/SIMD Multimedia Extension instruction set.

• ***Synergistic Processor Elements (SPE)*** **—** The eight SPEs are SIMD processors optimized for data-rich operations allocated to them by the PPE. Each of these identical elements contains a RISC core, software-controlled local store (256-KB) for instructions and data, and a large (128-bit, 128-entry) unified register file. The SPEs support a special SIMD instruction set, and they rely on asynchronous DMA transfers to move data and instructions between main storage (the effective-address space that includes main memory) and their local stores. SPE DMA transfers access main storage using PowerPC effective addresses. As on the PPE, address translation is governed by PowerPC Architecture segment and page tables. The SPEs are not intended to run an operating system.

• ***Element Interconnect Bus (EIB)*****—**The PPE and SPEs communicate coherently with each other and with main storage and I/O through the EIB. The EIB is a 4-ring structure (two clockwise and two counterclockwise) for data, and a tree structure for commands. The EIB's internal bandwidth is 96 bytes per cycle, and it can support more than 100 outstanding DMA memory requests between main storage and the SPEs.

`

EIB has two external interfaces:

Memory Interface Controller (MIC)—The MIC provides the interface between the EIB and main storage. It supports two Rambus Extreme Data Rate (XDR) I/O (XIO) memory channels and memory accesses on each channel of 1-8, 16, 32, 64, or 128 bytes.

Cell Broadband Engine Interface (BEI)—The BEI manages data transfers between the EIB and I/O devices. It provides address translation, command processing, an internal interrupt controller, and bus interfacing.

The channel can be configured to support either non-coherent transfers or coherent transfers that extend the logical EIB to another compatible external device, such as another Cell Broadband Engine.

`

2.5 CELL BE: Programming Models

2.5.1 Function Offload Model

In this model, SPEs are used as accelerators for performance critical procedures. Main application runs on the PPE and calls selected procedures to run on the SPEs. This model is sometimes referred to as 'Remote Procedure Call (RPC) model'.

2.5.2 Device Extension Model

This model is a special case of the Function Offload Model. In this case SPEs act like I/O devices. Mailboxes are used as command and response FIFOs between the PPE and the SPEs. All I/O devices are memory-mapped so the SPEs can interact with them.

2.5.3 Computation Acceleration Model

This is basically an SPE centric model. Here most of the computation intensive sections of the application run on SPEs. PPE acts as control and system service facility. Great speed-ups can be achieved when multiple SPEs work in parallel.

2.5.4 Streaming Model

Each SPE, in a pipeline, computes data that streams through. Here PPE acts as stream controller and the eight SPEs act as stream data processors. PPE and SPEs support message passing between the PPE, the processing SPE and other SPEs.

2.5.5 Shared Memory Multiprocessor Model

SPEs and PPE fully interoperate in a cache-coherent shared memory multiprocessor model. Shared Memory Loads are replaced by DMA from shared memory to LS followed by load from LS to register file. Then DMA operations use EA common to PPE and the SPEs. Shared Memory stores are replaced by store from register file to LS followed by DMA from LS to shared memory.

Our implementation follows two of the above models namely, Computation Acceleration Model and Streaming Model.

`

2.6 PPE: PowerPC Processor Element

The PPE is a 64 bit, "Power Architecture" capable of running POWER or PowerPC binaries as well as Extended Vector Scalar Unit (VSU). The PPE follows in-order execution, dual threaded and dual issue architecture.

2.6.1 Structure

The PPE is a General Purpose, dual threaded, 64-bit RISC processor with 32 KB L1 cache and 512 KB unified L2 cache. It is intended for central processing, running operating system, managing system resources and SPEs' threads. It can run 32-bit as well as 64-bit operating system and applications. PPE accesses main memory with Load and Store instructions that move data between main memory and a private register file the contents of which may be cached. There are six execution units and it can load 32 bytes and store 16 bytes per processor cycle.

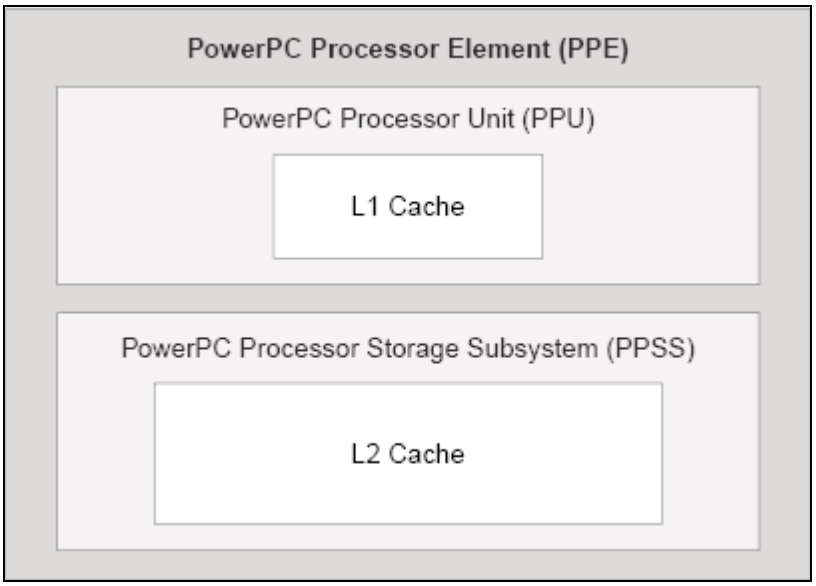


Figure 3 PPE Components [3]

2.6.2 Components

The PPE consists of two main units namely, Power Processor Unit (PPU) and Power Processor Storage Subsystem (PPSS).

`

2.6.2.1 PPU (Power Processor Unit):

The PPU executes the *PowerPC Architecture* instruction set and the Vector/SIMD Multimedia Extension instructions. It has following main functional units:

Instruction Unit (IU)—The IU performs the instruction-fetch, decode, dispatch, issue, branch, and completion portions of execution. It contains the L1 instruction cache.

Load and Store Unit (LSU)—The LSU performs all data accesses, including execution of load and store instructions. It contains the L1 data cache.

Vector/Scalar Unit (VSU)—The VSU includes a Floating-Point Unit (FPU) and a 128-bit Vector/ SIMD Multimedia Extension Unit (VXU), which together execute floating-point and Vector/ SIMD Multimedia Extension instructions.

Fixed-Point Unit (FXU)—The FXU executes fixed-point (integer) operations.

Memory Management Unit (MMU)—The MMU manages address translation for all memory accesses.

The PPU supports two simultaneous threads of execution and can be viewed as a 2-way multiprocessor with shared dataflow. This appears to software as two independent processing units. The state for each thread is duplicated, including all architected and special-purpose registers except those that deal with system-level resources, such as logical partitions, memory, and thread-control.

`

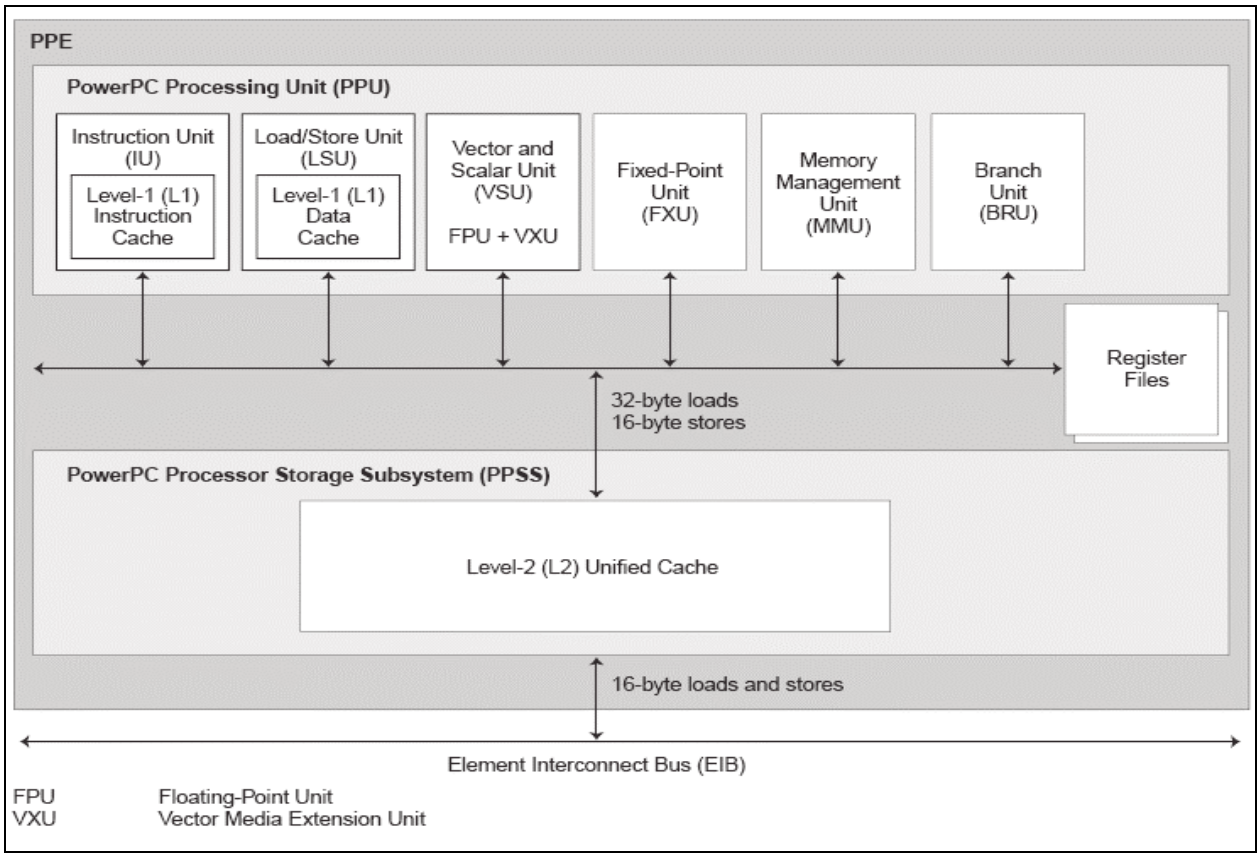


Figure 4 PPE Functional Units [3]

2.6.2.2 PPSS (Power Processor Storage Subsystem)

The PPSS handles memory requests from the PPE and external requests to the PPE from other processors or I/O devices. It includes a unified 512-KB level 2 (L2) instructions and data cache, various queues, and a bus interface unit that handles bus arbitration and pacing on the EIB.

`

2.7 SPE (Synergistic Processor Element)

2.7.1 Structure

An SPE is a self contained vector processor (SIMD) which acts as a co-processor which follows in-order to minimize circuitry to save power. Also it is statically scheduled where compiler plays a big role. Also as there is no dynamic prediction hardware, it relies on compiler generated hints for prediction. Each SPE consists of 128 x 128 register, a Local Store (SRAM) of 256 KB, DMA unit, instruction-control unit, a Load and Store unit, two fixed-point units, a floating-point unit channel and DMA interface.

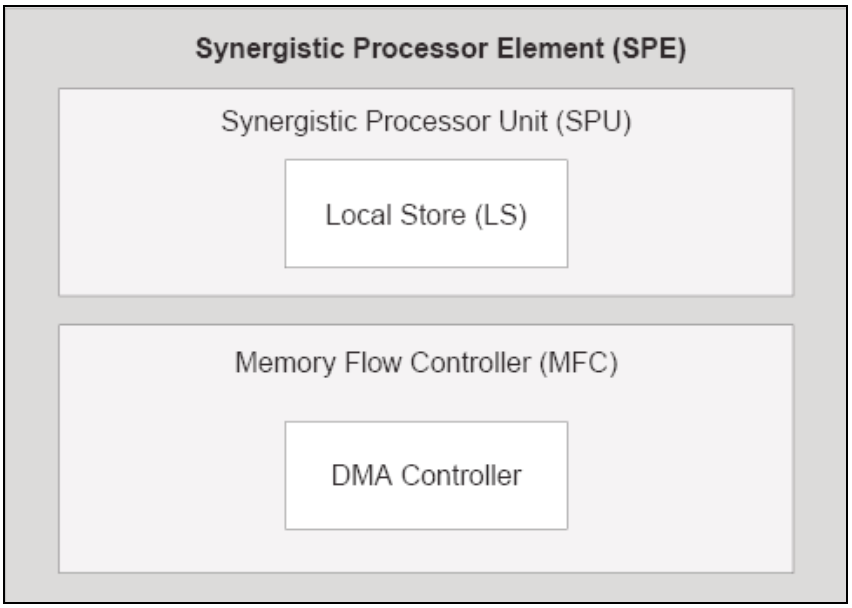


Figure 5 SPE Components [3]

2.7.2 Components

The SPE consists of two main units namely, Synergistic Processor Unit (SPU) and Memory Flow Controller (MFC).

`

SPU (Synergistic Processor Unit):

Each SPU is an independent processor with its own program counter and is optimized to run SPE threads spawned by the PPE. The SPU fetches instructions from its own LS and it loads and stores data from and to its own LS. With respect to accesses by its SPU, the LS are unprotected and untranslated storage.

MFC (Memory Flow Controller):

The MFC contains a DMA controller that supports DMA transfers. Programs running on the SPU, the PPE, or another SPU, use the MFC's DMA transfers to move instructions and data between the SPU's LS and main storage. (Main storage is the effective-address space that includes main memory, other SPEs' LS, and memory-mapped registers such as memory-mapped I/O [MMIO] registers.)

The MFC interfaces the SPU to the EIB; implements bus bandwidth-reservation features, and synchronizes operations between the SPU and all other processors in the system.

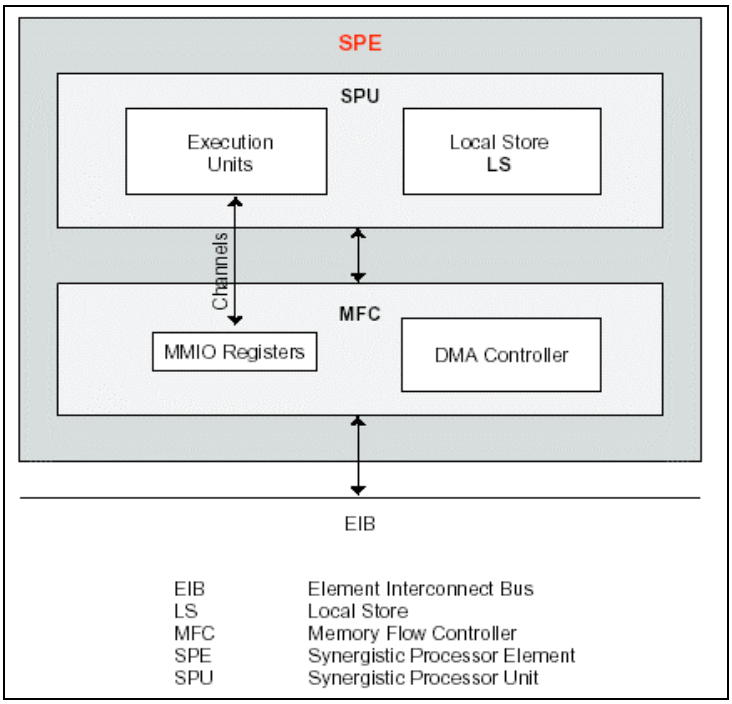


Figure 6 Interconnections in SPE [3]

`

To support DMA transfers, the MFC maintains and processes queues of DMA commands. After a DMA command has been queued to the MFC, the SPU can continue to execute instructions while the MFC processes the DMA command autonomously and asynchronously. The MFC also can autonomously execute a sequence of DMA transfers, in response to a DMA-list command. This autonomous execution of MFC DMA commands and SPU instructions allows DMA transfers to be conveniently scheduled to hide memory latency.

SIMD Vectorization:

A vector is an instruction operand containing a set of data elements packed into a one-dimensional array. The elements can be integer or floating-point values. Most Vector/SIMD Multimedia Extension and SPU instructions operate on vector operands. Vectors are also called *SIMD operands* or *packed operands*. SIMD processing exploits data-level parallelism. Data-level parallelism means that the operations required to transform a set of vector elements can be performed on all elements of the vector at the same time. That is, a single instruction can be applied to multiple data elements in parallel.

An Example:

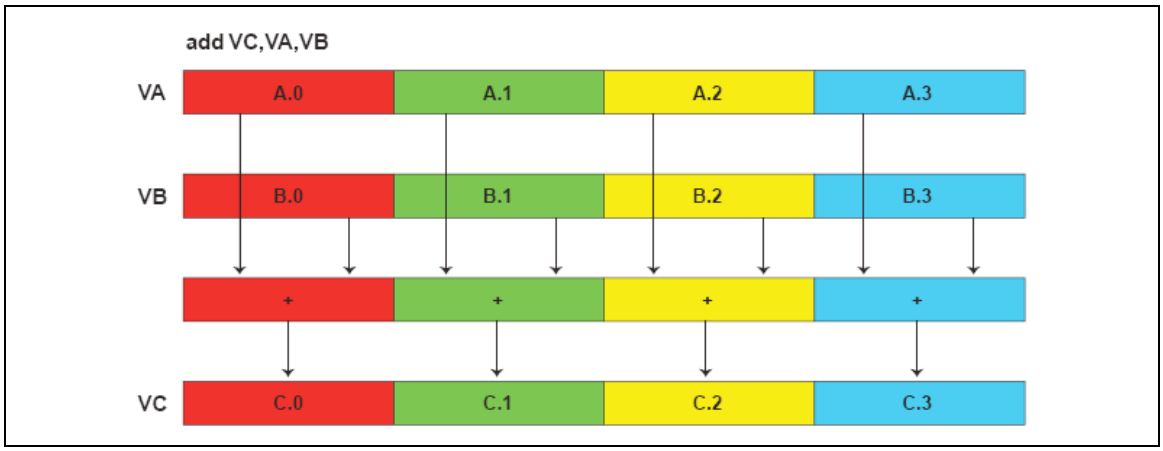


Figure 7 Vector Addition Example [3]

`

2.8 Communication between PPE - SPE

The PPE communicates with the SPEs through privileged-state and problem-state MMIO registers supported by the MFC of each SPE. These registers are accessed by the associated SPE through its channel mechanism which consist of unidirectional registers and queues and support logic. The three primary communication mechanisms between the PPE and SPEs are:

1. **Mailboxes**
2. **Direct Memory Access**
3. **Signal Notification Registers**

Mechanism	Description
DMA transfers	Used to move data and instructions between main storage and an LS. SPEs rely on asynchronous DMA transfers to hide memory latency and transfer overhead by moving information in parallel with SPU computation.
Mailboxes	Used for control communication between an SPE and the PPE or other devices. Mailboxes hold 32-bit messages. Each SPE has two mailboxes for sending messages and one mailbox for receiving messages.
Signal notification	Used for control communication from the PPE or other devices. Signal notification (also called *signaling)* uses 32-bit registers that can be configured for one-sender-to-one-receiver signalling or many-senders-to-one-receiver signalling.

Figure 8 Inter-Processor Communication Mechanisms [3]

2.8.1 Selected mechanism for our implementation

Our implementation uses “Mailboxes” for broadcasting characters as well as for sending acknowledgements. In the construction steps, when each SPE is working on their corresponding sub-trees they need to first complete the ongoing update before receiving another character. Also as we just need to communicate one character at a time, its better to use mailboxes than any other communication mechanism.

DMA is mainly used in Shared Memory model however we are maintaining suffix trees in the local store only. Signal Notification is mainly used for one-sender-to-one-receiver signaling or many-senders-to-one-receiver signaling, however, we have to mostly broadcast of characters which is one-sender-to-many-receivers signaling. So using mailboxes is the best option for our case.

`

2.9 Hands-on Cell Simulator

2.9.1 Simulator Basics

The IBM Full System Simulator for the Cell Broadband Engine is a generalized simulator that can be configured to simulate a broad range of full-system configurations. It supports full *functional simulation*, including the PPE, SPEs, MFCs, PPE caches, bus, and memory controller. It can simulate and capture many levels of operational details on instruction execution, cache and memory subsystems interrupt subsystems, communications, and other important system functions.

If accurate timing and cycle-level simulation are not required, the simulator can be used in its *functional-only mode*, running as a debugger to test the functions and features of a program. If cycle-level analysis is required, it can be used in *performance simulation* (or *timing simulation)* mode, to get accurate performance analyses. Simulator configurations are extensible and can be modified using Tool Command Language (Tcl) commands to produce the type and level of analysis required.

The simulator itself is a general tool that can be configured for a broad range of microprocessors and hardware simulations. The SDK, however, provides a ready-made configuration of the simulator for Cell Broadband Engine system development and analysis.

2.9.2 Operating-System Modes

The simulator has two modes of operation, with regard to operating systems: *Linux mode* and S*tandalone mode*.

2.9.2.1 Linux Mode

In Linux mode, after the simulator is configured and loaded, the simulator boots the Linux operating system on the simulated system. At runtime, the operating system is simulated along with the running programs. The simulated operating system takes care of all the system calls, just as it would in a non-simulation (real) environment.

`

2.9.2.2 Standalone Mode

In standalone mode, the application is loaded without an operating system. Standalone applications are user-mode applications that are normally run on an operating system. On a real system, these applications rely on the operating system to perform certain tasks, including loading the program, address translation, and system-call support. In standalone mode, the simulator provides some of this support, allowing applications to run without having to first boot an operating system on the simulator.

The simulator loads executables without address translation, so that the effective address is the same as the real address. Therefore, all addresses referenced in the executable must be valid real addresses. If the simulator has been configured for 64 MB of memory all addresses must fit in the range of x‘0’ to x‘3FFFFFF’.

2.9.3 Interacting with the Simulator

There are two ways to interact with the simulator:

- Issuing commands to the *simulated system*
- Issuing commands to the *simulator*

The simulated system is the Linux environment on top of the simulated Cell Broadband Engine, where you run and debug programs. You interact with it by entering commands at the Linux command prompt, in the *console window*. The console window is a Linux shell of the simulated Linux operating system.

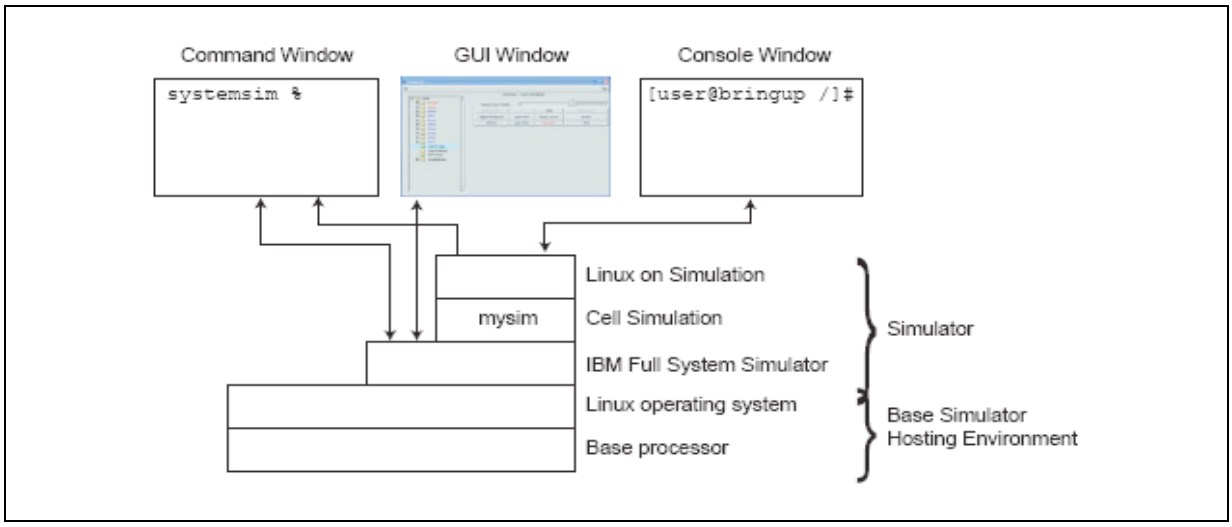


Figure 9 Simulator structure and screens [1]

`

You can also control the simulator itself, configuring it to do such tasks as collect and display performance statistics on particular SPEs, or set breakpoints in code. These commands are entered at the simulator command line in the *simulator command window*, or using the equivalent actions in the graphical user interface (GUI). The GUI is a graphical means of interacting with the simulator.

2.9.3.1 Graphical User Interface

The simulator's GUI offers a visual display of the state of the simulated system, including the PPE and the eight SPEs. You can view the values of the registers, memory, and channels, as well as viewing performance statistics. The GUI also offers an alternate method of interacting with the simulator

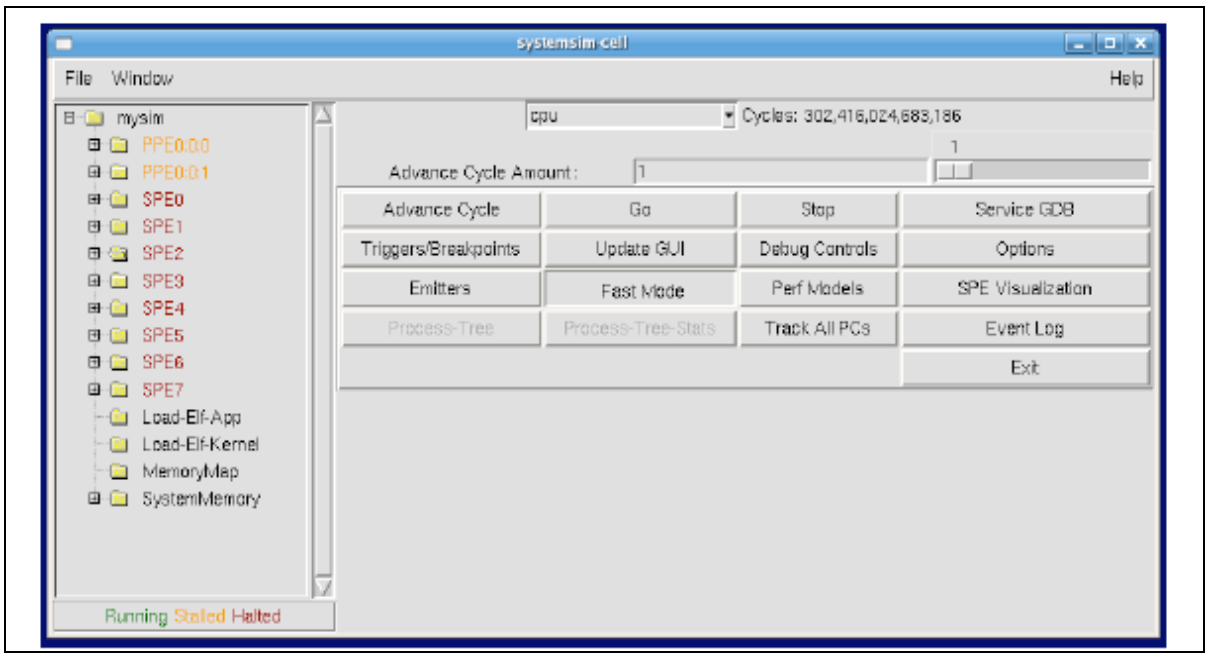


Figure 10 Main Graphical User Interface for the Simulator [1]

The main GUI window has two basic areas: the vertical panel on the left, and the rows of buttons on the right. The vertical panel represents the simulated system and its components. The rows of buttons are used to control the simulator.

`

Chapter 3

SUFFIX TREES AND ALGORITHMS

3.1 Suffix Trees

A suffix tree is constructed for a given string. When the nodes in a suffix tree are traversed in a depth first fashion, a suffix of the given string is obtained. So, a string of size m can be searched in a time of the order O(m). The ability to search efficiently with mismatches might be the suffix tree's greatest strength.

3.1.1 History

The concept was first introduced as a *position tree* by Weiner in 1973 in a paper which Donald Knuth subsequently characterized as "Algorithm of the Year 1973". The construction was greatly simplified by McCreight in 1976 [5], and also by Ukkonen in 1995 [6]. Ukkonen provided the first linear-time online-construction of suffix trees, now known as Ukkonen's algorithm [7].

3.1.2 Basic Definitions

When describing how to build a suffix tree for an arbitrary string, we will refer to the generic string S of length m. We do not use P or T (denoting pattern and text) because suffix trees are used in a wide range of applications where the input string sometimes plays the role of a pattern, sometimes a text, sometimes both, and sometimes neither. As usual the alphabet is assumed finite and known. After discussing suffix tree algorithms for a single string S, we will generalize the suffix tree to handle sets of strings.

Definition A suffix tree T for an m-character string S is a rooted directed tree with exactly m leaves numbered 1 to m. Each internal node, other than the root, has at least two children and each edge is labeled with a non-empty substring of S. No two edges out of a node can have edge- labels which begin with the same character. The key feature of the suffix tree is that for any leaf i, the concatenation of the edge- labels on the path from the root to leaf i exactly spell out the suffix of S that starts at position i. That is, it spells

`

out S[i..m]. For example, the suffix tree for the string xabxac is shown in Figure 1. The path from the root to the leaf numbered 1 spells out the full string S = xabxac, while the path to the leaf numbered 5 spells out the suffix ac which starts in position 5 of S.

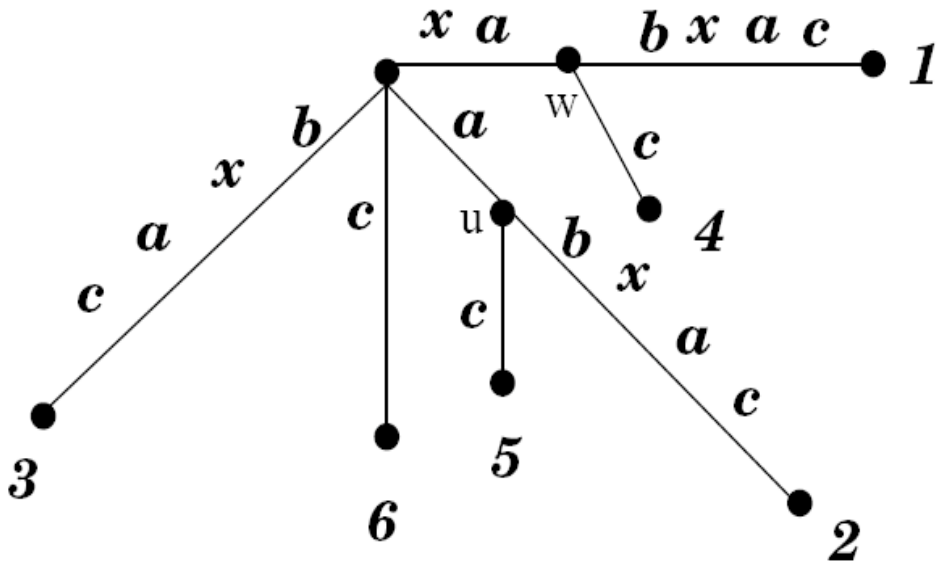


Figure 11 Suffix tree for string xabxac

3.2 Algorithms

3.2.1 O(n^2) algorithm to build a suffix tree

To solidify the definition of a suffix tree and develop the reader's intuition, we present a straightforward algorithm to build a suffix tree for string S. This naïve method first enters a single edge for suffix S[1..m]$ (the entire string) into the tree, then it successively enters suffix S[i..m]$ into the growing tree, for i increasing from 2 to m. We let Ni denote the intermediate tree that encodes all the suffixes from 1 to i.

In detail, tree N1 consists of a single edge between the root of the tree and a leaf labeled 1. The edge is labeled with the string S$. Tree Ni+1 is constructed from Ni as follows: Starting at the root of Ni find the longest path from the root whose label matches a prefix of S[i + 1..m]$. This path is found by successively comparing and matching characters in suffix S[i+1..m]$ to characters along a unique path from the root, until no further matches

`

are possible. The matching path is unique because no two edges out of a node can have labels which begin with the same character. At some point, no further matches are possible because no suffix of S$ is a prefix of any other suffix of S$. When that point is reached, the algorithm is either at a node, w say, or it is in the middle of an edge. If it is in the middle of an edge, (u, v) say, then it breaks edge (u, v) into two edges by inserting a new node, called w, just after the last character on the edge that matched a character in S[i + 1..m], and just before the first character on the edge that mismatched. The new edge (u,w) is labeled with the part of the (u, v) label that matched with S[i + 1..m], and the new edge (w, v) is labeled with the remaining part of the (u, v) label. Then (whether a new node w was created, or already existed at the point where the match ended), the algorithm creates a new edge (w, i+1) running from w to a new leaf labeled i+1, and it labels the new edge with the unmatched part of suffix S[i + 1..m]$. The tree now contains a unique path from the root to leaf i + 1, and this path is labeled with the string S[1 + 1..m]$. Note that all edges out of the new node w have labels which begin with different first characters, and so it follows inductively that no two edges out of a node have labels with the same first character.

3.2.2 Improvements to make the algorithm linear time

The most important improvement can be to use Suffix Pointers to traverse the tree specifically to nodes where update needs to be done at the time of online construction. Structure Suffix Tree contains a node pool which allocates new nodes when a new edge is to be created, however Structure Node contains a vector of edges. Each node contains a parent pointer along with the information of edge connecting between the two. This information makes updates faster when new characters are added to the string. Every node also has a ‘link’ field stores the suffix pointer to follow when updating. The function for online-construction now has the complexity O(n), so it is linear in the size of current string.

`

3.2.3 O(n) algorithm to build a suffix tree using suffix links

Suffix links are a key feature for linear-time construction of the tree. A chain of suffix links [3] is maintained at any point during the construction so that if a new character comes the updates can take place in linear time. The beginning of the chain is represented by the node pointed by the Start-Link pointer. A suffix link is a pointer from an internal node to another internal node. These internal nodes are those nodes for which a suffix of the string is formed if we traverse from the root node to this node. When a new character comes, various cases in which suffix links have to be updated and handled differently are described below**.**

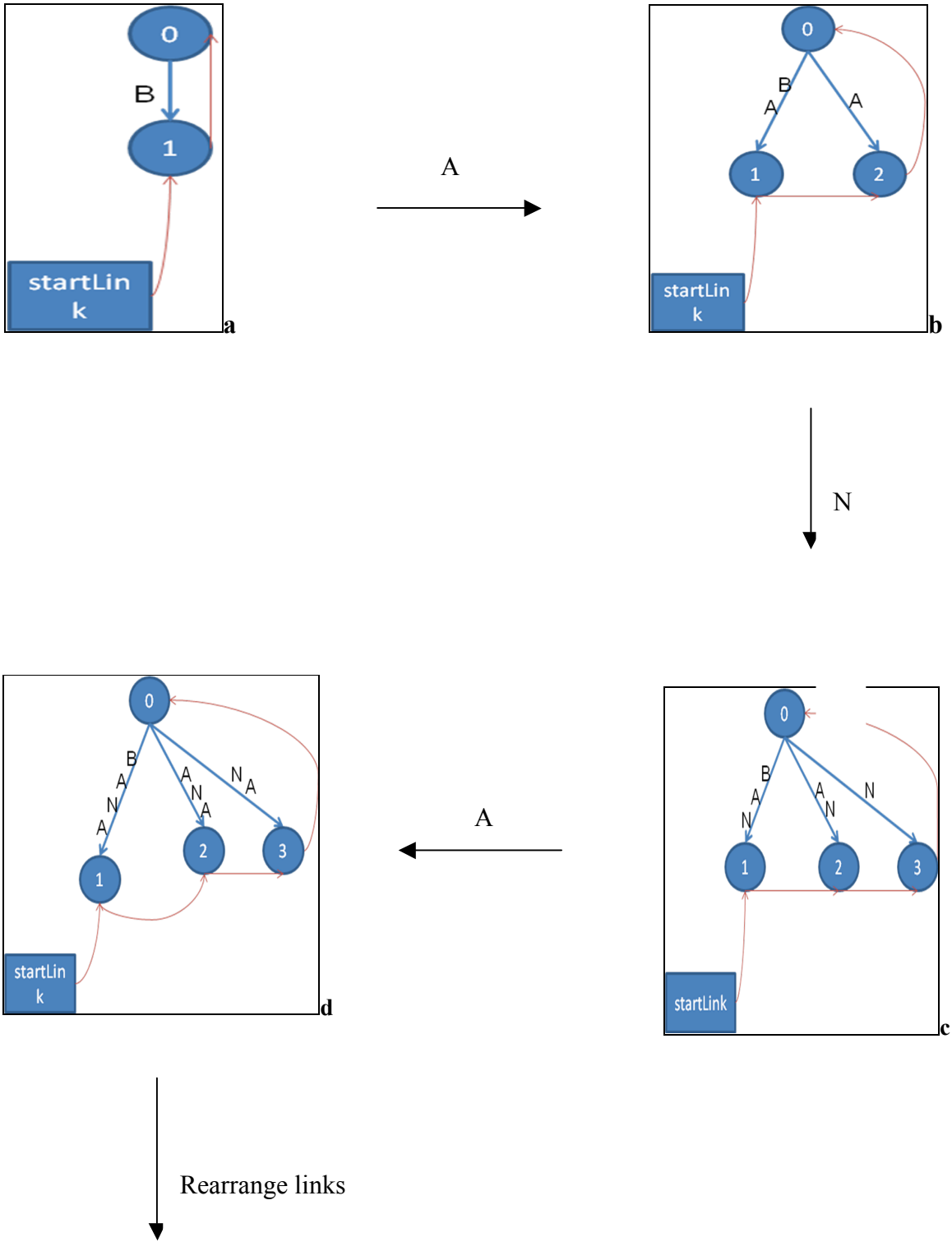
Case 1: First character comes, Fig. a Initiation Step. Simply a new node is allocated and its link pointer is set to root. Start-Link pointer is updated to point to this new node.

Case 2: For every next incoming character (say A), follow the path from Start-Link and append the new character to the string of incoming edge of that leaf-node. If suffix link ends on a node, and there is no pre-existing edge with the same starting character, then add a new edge to that node labeled with A e.g. Fig. b.

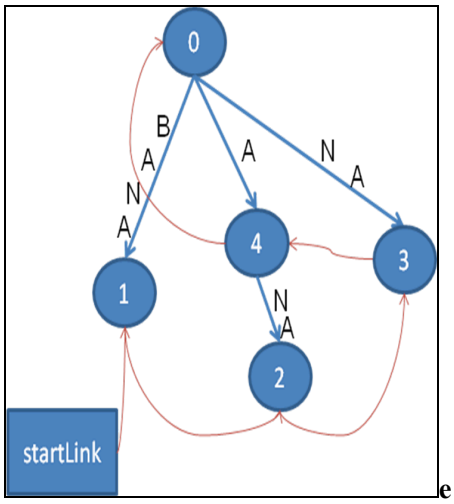
Case 3: If there exists an edge with the same starting character as the newly arrived character (say A), then split that edge, insert a new node. Splitting takes place only after A. Make the previous suffix link pointing to this new node and a link from the new node to the ROOT. E.g. Fig. d to Fig. e

Procedure for making Suffix Tree of a string, using the Suffix Link scheme

Sample string: **BANANA**

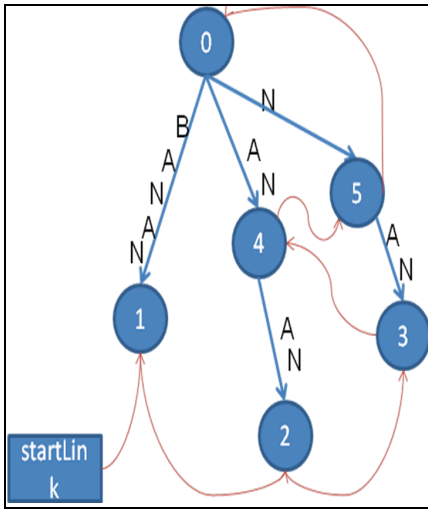

0
B
1
startLin
k
a
A
0
B
A
A
1
2
startLin
k
b
N
0
B
A
N
A
N
A
A
1
2
3
startLin
k
d
A
0
B
A
N
A
N
N
1
2
3
startLink
c
Rearrange links

`


0
B
A
N
A
A
N
A
4
3
1
N
A
2
startLink

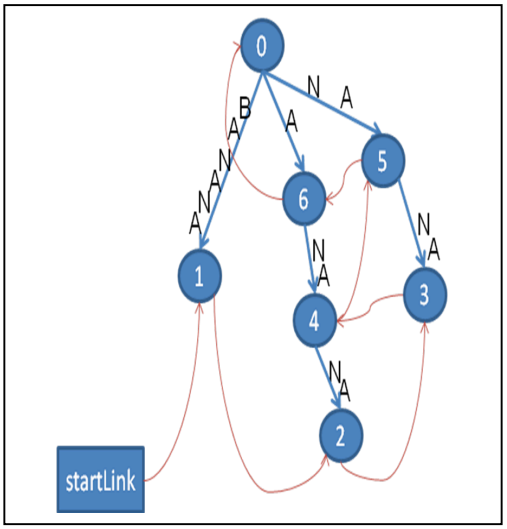
e

N


0
N
B
A
N
A
A
N
N
A
5
4
A
N
1
3
A
N
2
startLin
k

f

A


0
N
A
B
A
A
N
5
N
A
6
N
A
1
N
A
3
4
N
A
2
startLink

`

3.3 Porting on CELL BE

IBM Cell BE has a PPU which can directly run the uniprocessor code with some modifications. So, the first step was to get the uniprocessor tree construction code running on the PPU. Next step was to identify code segments which can be parallelized and hence assigned to the eight SPEs to gain speedup. There were many options. One was that each SPE just handles the update corresponding to one particular character and the whole tree is stored in main memory. In this case, not a lot of speedup was expected as there are a lot of dependencies in the order of updates depending on the arrival of characters. Also a lot of data transfer is needed in this case as updates have to be performed in the tree stored in main memory by the processors with local store (LS). So, this approach was discarded.

Then we analyzed a second approach of assigning the whole sub-trees to different processors depending on the beginning character. In a typical bioinformatics application, a nucleotide sequence contains four different characters in the corresponding string namely A, G, C and T. If we use four processors to handle sub-trees starting with these four characters, we can easily handle update tasks without losing efficiency in dependencies as updates in two sub-trees can be done in parallel without violating any rules. So in this new approach each SPE was identified by a unique character and it would then continue the formation of suffix tree branch. For each new character observed, a new SPE thread would be initiated. Now every incoming character from PPE is incorporated into the suffix sub-trees at each SPE. The only issue that remained unsolved at this moment was handling and updating link pointers as there could be links from one branch to another. That means we could have links from SPEi to SPEj. After some analysis of construction cases, we came up with a simple scheme which handled it efficiently by redirecting the link pointer to the head of sub-tree handled by SPE.

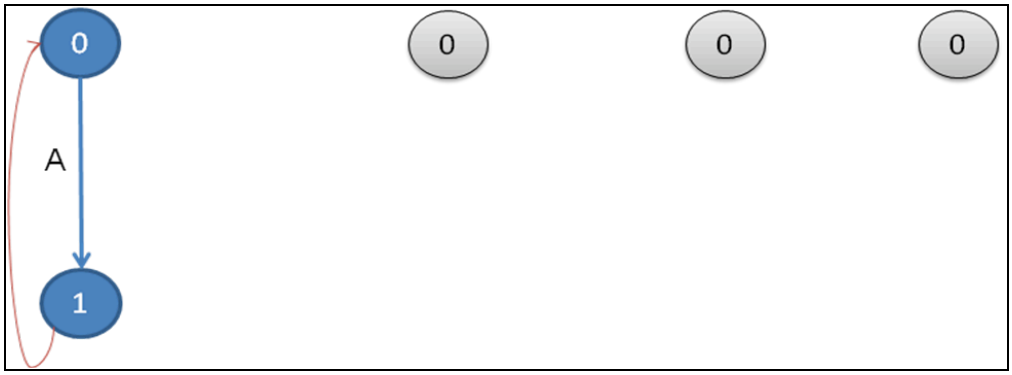
The above scheme was implemented successfully and gave expected performance speedups for eight character strings. But using our approach only four SPEs were to be used in a typical bioinformatics application which left the other four SPEs in dormant

`

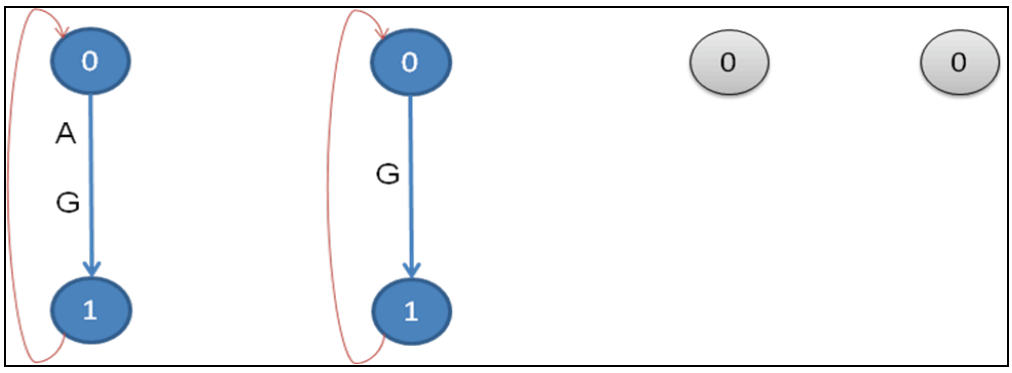
state. On exploring the possibility of exploiting the processing power of the dormant SPEs, we came up with some approaches. The final solution that we chose to implement was to use the other four SPEs in construction of another suffix tree in parallel assuming that the tree construction server is a busy one. In this approach one major issue was to see whether the PPE can handle broadcast of characters for an additional suffix tree without affecting the speed of construction of the first tree. But since sub-tree updates in individual SPEs take much more time than what the PPE takes in broadcasting characters, it was a justified step to go for the dual parallel construction mode. Later on, after final implementation, test results showed significant speedups while working in this mode. In this way, PPE utilization was improved and all SPEs could now be used simultaneous even for bioinformatics applications.

Example: AGGCCTTA (Adding characters one by one)

Starting character: A

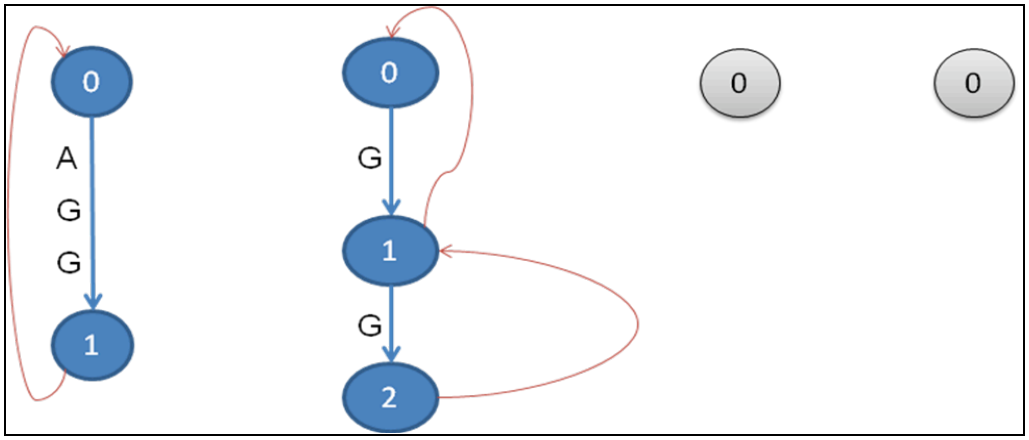


Next character: G

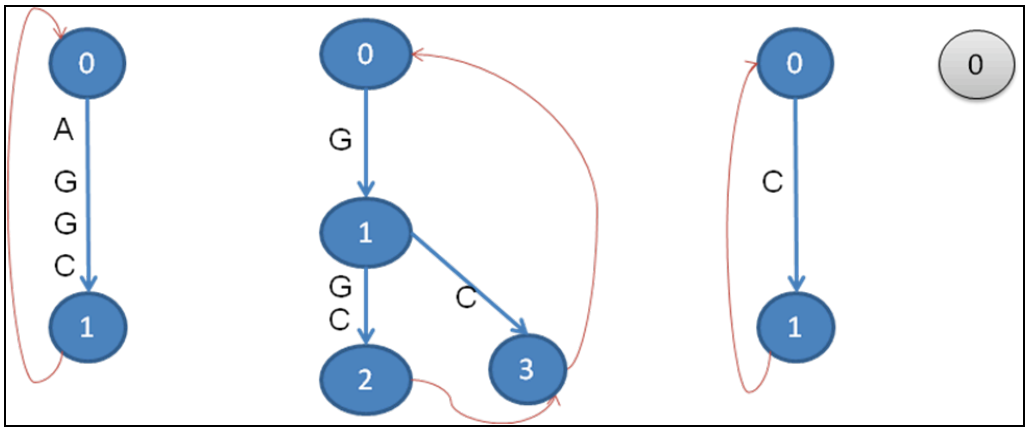


`

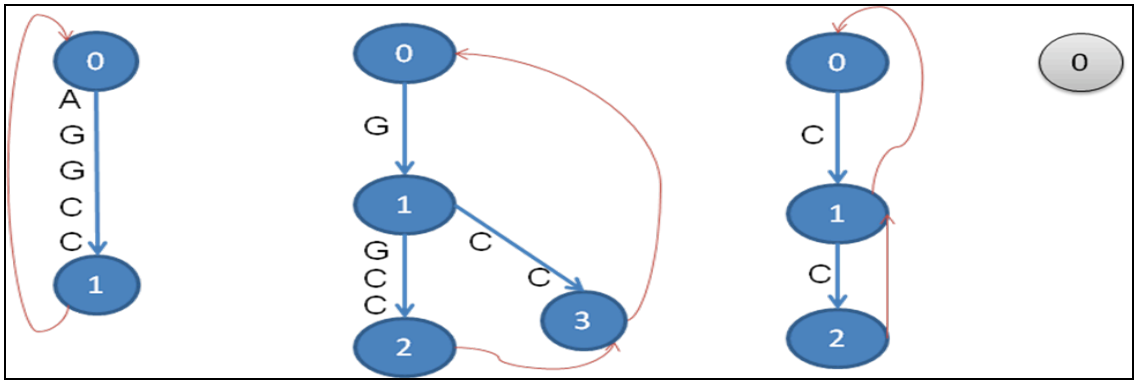
Next character: G



Next character: C

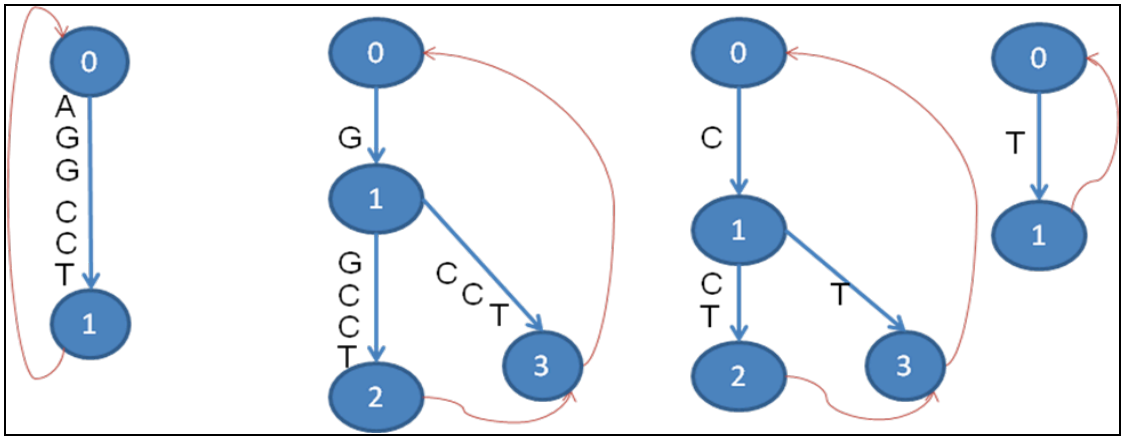


Next character: C

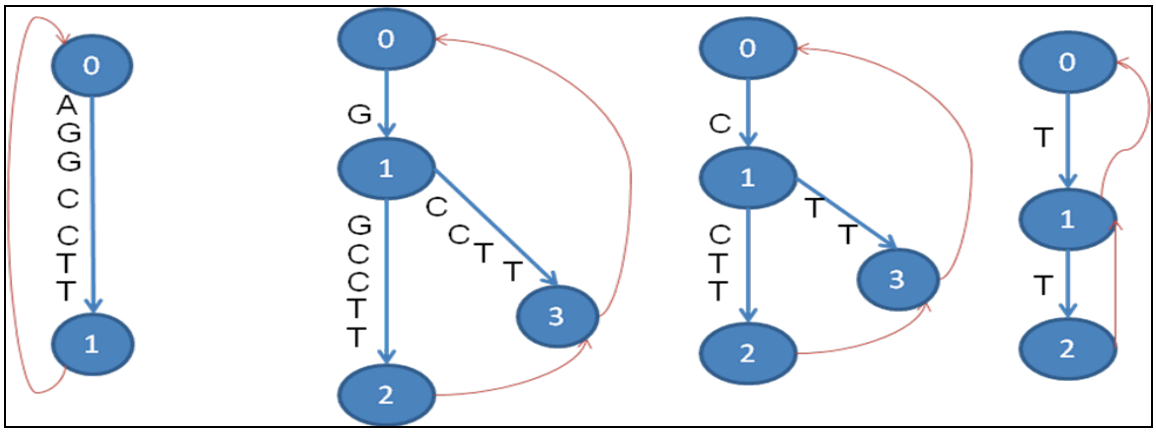


`

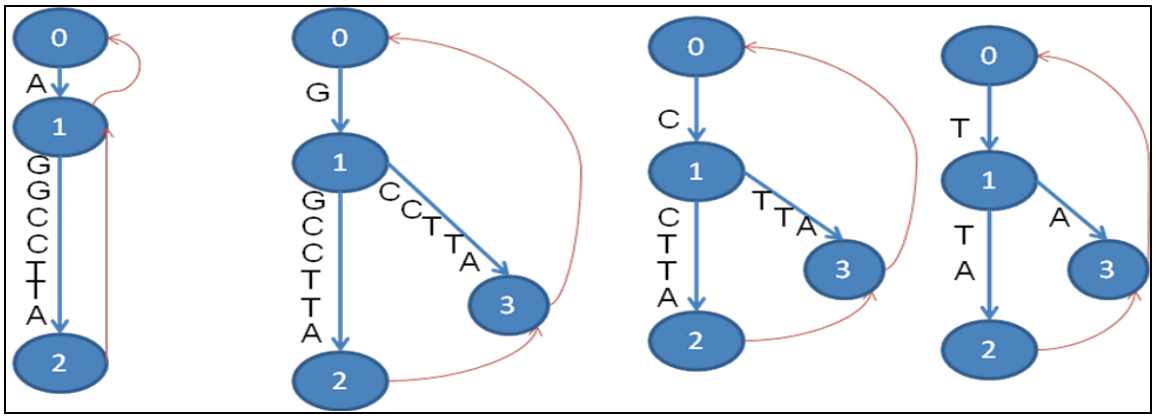
Next character: T



Next character: T



Next character: A



`

Chapter 4

IMPLEMENTATION DETAILS

4.1 Classes and Functions

4.1.1 Class *Edge*

Edge Class is used to represent the edges of the suffix tree. Each edge contains some part of the string used for generation of suffix tree. For space efficiency, we just store the beginning and end indices of the substring contained. Also an edge contains simulated pointers to origination and destination nodes. Both *Node* Class and *SuffixTree* Class are declared as friend classes.

Table 1 Class *Edge* Members

Members	*Description*
beg_index	Integer denoting the beginning index of substring contained in the *Edge*.
end_index	Integer denoting the end index of substring contained in the *Edge*.
fromNode	Integer used to store the simulated pointer to originating *Node* for the *Edge*.
toNode	Integer used to store the simulated pointer to destination *Node* for the *Edge*.

Table 2 Class *Edge* Functions

Functions	*Description*
Edge(int beg, int end, int from, int to)	Constructor for initializing substring indices and origination and destination *Node* pointers.

`

4.1.2 Class *Node*

Node Class is used to represent actual nodes in the suffix tree structure. It contains a vector of edges that originate at it. The parent node and edge information is also stored. For various applications, maintaining a mapping of existence of endmarkers in the subtree is helpful. A count of characters from root to this node is also maintained for the same purpose. *SuffixTree* Class is declared as a friend class.

Table 3 Class *Node* Members

Members	*Description*
Vector<Edge> v	A vector of *Edge* representing all those originating from the current *Node*.
Parent	Integer used to store the simulated pointer to the parent *Node* of current *Node*.
ParentEdge	Integer used to store the simulated pointer to the *Edge* connecting the parent *Node* to this *Node.*
link	Integer used to store the simulated pointer to the next *Node* in the suffix link chain.
mark	Byte representing the existence of endmarkers in the subtree rooted at current node.
count	Integer which stores the number of characters starting from *ROOT Node* following all edges to reach current *Node*

Table 4 Class *Node* Functions

Functions	*Description*
Node()	Constructor which cleans the vector *v*. Sets *link* and *Parent* to -1 to prevent linking to garbage. Clears *mark* and *count*.
void Insert(Edge *e*)	Inserts edge *e* into the vector *v*.

`

4.1.3 Class *Suffix Tree*

Suffix tree class is used to represent the main suffix tree structure. It has an array of nodes which can be allocated on-demand for updating the tree structure. A suffix link chain is maintained starting at startlinkNode. Various useful functions are provided for initialization and update of suffix tree as well as for applications such as longest common substring search or longest palindrome search. Functionality for construction of Generalized Suffix Trees is also provided which is helpful in most applications.

Table 5 Class *SuffixTree* Members

Members	*Description*
startlinkNode	Stores the simulated pointer to starting link of the suffix link chain.
Max	Maximum limit on number of *Node*s.
Nodecount	Current number of allocated *Node*s.
Node* arr	array of *Node*s
Str	String for which Suffix Tree is to be constructed.
str_len	Integer stores the length of *str*.
Maxcountnode	*Node* which has the maximum *count* value.

Table 6 Class *SuffixTree* Functions

Functions	*Description*
SuffixTree(int maxno, char* strval)	Constructor which sets the maximum allowed number of *Node*s and sets the string value
void InitializeTree(char* string)	Initializes suffix tree with specified string value
void stringUpdate(char c)	Appends character c to string
int NodeAllocate()	Returns the simulated pointer to next available unused *Node*
void UpdateNodeMark(int currnode,	Notifies current *Node*, of the existence of

`

char endmarker)	endmarker in its subtree and updates the mark value
void OnlineUpdate(char c)	Incorporates new character c into the already constructed suffix tree
void DisplaySuffixes()	Displays all the possible suffixes
void LongestCommonSubstring()	Finds and displays the longest common substring among
bool StringSearch(char* query, int querylength)	Search for the query string in the built suffix tree
void InsertNode(Edge& e, int mid)	Splits the *Edge* e at index mid, inserting a new *Node* there

4.2 Problems Faced

Some main problems faced during the project are described below:

- Getting familiar with the new architecture and the IBM cell simulator.
- Identifying parallelizable steps in the construction of suffix trees for efficient load distribution at PPE to distribute tree-update tasks among SPEs.
- Handling of link pointers in tree construction when branches exist in different SPEs.
- Synchronization of outgoing and incoming messages at mailboxes during exchange of data or control information.
- Making use of all the eight SPEs however only sub-trees starting with A, G, C and T exist for real-life problems. An innovative dual construction mode solution was implemented which enables us to create two suffix trees in parallel making use of all the eight processors simultaneously.
- To run programs and observe results, it takes hours on simulator and thus make it difficult to modify and test for all possible cases of application.

`

Chapter 5

SUFFIX TREE APPLICATIONS & RESULTS

Suffix Trees can be used to solve a large number of string problems that occur in text-editing, free-text search, computational biology, and other application areas. A generalized suffix tree is implemented as it plays a central role in many applications involving suffix trees.

5.1 Generalized Suffix Tree

A generalized suffix tree is a suffix tree for a set of strings. Given the set of strings $D = S1, S2, S3 \ldots Sd$ of total length n, it is a suffix tree containing all n suffixes of the strings. It can be built in $\Theta(n)$ time and space, and you can use it to find all z occurrences of a string P of length m in $O(m + z)$ time, which is asymptotically optimal (assuming the size of the alphabet is constant). When constructing such a tree, each string should be padded with a unique out-of-alphabet marker symbol (or string) to ensure no suffix is a substring of another, guaranteeing each suffix is represented by a unique leaf node.

For construction of generalized suffix trees, our algorithm for construction of normal suffix tree was modified to handle cases after end-markers have been incorporated into the tree. For any path from the root node to a leaf node, an end-marker cannot be followed by another character.

`

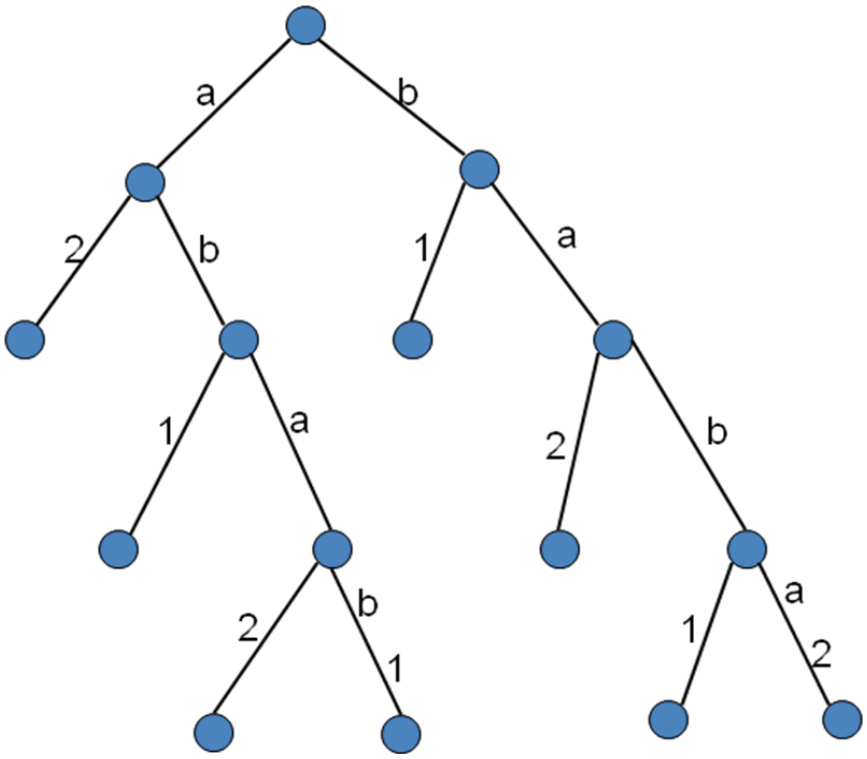


Figure 12 Generalized Suffix Tree of "abab1baba2"

5.2 String Search

The simplest and least efficient way to see where one string occurs inside another is to check each place it could be, one by one, to see if it's there. So first we see if there's a copy of the needle in the first few characters of the haystack; if not, we look to see if there's a copy of the needle starting at the second character of the haystack; if not, we look starting at the third character, and so forth. In the normal case, we only have to look at one or two characters for each wrong position to see that it is a wrong position, so in the average case, this takes O($n + m$) steps, where n is the length of the haystack and m is the length of the needle; but in the worst case, searching for a string like "aaaab" in a string like "aaaaaaaaab", it takes O(nm) steps.

Table 7 Classes of string searching algorithms

	Text not preprocessed	*Text preprocessed*
Patterns not preprocessed	Elementary algorithms	**Index methods**
Patterns preprocessed	Constructed search engines	Signature methods

`

Index Methods: Faster search algorithms are based on preprocessing of the text. After building a substring index, for example a suffix tree or suffix array, the occurrences of a pattern can be found quickly. As an example, a suffix tree can be built in $\Theta(m)$ time, and all z occurrences of a pattern can be found in $O(m + z)$ time (if the alphabet size is viewed as a constant).

Our Solution

We are able to construct Suffix Tree in parallel with a theoretical complexity of O(m/p) where p is the number of processors used and generally conforms with the number of distinct characters available in the string. Each SPE handles a branch of the suffix tree and hence can handle queries in parallel. For a busy server handling randomized queries continuously, it can be shown that all the processors are used and hence significant speedup is observed for this pipelined search.

Example

The following figure shows the suffix tree formed for the string input “Mississippi”. For this example four SPEs are initiated and corresponding sub-trees formed. A string “ssippi” is searched for in this suffix tree. The search returns true after it discovers that a path exists from the root node such that on concatenation of contents of the edges on the path the searched string is obtained. Thick arrows show the matching path for “ssippi”.

`

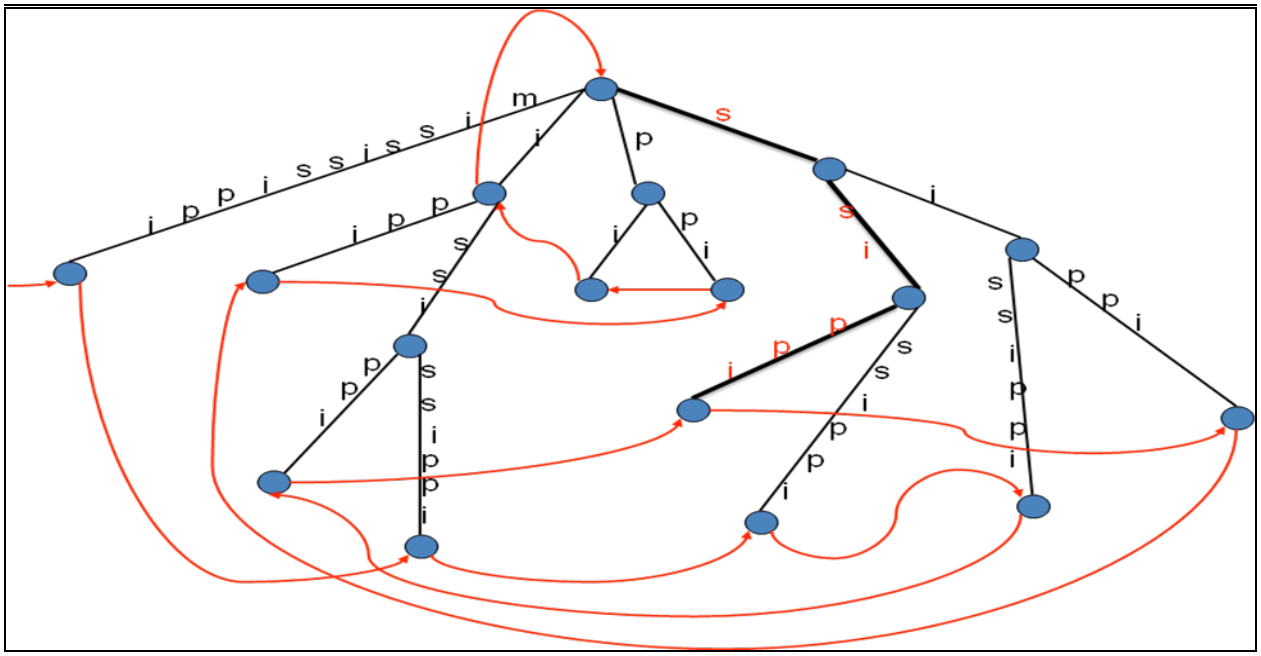


Figure 13 Suffix Tree of "mississippi" with search query "ssipp" returns 'true'

For another search string “ssipi”, it returns false as there is mismatch after ‘p’ as shown in the following figure.

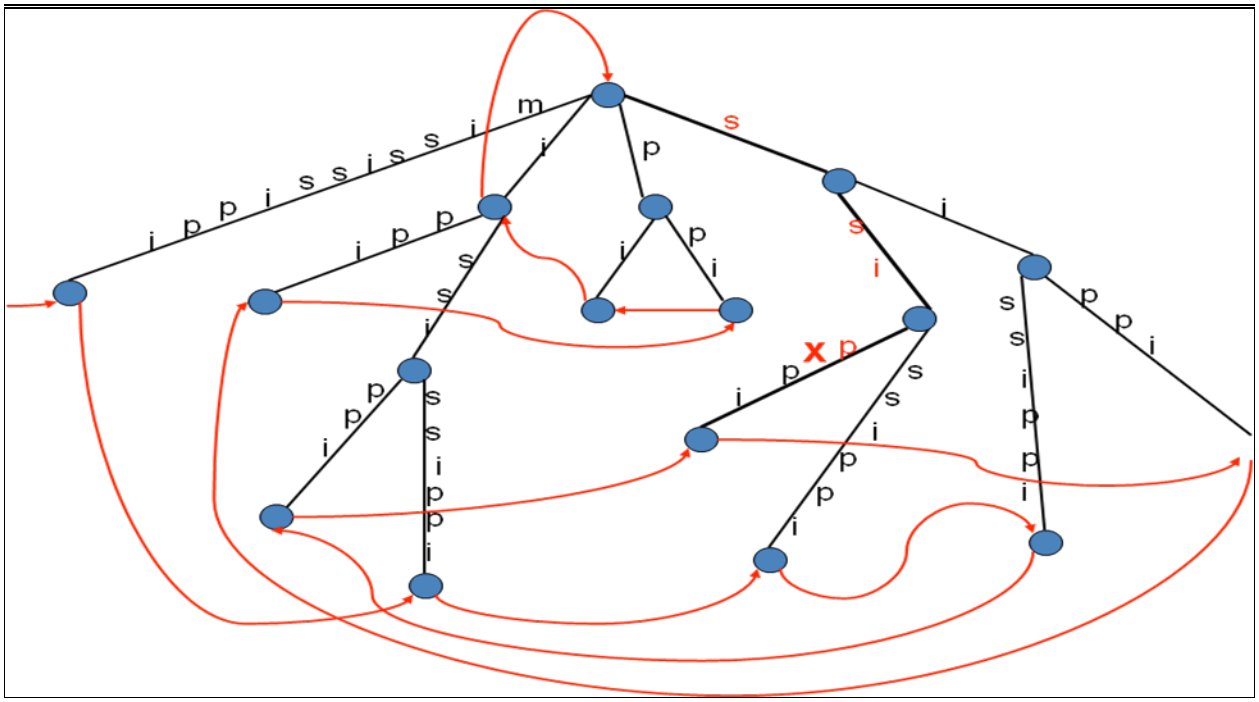


Figure 14 Suffix tree for Mississipi with search query "ssipi" returns 'false'

`

5.3 Longest Repeated Substring

The **longest repeated substring problem** is finding the longest substring of a string that occurs at least twice. This problem can be solved in linear time and space by building a suffix tree for the string, and finding the deepest internal node in the tree. The string spelled by the edges from the root to such a node is a longest repeated substring. The problem of finding the longest substring with at least *k* occurrences can be found by first preprocessing the tree to count the number of leaf descendants for each internal node, and then finding the deepest node with at least *k* descendants.

Our Solution

We have solved this problem by first building a generalized suffix tree with an end-marker containing only one string. In this case the longest repeated substring is indicated by the deepest fork node in the suffix tree, where depth is measured by the number of characters traversed from root. The longest repeated substring problem can be solved in O(n) time complexity in uniprocessor. In our case, theoretical speedup of Px is achieved where P is the number of processors used. Actual observations and speedup results are shown after example.

Example

The suffix tree for “mississippi” is constructed as follows. All the (count, mark) values are displayed along with all internal nodes.

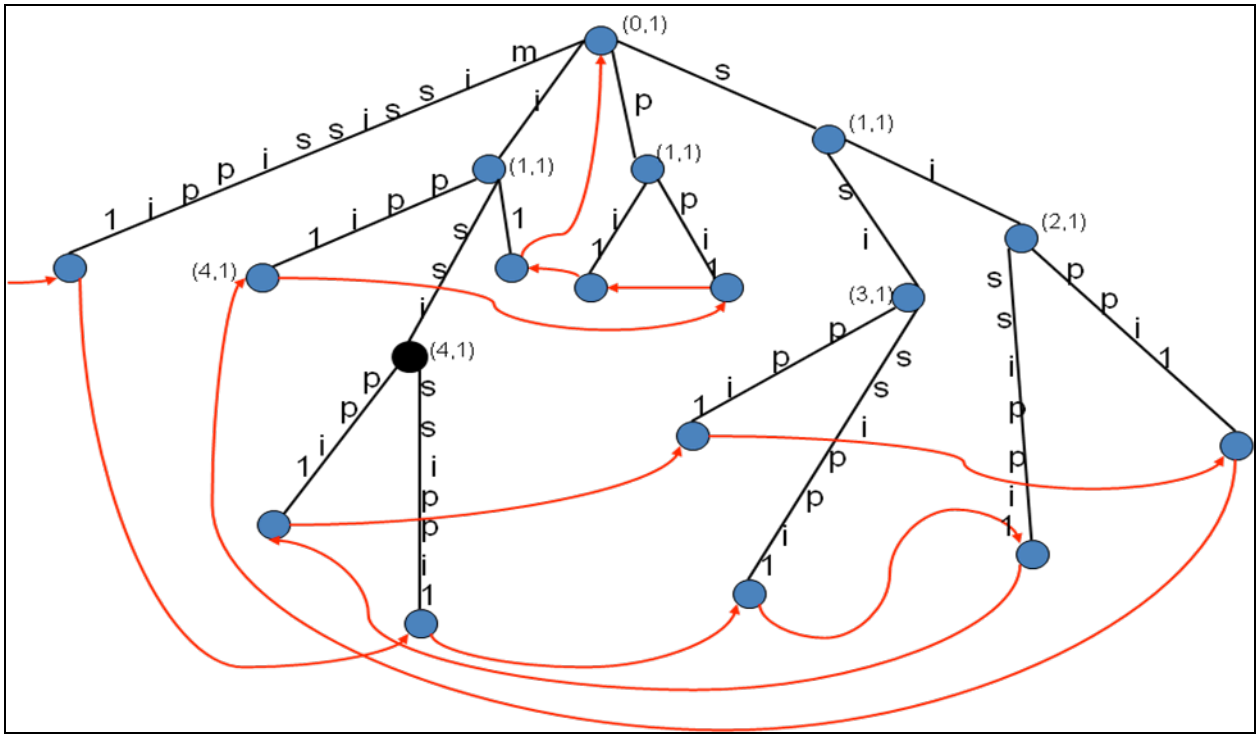


Figure 15 Suffix Tree for "mississippi" and finding longest repeated substring

The longest repeated substring is obtained by finding the internal node with maximum count value. The dark shaded node has the highest count value of 4, so it corresponds to the longest repeated substring for “mississippi”. Hence, the answer is “issi”.

5.4 Longest Common Substring

The **longest common substring problem** is to find the longest string (or strings) that is a substring (or are substrings) of two or more strings.

Problem Definition

Given two strings, S of length m and T of length n, find the longest strings which are a substrings of both S and T. A generalization is the **k-common substring problem**. Given the set of strings $S = \{S_1,...,S_K\}$, where $| S_i | = n_i$ and $\Sigma n_i = N$. Find for each $2 \leq k \leq K$, the longest strings which occur as substrings of at least k strings.

`

Our Solution

The *longest common substring* of two strings, txt_1 and txt_2, can be found by building a *generalized* suffix tree for txt_1 and txt_2: Each node is *marked* to indicate if it represents a suffix of txt_1 or txt_2 or both. The deepest node marked for both txt_1 and txt_2 represents the longest common substring. Equivalently, one can build a (basic) suffix tree for the string $txt_1\$txt_2\#$, where `$' is a special terminator for $txt_1$ and `#' is a special terminator for txt_2. The longest common substring is indicated by the deepest fork node that has both `...$...' and `...#...' (no $) beneath it.

First we build a Generalized suffix tree in parallel using SPEs with complexity O(n/p). We have incorporated a *mark* and *count* value in each node and these values are updated each time a new end-marker is introduced. The variable *mark* keeps a mapping of existence of end-markers in the sub-tree rooted at current node and *count* keeps track of number characters present if we follow edges from *root* to current node. At the time of construction we store a simulated pointer *maxcountnode* which points to the node with maximum value of *count* such that all or more than k end-markers exist in the sub-tree rooted at current node(according to variable *mark*) depending on whether it's a k-common substring problem or not. Since updating these variables is done simultaneously, at the end of the construction we have already found the deepest fork node of the tree. The characters contained in the edges on following the path from root node to this deepest fork node give the Longest Common Substring.

Example:

In the figure below you see the suffix tree for the strings "ABAB" and "BABA", padded with unique string terminators, to become "ABAB1" and "BABA2". The nodes representing "A", "B", "AB" and "BA" all have descendant leaves from all of the strings, numbered 1 and 2. In this case there are two possible solutions as shown by the non-shaded nodes. The longest common substring between the two strings is 'ABA' or 'BAB'.

`

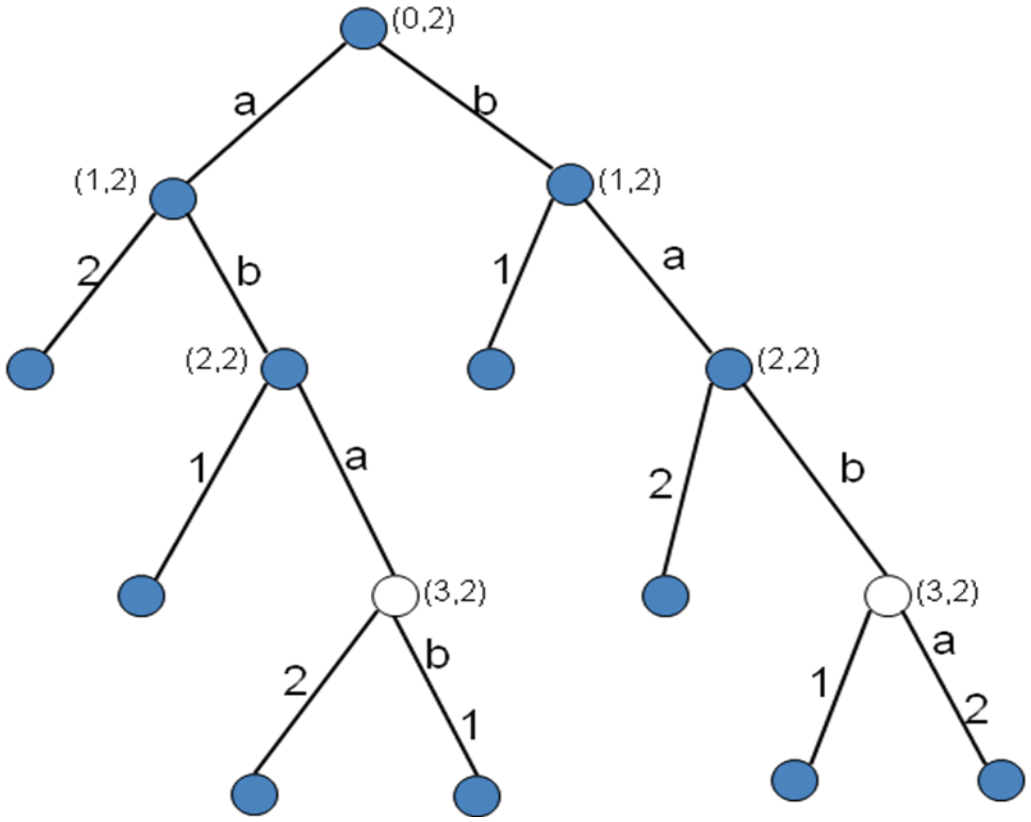


Figure 16 Longest Common Substring example

5.5 Palindromes

A palindrome is a string, P, such that P=reverse(P). e.g. `abba'=reverse(`abba'). e.g. `ississi' is the *longest* palindrome in `mississippi'. The longest palindrome of `txt[1..n]` can be found in O(n) time, e.g. by building the suffix tree for `txt$reverse(txt)#` or by building the generalized suffix tree for `txt` and `reverse(txt)`.

In most genomes or sets of genetic instructions, palindromic motifs are found. However, the meaning of palindrome in the context of genetics is slightly different from the definition used for words and sentences. Since the DNA is formed by two paired strands of nucleotides, and the nucleotides always pair in the same way (Adenine (A) with Thymine (T), Cytosine (C) with Guanine (G)), a (single-stranded) sequence of DNA is

`

said to be a **palindrome** if it is equal to its complementary sequence read backwards. For example, the sequence `ACCTAGGT` is palindromic because its complement is `TGGATCCA`, which is equal to the original sequence in reverse.

A palindromic DNA sequence can form a hairpin. Palindromic motifs are made by the order of the nucleotides that specify the complex chemicals (proteins) which, as a result of those genetic instructions, the cell is to produce. They have been specially researched in bacterial chromosomes and in the so-called bacterial interspersed mosaic elements scattered over them. Recently a research genome sequencing project discovered that many of the bases on the Y chromosome are arranged as palindromes. A palindrome structure allows the Y chromosome to repair itself by bending over at the middle if one side is damaged.

It is believed that palindromes are also found frequently in proteins, but their role in the protein function is not clearly known.

Our Solution

The problem can be viewed as a modification of 'Longest k-Common Substring'. To find the longest palindrome, we generate a generalized suffix tree of the string and the reverse of the string. Now the problem is equivalent to longest 2-common substring search which is solved as described above. Now the path from root to the deepest fork node contains the palindrome sequence.

Example

The following figure shows a generalized suffix tree of the given string "abab1baba2" along with (count, mark) values for each node. This value is useful for finding out the deepest fork node and hence the longest possible palindrome in the given string is searched. In this case, the non-shaded nodes represent the deepest fork nodes and two

`

palindromes have been found as a result of search. “bab” and “aba” both of which are palindromes of length 3.

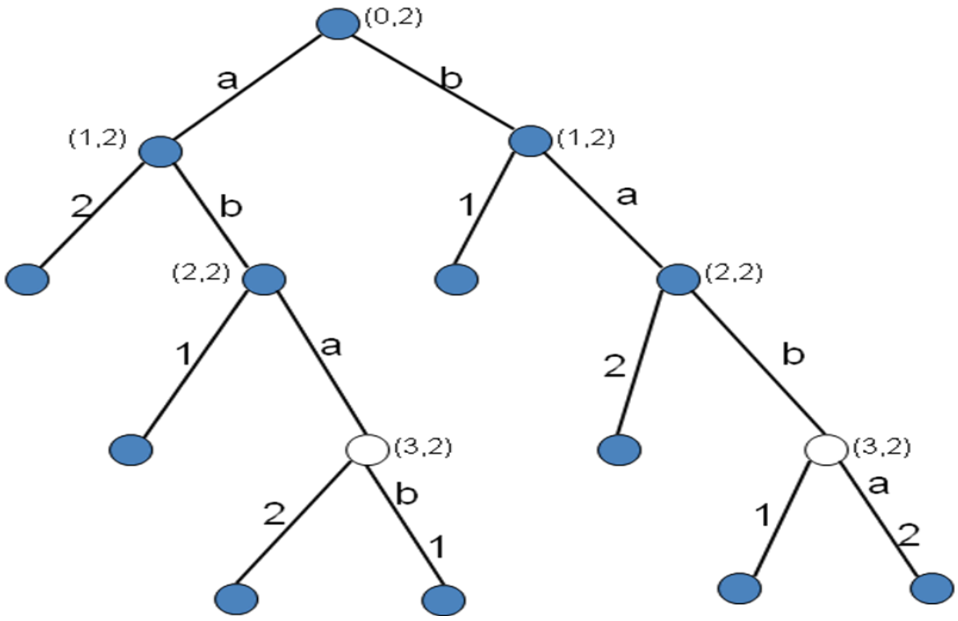


Figure 17 Generalized Suffix Tree for "abab1baba2" and palindrome search

5.6 Dual Parallel Construction Mode

Considering the real-life scenario where nucleotide sequences contain only four character (A, G, C and T) strings only four processors are used at a time. To make use of all 8 SPEs, we have suggested and implemented an innovative solution which makes use of remaining four processors for constructing a second suffix tree in parallel. As SPEs are handling more compute intensive task, so PPE can be assigned this second construction task monitoring without much overhead and hence significant improvement in processing power utilization is observed.

A comparison between the single mode and the dual mode is shown in the following figure. It clearly shows that only four processors are active in case of single mode and hence the remaining four remain unused. However, dual mode uses all the SPEs the do twice the work and hence better utilization.

Single Mode: One string is being processed on Cell BE (4 SPEs used)

Dual Mode: Two strings are being processed on Cell BE (8 SPEs used)

`

Observations and Speedup:

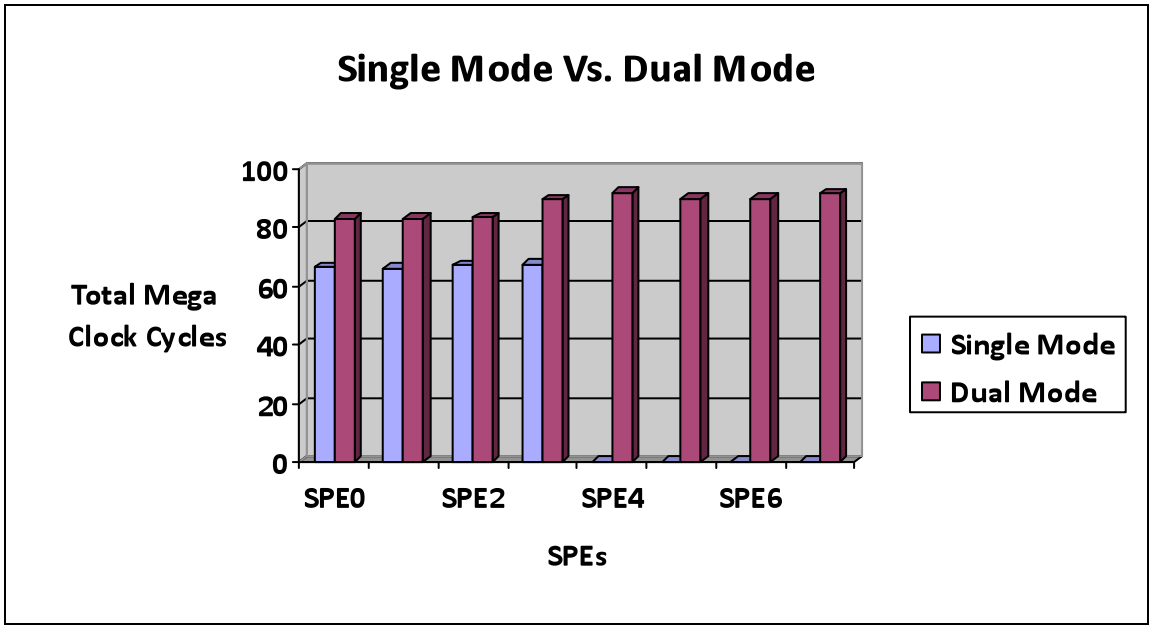


Figure 18 Comparison for string length = 1200

Time Calculations

Max. value of clock cycles required by any SPE (single mode) = 67.192402 x 10^6

Max. value of clock cycles required by any SPE (dual mode) = 91.741324 x 10^6

Frequency of each SPE = 3.2 GHz

Execution time on Cell BE = $\frac{\text{Average number of clock cycles required by each SPE}}{\text{Frequency of each SPE}}$

⇨ Execution time on Cell BE (single mode) = 0.019555 seconds

⇨ Execution time on Cell BE (dual mode) = 0.026700 seconds

Execution time on single processor (string length 1200 characters) = 0.152353 seconds

Speedup (single mode) = $\frac{\text{Execution time on single processor}}{\text{Execution time on Cell BE}}$

`

⇨ Speedup (single mode) = 7.79x

Speedup (dual mode) = $\frac{\text{2*Execution Time on Single Processor}}{\text{Execution Time on Cell BE}}$

⇨ Speedup (dual mode) = 11.41x

Observations for Applications running in Dual Mode

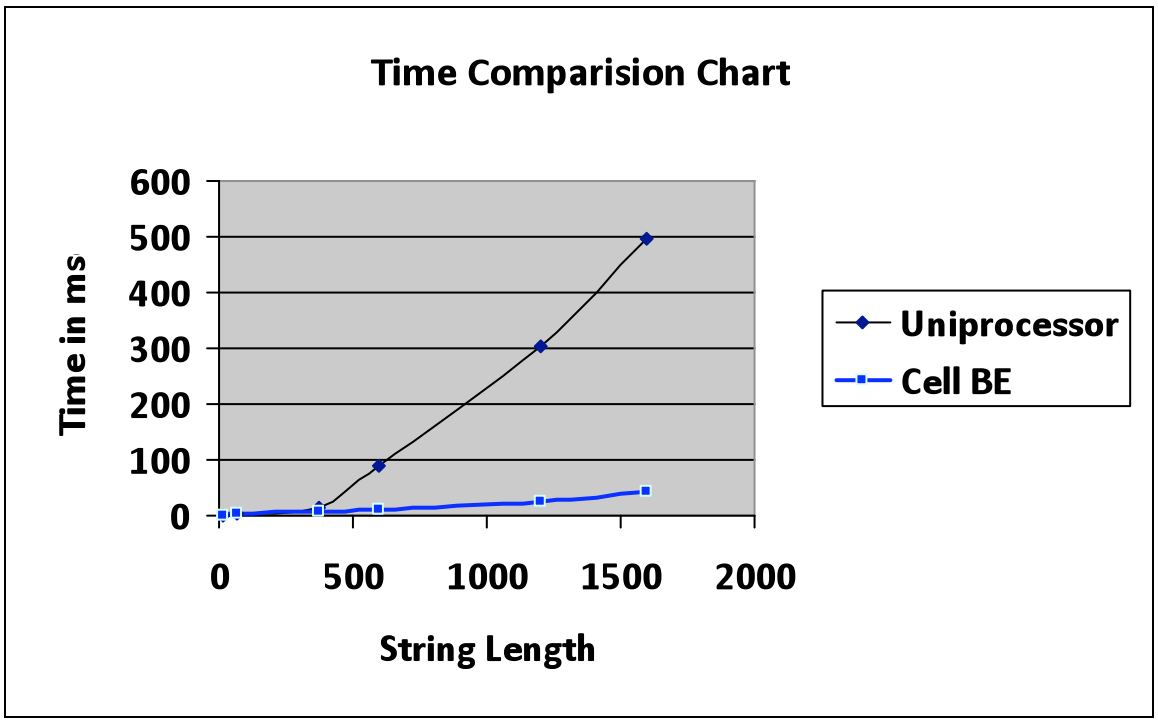


Figure 19 Time Comparison between Uniprocessor and Cell BE

Table 8 Observation Table for Applications

String Length	18	70	370	600	1200	1600
Uniprocessor (time in ms)	**0.596**	**4.783**	**13.008**	**88.284**	**304.706**	**494.716**
Cell BE	**0.751**	**4.923**	**6.89**	**9.008**	**26.7**	**43.018**
Speedup	**0.794**	**0.97**	**1.88**	**9.8**	**11.41**	**11.5**

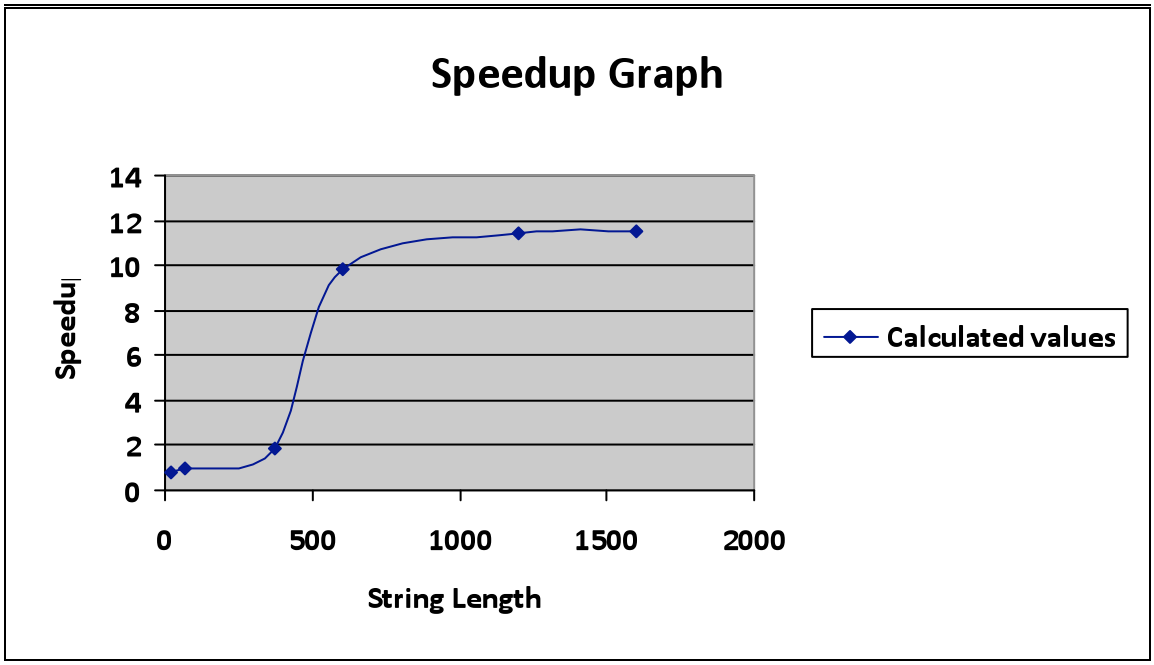


Figure 20 Speedup Graph for applications running in dual mode

Table 8 above shows all the observations when two simultaneous applications were run using dual mode and its comparison with the time it takes to run on a uniprocessor. It also shows the speedups obtained according to the formulae already discussed. The two graphs are plotted, one for time comparison and another for speedup against string length. As shown Figure 21, the application running in dual mode in Cell BE clearly outperforms that running on a uniprocessor. By Figure 22, it is observed that initially speedup is less than one because overhead cost is more. When length of the string is increased these overheads become negligible and the speedup becomes stable at some value after which on increasing string length a constant speedup is obtained.

`

Chapter 6

SUMMARY AND CONCLUSIONS

6.1 Summary

In this project, the parallelization techniques for suffix tree construction and applications on IBM Cell broadband engine have been proposed and successfully implemented. The main features of the project can be summarized as under:

In this project, we have implemented some improvements to the design of construction of suffix trees to make use of parallel processing power of the synergistic processor elements of the cell processor. Then we proposed and implemented a dual parallel construction mode to make use of all eight processors simultaneously to make maximum utilization of the processors even in case of bioinformatics applications where for one construction only four SPEs are generally active. All major applications namely String search, largest common substring, largest repeated substring and largest palindrome search showed significant speedups when generalized suffix trees were constructed in dual mode. For string length greater than 1000, speedup close to 11.5 is achieved.

6.2 Conclusions

The timing of this project was important as a boom is expected in the multiprocessor market in near future and algorithms need to be developed in time to make maximum utilization of the available processing power. The work has suggested and implemented methods for remarkable improvement in time requirements for current and future bioinformatics applications involving sequence matching. Although this work has been done with an aim to improve bioinformatics applications, but as suffix trees are useful in many other areas such as data compression and document search using word trees, the speedups achieved can be further useful in these areas.

`

REFERENCES

[1] CBE_Tutorial_v2.0_15

[2] Cell Broadband Engine Workshop Slides.

[3] Cell Broadband Engine Programming Handbook

[4] http://en.wikipedia.org/wiki/Suffix_tree - for basic understanding

[5] E. M. McCreight. 'A Space-Economical Suffix Tree Construction Algorithm'. Jrnl. of Algorithms, **23**(2) pp262-272, 1976.

[6] E. Ukkonen. 'Constructing Suffix Trees On-Line in Linear Time'. In *Algorithms, Software, Architecture*, J.v.Leeuwen (ed), vol#1 of Information Processing 92, Proc. IFIP 12th World Computer Congress, Madrid, Spain, Elsevier Sci. Publ., pp484-492, 1992.

[7] E. Ukkonen. 'On-line Construction of Suffix Trees'. Algorithmica, 14(3) pp249-260, 1995.

[8] P. Weiner. 'Linear Pattern Matching Algorithms'. Proc. 14th IEEE Annual Symp. on Switching and Automata Theory, pp1-11, 1973.

[9] D.E. Knuth, The art of computer programming, volume 3: (2nd ed.) sorting and searching, Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, 1998.

[10] D.E. Knuth, Morris, J.H. JR., and V.R.Pratt 'Fast pattern matching in strings'. Comput. Sei. Rep. STAN-CS-74-440, Stanford U., Stanford, Calif., Aug. 1974.

[11] U.Manber & G.Myers (1993). 'Suffix arrays: A new method for on-line string searches'. SIAM Journal on Computing, 22(5), 935—948.

[12] J. Ian Munro, Venkatesh Raman, S. Srinivasa Rao (1998). 'Space Efficient Suffix Trees'.

[13] Alfred V. Aho , John E. Hopcroft, The Design and Analysis of Computer Algorithms, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1974.

[14] Ramesh Hariharan, 'Optimal parallel suffix tree construction', Journal of Computer and System Sciences, v.55 n.1, p.44-69, Aug. 1997.

`

APPENDIX

Code segment examples for mailboxes:

PPE code

```
//Write "data" to spe_id[i] mailbox
while(!spe_stat_in_mbox(spe_id[j]));
spe_write_in_mbox(spe_id[j], data);

//Read message sent from spe_id[i]
while(!spe_stat_out_mbox(spe_id[j]));
data = spe_read_out_mbox(spe_id[j]);
```

SPE code

```
//Read message sent from PPE
while(!spu_stat_in_mbox());
data = spu_read_in_mbox();

//Write "data" to PPE
while(!spu_stat_out_mbox());
spu_write_out_mbox(data);
```