PERFORMANCE IMPROVEMENT OF PARALLEL PATTERN MAT[redacted]GORITHM USING CELL B[redacted]D ENGINE

*mgcl#012

A DISSERTATION

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

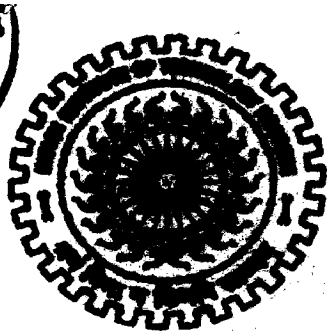
MASTER OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

By

RAJARSHI CHOWDHURY



DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY ROORKEE
ROORKEE - 247 667 (INDIA)
JUNE, 2008

CANDIDATE'S DECLARATION

I hereby declare that the work, which is being in this dissertation report, entitled **"Performance Improvement Of Parallel Pattern Matching Algorithm using Cell Broadband Engine"**, is being submitted in partial fulfillment of the requirements for the award of the degree of **Master of Technology** in **Computer Engineering**, in the Department of Electronics and Computer Engineering, Indian Institute of Technology, Roorkee is an authentic record of my own work, carried out from June 2007 to May 2008, under guidance and supervision of **Dr. Ankush Mittal**, Associate Professor and **Dr. Rajdeep Niyogi**, Assistant Professor, Department of Electronics and Computer Engineering, Indian Institute of Technology, Roorkee.

The results embodied in this dissertation have not submitted for the award of any other Degree or Diploma.

Date: 05.06.2008

Place: Roorkee

Rajarshi Chowdhury

(Rajarshi Chowdhury)

CERTIFICATE

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

Dr. Ankush Mittal

Associate Professor

Dr. Rajdeep Niyogi

Assistant Professor

ACKNOWLEDGEMENT

At the outset, I express my heartfelt gratitude to Dr. Ankush Mittal, Associate. Professor and Dr. Rajdeep Niyogi, Assistant Professor, Department of Electronics and Computer Engineering at Indian Institute of Technology Roorkee, for their valuable guidance, support, encouragement and immense help. I consider myself extremely fortunate for getting the opportunity to learn and work under their able supervision. I have deep sense of admiration for their innate goodness and inexhaustible enthusiasm. It helped me to work in right direction to attain desired objectives. Working under their guidance will always remain a cherished experience in my memory and I will adore it throughout my life.

My sincere thanks are also due to rest of the faculty in the Department of Electronics and Computer Engineering at Indian Institute of Technology Roorkee, for the technical knowhow and analytical abilities they have imbibed in us which have helped me in dealing with the problems I encountered during the project.

I am thankful to Mr. Ravi Gupta, research scholar in my department for his constant encouragement in the initial stages of my work. I am grateful to Mr. Vinay Kumar, Mr. Avinash Sharma and Mr. C. Shekar, my colleagues for being excellent peers and creating a congenial environment for work.

I am greatly indebted to all my friends, who have graciously applied themselves to the task of helping me with ample morale support and valuable suggestions. Finally, I would like to extend my gratitude to all those persons who directly or indirectly helped me in the process and contributed towards this work.

The pleasure of nearing completion of the course requirements is immense, but with it carries the pain of leaving behind these wonderful two years of life in the sprawling green campus of this great historical institute. I am proud for being the student of this reputed institute.

I dedicate this work to my family for his support and encouragement throughout my life.

Rajarshi Chowdhury
M. Tech. (CSE)

ABSTRACT

A pattern matching algorithm is used to find the presence of the pattern in a given block of data. Pattern matching is used to test whether things have a desired structure, to find relevant structure, to retrieve the aligning parts, and to substitute the matching part with something else. Sequence (or specifically text string) patterns are often described using regular expressions (i.e. backtracking) and matched using respective algorithms. Sequences can also be seen as trees branching for each element into the respective element and the rest of the sequence, or as trees that immediately branch into all elements. Pattern matching is useful in the field of text editing, highly computation intensive works like bioinformatics (pattern matching in biological sequence databases or amino acid sequence databases), networking (high-speed intrusion detection system), syntax analysis, operating systems, internet related searches to name a few.

Sequential pattern matching algorithms have almost reached their limits in terms of performance improvement. They are already linear in time complexity and the effective decrement in number of character comparisons is too less. In our work we first analyze most of the existing sequential and parallel pattern matching algorithms in terms of complexity, character comparisons and time of execution. Based on our analysis we try to parallelize the fastest sequential pattern matching algorithm – TVSBS to study the performance improvement. We use IBM Cell-Broadband Engine to implement our algorithm.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

A	Adenine
A	Alanine
C	Cysteine
C	Cytosine
D	Aspartic acid
DNA	Deoxyribonucleic Acid
F	Phenylalanine
G	Glutamic acid
ESMA	Exact String Matching Algorithm
H	Histidine
I	Isoleucine
G	Guanine
K	Lysine
L	Leucine
M	Methionine
N	Asparagine
P	Proline
Q	Glutamine
R	Arginine
S	Serine
T	Threonine
V	Valine
RNA	Ribose Nucleic Acid
W	Tryptophan
T	Thymine
Y	Tyrosine

Chapter 1

INTRODUCTION

Pattern matching algorithms are used to find one or more occurrence of the given pattern in a given text. There are many sequential pattern matching algorithms having various complexities and efficiencies. But if we run these algorithms on bioinformatics or biomedical databases, they take up huge processing time and cost because of the size of the database. So, to improve the performance of pattern matching algorithms they were made to be executed in parallel. Multi processor systems are costly solutions for this parallelization. Multi-core systems are able to deliver better solution in a cost effective way. As all the commercial processors turn multi-core [1] our goal is to optimize the pattern matching problem for these systems. Cell Broadband Engine is a new 9-core heterogeneous multi-core processor from IBM. The architecture of Cell Broadband Engine proposes tremendous improvement over its other counterparts. We provide a performance comparison for our implementation of parallelized pattern matching in Cell Broadband Engine.

The minimum number of comparisons needed for pattern matching is still an open problem. Derivations of new and better algorithms for pattern matching are still possible but chances are rare that any new algorithm will drastically improve performance over the previous best. This has been evident for the last few years. The best algorithm so far, TVSBS [2], improves over SSABS only in number of comparisons – not in time or space complexity. The only way to make things fast is to exploit parallelism in a multi-processor or in a multi-core environment. Parallel String matching algorithms based on dataflow architecture is in existence from as early as 1999. Before that PRAM algorithms for parallel pattern matching also had been developed that produced a Boolean array $MATCH[1 \ldots N]$ as output. The Boolean array contained a true value at each position where an occurrence of the pattern started. CRCW-PRAM was one of the pioneers in these types of algorithms. Later Galil algorithm [3] produced the best result, $O(\log m)$ in case of constant size alphabets. It has been also proved that the lower bound on the number of

comparisons in case of a parallel string matching algorithm is $O(\log\log m)$ for pattern $P[1 \ldots m]$ in text $T[1 \ldots 2m]$. The optimal algorithm for parallel multiple pattern matching works in a complexity $O(\log L)$ where each pattern is of length L. Also algorithm for 2-D parallel pattern matching exists that works in $O(n^2)$ [4].

Recent works in parallel pattern matching [5] are in fields of high-speed intrusion detection system [6] and source-level programming [7] among others. Still now very less work has been done in this problem that exploits the capabilities of either homogeneous or heterogeneous multi-core architecture. An algorithm that takes advantage of this has the potential to deliver an improved running time for the pattern matching problem. In this paper we propose a parallelized version of TVSBS [2] implemented in IBM Cell Processor and the performance gain due to the parallelization. The preprocessing phase of the algorithm is performed serially whereas the searching phase is performed in parallel. A performance comparison of non-parallel algorithms in the same machine (using only 1 core) is given and the execution time gain is plotted in case of the parallelized version.

1.1 Pattern Matching Algorithms

Some broad categories of pattern matching are primitive pattern matching, tree pattern matching, multiple pattern matching and 2D pattern matching. Our study concerns the primitive pattern matching algorithms. The general behavior of sequential pattern matching algorithm is based on comparison of characters one at a time. It includes two distinct steps. The first is aligning the pattern against a given text (the aligned portion, known as **window**) and comparison among the characters of the pattern and the window. In case of a mismatch, the next step is to reposition the window by shifting it one character to the right and repeating the first step. The worst case complexity of this procedure is $O(mn)$. In order to improve the complexity, all the algorithms try to pre-calculate a shift value table and use that while shifting. This introduces a new pre-processing step but decreases the overall complexity to linear order, typically $O(m+n)$. These approaches are based on finite state automaton. However there are algorithms that make the character comparisons from left-to-right and algorithms that make the same from right-to-left.

1.1.1 Sequential Pattern Matching

We formally define a class of sequential pattern matching algorithms that includes all variations of Morris-Pratt algorithm [18]. For the last twenty years it was known that the complexity of such algorithms is bounded by a linear function of the text length. Recently, substantial progress has been made in identifying lower bounds. But the lower bound or in other words the minimum number of comparisons needed for sequential pattern matching algorithms is still an open problem.

1.1.2 Parallel Pattern Matching

We describe a parallel algorithm that finds all occurrences of a pattern string in a subject string in O(log n) time, where n is the length of the subject string. The number of processors employed is of the order of the product of the two string lengths.

1.1.3 Multi-Dimensional Pattern Matching

Multi-dimensional pattern matching deals with type of text and pattern having more than one dimension. Normally multi-dimensional pattern matching extends to 3 dimensions.

1.2 Problem Statement

The pattern matching problem can be formally defined as follows: given a pattern $P \in \Sigma^*$ with length $|P| = m$ and text string $T \in \Sigma^*$ with length $|T| = n$, where $m, n > 0$ and $m \leq n$ if P occurs as a substring of T, find the first occurrence, that is find s such that $T[s+1 \ldots s+m] = P[1 \ldots m]$ $(0 \leq s \leq n-m)$. There has been many paradigm shifts toward the solution of the problem. Also new type of pattern matching problems came up due to special need of other application areas. To improve the performance of pattern matching algorithms they were made to be executed in parallel.

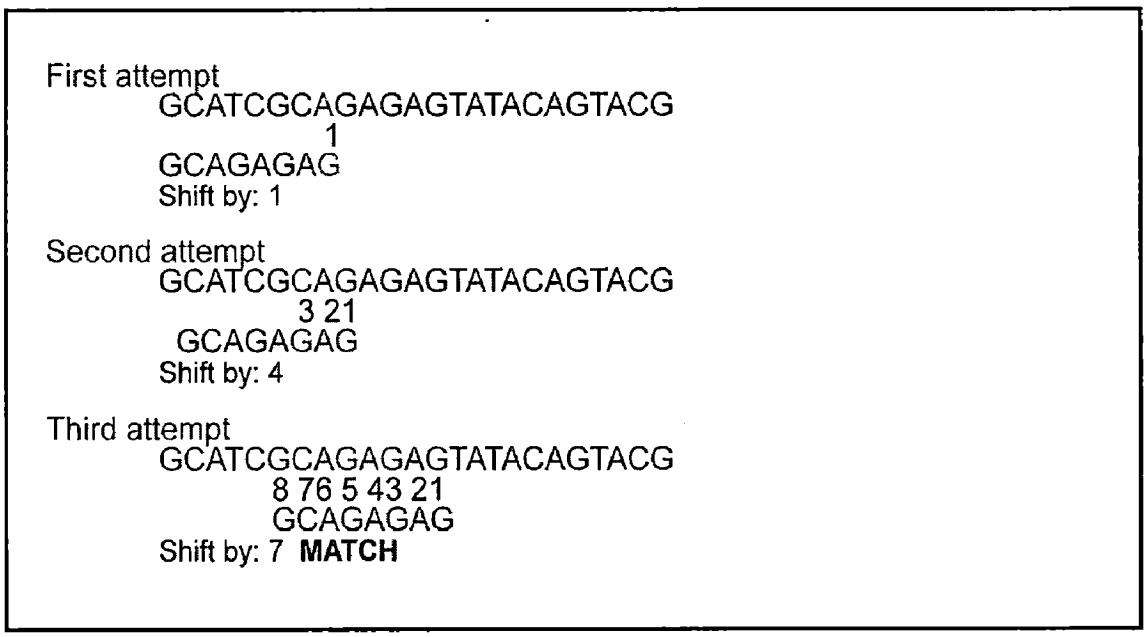


Figure 1.1: Example of Pattern Matching using Boyer-Moore Algorithm

1.3 Organization of the Report

The rest of the report is organized as follows. In chapter 2 we analyze the existing sequential and parallel pattern matching algorithms in a common framework and in a common dataset. We also provide a graphical representation of our result to show which algorithm performs the best among those discussed. Chapter 3 deals with the preliminaries of patterns in bioinformatics and IBM Cell Broadband Engine Architecture. In chapter 4 we discuss the methodology applied to parallelize pattern matching algorithm for Cell Broadband Engine. Chapter 5 consists of the implementation details, data sets and results. We conclude the report in chapter 6.

Chapter 2

ANALYSIS OF EXISTING PATTERN MATCHING ALGORITHMS

The general behavior of sequential pattern matching algorithm is based on comparison of characters one at a time. It includes two distinct steps. The first is aligning the pattern against a given text (the aligned portion, known as **window**) and comparison among the characters of the pattern and the window. In case of a mismatch, the next step is to reposition the window by shifting it one character to the right and repeating the first step. The worst case complexity of this procedure is $O(mn)$. In order to improve the complexity, all the algorithms try to pre-calculate a shift value table and use that while shifting. This introduces a new pre-processing step but decreases the overall complexity to linear order, typically $O(m+n)$. These approaches are based on finite state automaton. However there are algorithms that make the character comparisons from left-to-right and algorithms that make the same from right-to-left.

We now discuss some algorithms that perform the character comparison from left-to-right. The first is the Baeza et al. [14] algorithm (also known as Shift-and and Shift-Or algorithm) that pre-computes a set of bitmasks containing one bit for each element of the pattern and then does the rest of the work with bitwise operations. The algorithm is used in UNIX command "grep" as it also has the potential for "approximately equal" matches. Following the brute-force approach, there is Morris-Pratt algorithm and following that is the Knuth et al. [KMP] algorithm that works by pre-calculating the required shift value in case a mismatch occurs. For a long time KMP algorithm was the fastest algorithm for single-dimensional exact pattern matching. After KMP's success many algorithms refined the KMP-way and reduced effective number of comparisons. These include Apostolico and Crochemore algorithm [22], Not-so-naïve algorithm, DFA algorithm and Simon Algorithm [16]. Apostolico and Crochemore algorithm decreased the number of failure attempts to reduce character comparisons. Not-so-naïve algorithm followed the searching behavior of Apostolico and Crochemore algorithm to improve the performance of

brute-force algorithm. The DFA algorithm used finite automaton techniques and the Simon algorithm that improves over DFA.

Next we discuss the right-to-left type of algorithms. The first algorithms in this category is Boyer and Moore algorithm that introduced the idea of "bad-character shift" and "good-suffix shift". The algorithms that improved Boyer and Moore [8] were Tuned-BM, Turbo-BM , Apostolico and Giancarlo , Quick-Search and Zhu and Takaoka . Improving the time-complexity of Quick-Search algorithm was SSABS algorithm and Berry and Ravindran algorithm [11]. The latest and best algorithm in this type of pattern matching uses the concepts of SSABS and Berry and Ravindran algorithm [18] and is known as TVSBS algorithm. TVSBS has a time complexity of $O(n/(m+2))$ and performs at most $O(m(n-m+1))$ character comparisons in the worst case.

There are also other algorithms that do not fall into any of these two prominent categories. This class includes examples like Colussi algorithm, Two Way algorithm, String Matching on Ordered alphabets, Horspool Algorithm, Smith algorithm, Raita algorithm to mention a few.

2.1 Complexity Comparison of Sequential Pattern Matching Algorithms

Brute-force exact pattern matching algorithm has a time complexity of $O(mn)$ whereas Morris-Pratt [10] algorithm has a linear time complexity of $O(m+n)$. We find that the number of character comparisons performed by these two algorithms we shall see that Brute-force performs at most $2n$ comparisons and Morris-Pratt performs at most $2n-1$ comparisons. Other algorithms that operate in similar number of character comparisons are Boyer-Moore ($3n$ chracter comparisons), Apostolico-Crochemore ($3/2n$ character comparisons) and Colussi ($3/2n$ character comparisons). Some other algorithms like quick search and skip-search has a quadratic time complexity at the worst case. Table 2.1 gives an overview of complexities of the most important pattern matching algorithms. TVSBS is derived upon the fact that Berry-Ravindran algorithm has the best pre-processing step, where as SSABS has the best searching phase. Figure 2.1 shows the performance

comparison among Brute-Force, Morris-Pratt, KMP, Colussi, AXAMAC, Boyer-Moore, Quick-Search, Skip-Search, SSABS. SSABS is the best among these. This graph uses the data achieved by S.S. Sheik et al. at Bioinformatics Centre (DIC), the Interactive Graphics Based Molecular Modelling facility (IGBMM) and the Supercomputer Education and Research Centre, IISc, Bangalore, using the old Swiss-Prott dataset. X-axis represents the length of the pattern and Y-axis represents time of processing in milliseconds.

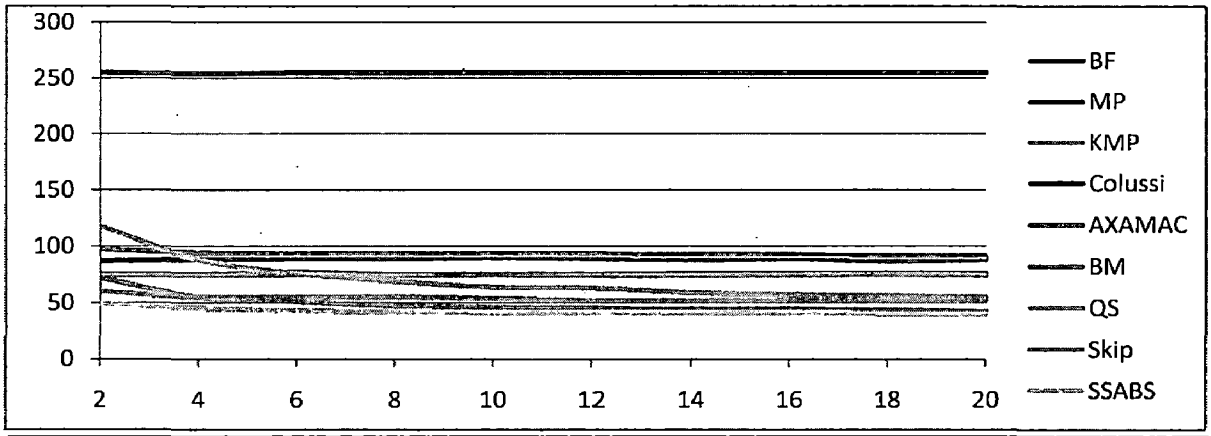


Fig.2.1 Graph showing the performance comparisons of algorithms Brute-Force (BF), Morris-Pratt [MP], KMP, Colussi, AXAMAC, Boyer-Moore [BM], Quick-Search [QS], Skip-Search, SSABS.

Table 2.1: Asymptotic performance comparison of exact string-matching algorithms

ESMAs	tc	sc	pt	cc
Brute Force Algorithm	O(mn)	constant extra space	no preprocessing	2n
Morris-Pratt Algorithm	O(n+m)	O(m)	O(m)	2n-1
Apostolico-Crochemore Algorithm	O(n)	O(m)	O(m)	3/2n
Boyer-Moore Algorithm	O(mn)	O(m +|Σ|)	O(m +|Σ|)	3n
Quick Search Algorithm	O(mn)	O(|Σ|)	O(m +|Σ|)	quadratic worst case

SSABS Algorithm	O([n/(m+1)])	-	-	O(m(n-m+1)) worst case
Zhu-Takaoka Algorithm	O(mn)	O(m+|Σ|^2)	O(m+|Σ|^2)	quadratic worst case
Berry-Ravindran Algorithm	O(mn)	O(m+|Σ|^2)	O(m+|Σ|^2)	-
TVSBS Algorithm	O([n/(m+2)])	O(|Σ|+k^|Σ|)	O(|Σ|+k^|Σ|)	O(m(n-m+1)) worst case
Colussi Algorithm	O(n)	O(m)	O(m)	3/2n
Skip Search Algorithm	O(mn)	O(m +|Σ|)	O(m +|Σ|)	O(n), quadratic worst case

2.2 Performance comparison of Pattern Matching Algorithms

We compared all the pattern matching algorithms for pattern lengths 4, 8, 12, 16 and 20 against alphabet sizes of 10 and 20. Different alphabet sizes show almost the same performance difference among different algorithms. From table 2.2 and table 2.3 we conclude that TVSBS is the best sequential pattern matching algorithm in most of the cases. We measured the times in milliseconds and calculated the relative scores with respect to TVSBS (reference frame 1). We measured time in milliseconds and the divided the time needed by other algorithms with the time needed by TVSBS algorithm. In this way we derived a ratio of performance of all the algorithms. Table 2.2 displays results of alphabet size 10 and Table 2.3 does so for alphabet size 20. We used **UniProt KnowledgeBase** for the experiment. The Machine Used was a PC with Intel Core2Duo E4300 (1.8 GHz) running on nForce 650i Ultra, 2 GB DDR2 800MHz RAM, GeForce 8600GT graphics processor. Standard GCC compiler with compiler optimization turned off was used. The Operating System was Fedora Core 5.

SSABS Algorithm	O([n/(m+1)])	-	-	O(m(n-m+1)) worst case
Zhu-Takaoka Algorithm	O(mn)	O(m+|Σ|^2)	O(m+|Σ|^2)	quadratic worst case
Berry-Ravindran Algorithm	O(mn)	O(m+|Σ|^2)	O(m+|Σ|^2)	-
TVSBS Algorithm	O([n/(m+2)])	O(|Σ|+k^|Σ|)	O(|Σ|+k^|Σ|)	O(m(n-m+1)) worst case
Colussi Algorithm	O(n)	O(m)	O(m)	3/2n
Skip Search Algorithm	O(mn)	O(m +|Σ|)	O(m +|Σ|)	O(n), quadratic worst case

2.2 Performance comparison of Pattern Matching Algorithms

We compared all the pattern matching algorithms for pattern lengths 4, 8, 12, 16 and 20 against alphabet sizes of 10 and 20. Different alphabet sizes show almost the same performance difference among different algorithms. From table 2.2 and table 2.3 we conclude that TVSBS is the best sequential pattern matching algorithm in most of the cases. We measured the times in milliseconds and calculated the relative scores with respect to TVSBS (reference frame 1). We measured time in milliseconds and the divided the time needed by other algorithms with the time needed by TVSBS algorithm. In this way we derived a ratio of performance of all the algorithms. Table 2.2 displays results of alphabet size 10 and Table 2.3 does so for alphabet size 20. We used **UniProt KnowledgeBase** for the experiment. The Machine Used was a PC with Intel Core2Duo E4300 (1.8 GHz) running on nForce 650i Ultra, 2 GB DDR2 800MHz RAM, GeForce 8600GT graphics processor. Standard GCC compiler with compiler optimization turned off was used. The Operating System was Fedora Core 5.

The 2nd column of the 2nd row in table 2.2 says that the brute-force algorithm takes a time of 4.59 units in case of UniProt knowledgebase with alphabet length 10 and pattern length 4 if the TVSBS takes 1 unit of time.

Table 2.2: Performance comparison of ESMAs. Database Used: UniProt Knowledgebase. Alphabet size: 10

	Pattern Length				
Name	**4**	**8**	**12**	**16**	**20**
Brute Force	4.59	4.85	4.91	4.86	4.81
Moriss – Pratt	1.66	1.74	1.61	1.88	1.68
Knuth – Morris – pratt	1.71	1.77	1.81	1.61	1.79
COLUSSI	1.69	1.81	1.86	1.71	1.69
Galil – Giancarlo	1.78	1.91	1.91	1.81	1.89
Apostolico – Crochemore	1.85	1.89	1.89	1.86	1.81
Boyre – Moore	1.33	1.19	1.27	1.28	1.31
Turbo – Boyre – Moore	2.20	1.94	2.17	2.01	2.25
Apostolico – Giancarlo	4.01	3.12	3.65	3.82	3.16
Reverse Colussi	1.64	1.35	1.46	1.56	1.35
HORSPOOL	1.40	1.28	1.28	1.47	1.26
TUNED Boyre – Moore	1.49	1.38	1.37	1.31	1.21
Quick Search	2.11	1.94	2.06	2.08	2.28
SMITH	2.16	1.89	1.95	2.08	1.83
Zhu – Takaoka	1.26	1.06	1.05	1.25	1.15
Berry – Ravindran	1.80	1.54	1.44	1.73	1.63
AUT	5.56	5.88	5.85	5.87	5.46
SIMON	2.14	2.24	2.18	2.28	1.95
Forward Dawg Matching	10.7	13.17	11.8	10.5	11.8
RF	3.71	2.68	2.95	3.18	3.85
Turbo Reverse Factor	4.45	3.25	4.28	3.85	4.18
Backward Oracle Matching	1.75	1.43	1.68	1.52	1.68
SKIP Search	1.91	1.91	1.98	2.08	1.95

KMPSKIP	1.55	1.52	1.48	1.58	1.46
BNDM	1.43	1.16	1.53	1.24	1.18
Karp – Robin	1.74	1.29	1.64	1.53	1.75
Shift-Or	1.10	1.84	1.85	1.54	1.93
Not-So-Naïve	1.82	1.91	1.48	1.93	1.34
RAITA	1.22	1.14	1.28	1.28	1.33
Galil – Seiferas	3.45	3.65	3.48	3.16	3.66
Two-Way	1.51	1.47	1.48	1.33	1.44
String Matching on Ordered Alphabets	2.55	2.68	2.68	2.73	2.68
Optimal Mismatch	1.28	1.17	1.28	1.08	1.18
Maximal Shift	1.52	1.14	1.28	1.48	1.66
SSABS	1.08	1.07	1.02	0.99	1.11
TVSBS	1	1	1	1	1

Table 2.3: Performance comparison of ESMAs. Database Used: UniProt Knowledgebase. Alphabet size: 20

	Pattern Length				
Name	**4**	**8**	**12**	**16**	**20**
Brute Force	4.49	4.53	4.34	4.53	4.34
Moriss – Pratt	1.45	1.74	1.61	1.88	1.68
Knuth – Morris – pratt	1.56	1.77	1.81	1.61	1.79
COLUSSI	1.13	1.81	1.86	1.71	1.69
Galil – Giancarlo	1.61	1.91	1.91	1.51	1.89
Apostolico – Crochemore	1.57	1.89	1.89	1.86	1.81
Boyre – Moore	1.8	1.19	1.36	1.28	1.31
Turbo – Boyre – Moore	2.12	1.94	2.17	2.01	2.25
Apostolico – Giancarlo	4.25	3.12	3.65	3.82	3.16
Reverse Colussi	1.64	1.35	1.57	1.56	1.35
HORSPOOL	1.40	1.28	1.1	1.47	1.26
TUNED Boyre – Moore	1.49	1.38	1.37	1.31	1.21
Quick Search	2.11	1.94	2.06	2.3	2.28

SMITH	2.6	1.89	1.1	2.08	1.83
Zhu – Takaoka	1.26	1.06	1.05	1.25	1.15
Berry – Ravindran	1.80	1.54	1.44	1.73	1.63
AUT	5.56	5.88	5.56	5.87	5.46
SIMON	2.14	2.24	2.18	2.28	1.95
Forward Dawg Matching	10.7	13.23	11.8	10.8	11.8
RF	3.71	2.68	2.77	3.18	3.85
Turbo Reverse Factor	4.45	3.25	4.28	3.85	4.18
Backward Oracle Matching	1.87	1.43	1.68	1.45	1.45
SKIP Search	1.91	1.91	1.45	2.08	1.95
KMPSKIP	1.55	1.8	1.86	1.58	1.46
BNDM	1.43	1.16	1.53	1.24	1.18
Karp – Robin	1.74	1.29	1.64	1.45	1.75
Shift-Or	1.21	1.84	1.78	1.54	1.56
Not-So-Naïve	1.82	1.56	1.48	1.93	1.34
RAITA	1.22	1.14	1.28	1.87	1.33
Galil – Seiferas	3.45	3.34	3.48	3.16	3.66
Two-Way	1.51	1.23	1.23	1.45	1.44
String Matching on Ordered Alphabets	2.55	2.68	2.68	2.73	2.68
Optimal Mismatch	1.28	1.17	1.28	1.08	1.18
Maximal Shift	1.52	1.14	1.28	1.45	1.66
SSABS	1.08	1.07	1.02	0.99	1.11
TVSBS	1	1	1	1	1

2.3 Parallelized Pattern Matching Algorithms

To provide optimal speedup to the pattern matching problems and to exploit the architectural efficiency of distributed systems, pattern matching problems were ported into parallel systems. Most of the parallel solutions to the pattern matching problems are due to specific applications. In 1991 it was first proposed and proved that the best parallel solution of pattern matching problem will have a lower bound of $\Omega(\log\log m)$ [4]. In the special case of a single processor, we have the classical

string matching problem. For this problem the algorithm due to Knuth, Morris, and Pratt (KMP77) preprocesses the pattern string in *O(M)* time and then processes each text set of size *N* in time *O(N)*. Clearly both the bounds are asymptotically optimal. Aho and Corasick [3] extended the approach in KMP77 and obtained an optimal algorithm for multiple pattern matching, that is, one that runs in *O(M)* preprocessing time and *O(N)* text processing time; their algorithm works in the more general case when the pattern strings are not necessarily of identical lengths.

We are concerned with parallel algorithms for pattern matching. For the special case of $k=1$, algorithms are known that are simultaneously time and work optimal for both the preprocessing and the text processing stages. However, unlike the sequential setting, their techniques cannot be extended to give optimal parallel algorithms [5] for the multiple pattern matching problem. This is because the notion of periodicity of a string that is crucial in existing parallel algorithms does not seem to extend naturally to multiple strings. Based on alternate strategies several efficient, but suboptimal, algorithms were designed for multiple pattern matching. These performed at least O(*N* log *L)* work for text processing. The known optimal algorithms for multiple pattern matching include deterministic and randomized variants.

Dataflow parallel approaches solve the exact matching and the k-mismatches problems with time complexities of $O((n / d) + \alpha)$, where $\alpha = \log m$ for the hierarchical scheme, m for the linear scheme, and 0 for the broadcasting scheme. Required time to process length n reference string is reduced by a factor of d by using d identical computation parts in parallel. With linear systolic array architecture, m PEs are needed for serial design and d*m PEs are needed for parallel design, where m is the pattern size and the d is the controllable degree of the parallelism (i.e. number of streams used).

2.4 Patterns in Bioinformatics

The past decade has witnessed an explosion of the amount and complexity of bioinformatics data such as DNA and protein sequences, gene and protein expressions, structures, pathways, genetic information, biomedical text data, and

molecular images. Although the analyses of these data involve pattern recognition and data mining, novel and efficient data analysis techniques are yet to be discovered to realize their true potential.

Bioinformatics is aimed at discovering knowledge from life sciences data with the aid of Information Technology, to find answers to unresolved problems in biology. One of the important discoveries of pattern recognition in bioinformatics is that specific patterns of our genomes and proteomes are able to tell our characters and how prone we are for certain diseases. In the coming years, medical practitioners will be able to personalize our medication by just looking at these patterns.

DNA molecules store the blueprint of cell function. Information stored in DNA, a chain of four nucleotides (A, T, G, and C), is first transcribed to mRNA and then translated to the functional form of life, proteins. The initiation of translation or transcription process depends on the presence of specific signals and patterns, referred to as motifs, present in DNA and RNA. Research on *in silico* detection of specific patterns of DNA sequences such as genes, binding sites, and promoters, leads to better understanding of molecular level function of a cell. Comparative genomics focus on comparison of different genomes to find conserved patterns or significant mutations over the evolution, which could possess some functional significance. Construction of evolutionary trees is useful to infer how genome and proteome are evolved and branch across species by ways of a complete library of motifs and genes.

A protein's functionality or interaction with other proteins is mainly determined by its 3-D structure. Prediction of protein's 3-D structure from its 1-D amino-acid sequence remains an important problem in structural genomics; protein-protein interactions are responsible for most molecular functions in living cells. Computational modeling and visualization tools of 3-D structures of proteins and interaction help biologists to infer cellular activities.

The challenge in functional genomics is to analyze gene expressions accumulated by microarray techniques to discover co-regulated genes and thereby

gene regulatory networks. Discovering and understanding how genes and proteins interact in specific pathways are gateways to systems biology. Molecular and cellular imaging provides techniques for *in vivo* sensing or imaging of cellular events such as movement of cells and subcellular localization of proteins. Potential techniques to fuse and integrate different types of life sciences data are yet to be realized.

The ever expanding knowledge of biomedical and phenotype data, combined with genotypes, is becoming difficult to be analyzed by traditional methods. Advanced data mining techniques, where the use of metadata for constructing precise descriptors of medical concepts and procedures, are required in the field of medical informatics. The vast amount of biological literature is posing new challenges in the field of text mining. These text mining techniques along with the aid of information fusion methods could help find pathways and interaction networks.

Today, high throughput and high content screening techniques allow biologists to gather data at an unprecedented rate. However, pattern recognition techniques to make inferences from these data are not evolving at a rate sufficient to meet the demand.

2.5 Applications of Pattern Matching

Though the classical problem and solution of pattern matching are applicable to a wide variety of applications ranging from Operating systems to Computer Networks, we are interested in its application in the field of bioinformatics. Here are some fields in bioinformatics where pattern matching can be useful:

1) Computational and comparative genomics
2) Functional genomics
3) Structural genomics and proteomics
4) Cheminformatics, chemigenomics
5) Systems biology, pathway analysis
6) Phylogenic analysis of species, sequences, structures, etc.
7) Immunoinformatics

Chapter 3

CELL BROADBAND ENGINE ARCHITECTURE

While chip multiprocessors (CMPs) have been touted as an approach to deliver increased performance, adoption had been slow because frequent scaling for uniprocessor-based design was continuing to deliver performance improvements. However, at the turn of the millennium, the diminishing returns of uniprocessor designs became painfully clear and we set out to leverage chip multiprocessing to deliver a significant performance boost over traditional uniprocessor-centric solutions.

Thus, a confluence of factors is leading to a surge in CMP designs across the industry. From a purely performance centric view, frequency scaling is running out of steam: technology-based frequency improvements are increasingly difficult, while the performance potential of deeper pipelining is all but exhausted. As demonstrated by Srinivasan *et al.*, the low power/performance efficiency of deep pipelining makes deeply pipelined designs unattractive under power dissipation constraints.

The emergence of chip multiprocessors is the effect of a number of shifts taking place in the industry: limited marginal returns on deep pipelining reduced benefits of technology scaling for higher frequency operation, and a power crisis making many "traditional" solutions non-viable. Another challenge for architects of high performance system include burgeoning design and verification complexity and cost, to find ways to translate the increased density of new CMOS technologies based on Dennard's scaling theory into delivered performance.

The situation in many ways mirrors the dawn of RISC architectures, and it may be useful to draw the parallels. Then as now, technological change was rife. The emerging large scale integration production enabled the building of competitive processors using a single chip, with massive cost reductions. Alas, the new technology presented constraints in the form of device count, limiting design

complexity and making a streamlined new class of architectures – microprocessors – a preferred class.

At the same time, pipelined designs showed a significant performance benefit. With the limited CAD tools available for design and verification at the time, this gave a significant practical advantage to simpler designs which were tractable with the available tools. Finally, the emergence of new compiler technologies helping to marshal the performance potential using instruction scheduling to exploit pipelined designs and performing register allocation to handle the increasingly severe disparity between memory and processor performance rounded out the picture.1 Then as now, innovation in the industry was reaching new heights. Where RISC marked the beginning of single chip processors, chip multiprocessors mark the beginning of single chip systems. This increase in new innovative solutions is a response to new constraints defying the established solutions, and giving new technologies an opportunity to overcome the incumbent technology's advantages in terms of optimization efforts.

When the ground rules change, high optimization often means that established technologies cannot respond to new challenges. Innovation starts slowly, but captures public perception in a short, sudden instant when the technology limitations become overbearing. Thus, while chip multiprocessors have conceptually discussed for over a decade, they have become the newest set of performance methods to deliver increasing system performance across a wide range of applications. Where a few years ago, the "treasure chest" of architecture methods seemed all but exhausted, with high-end solutions implementing all of them (pipelining, dynamic prediction, register renaming, out of order execution, multi-level cache hierarchies,. . .), the chip multiprocessor revolution is filling the treasure chest with new concepts.

3.1 Cell Broadband Engine

The Cell Broadband Engine was designed from ground up to address the diminishing returns available from a frequency-oriented single core design point by exploiting application parallelism and embracing chip multiprocessing. We refer to Kahle *et*

al.[19] for a detailed overview of the Cell Broadband Engine Architecture, and Hofstee for an analysis of Cell Broadband Engine power efficiency. Gschwind *et al.* gives an overview of the Cell Synergistic Processor architecture based on a pervasively data parallel computing (PDPC) approach, and Flachs *et al.* describes the SPU microarchitecture. To deliver a quantum leap in application performance in a power constrained environment, we decided to exploit application parallelism at all levels:

data level parallelism with pervasive SIMD instruction support,

instruction-level parallelism using a statically scheduled and power aware microarchitecture,

thread-level parallelism with a multi-core design approach, and

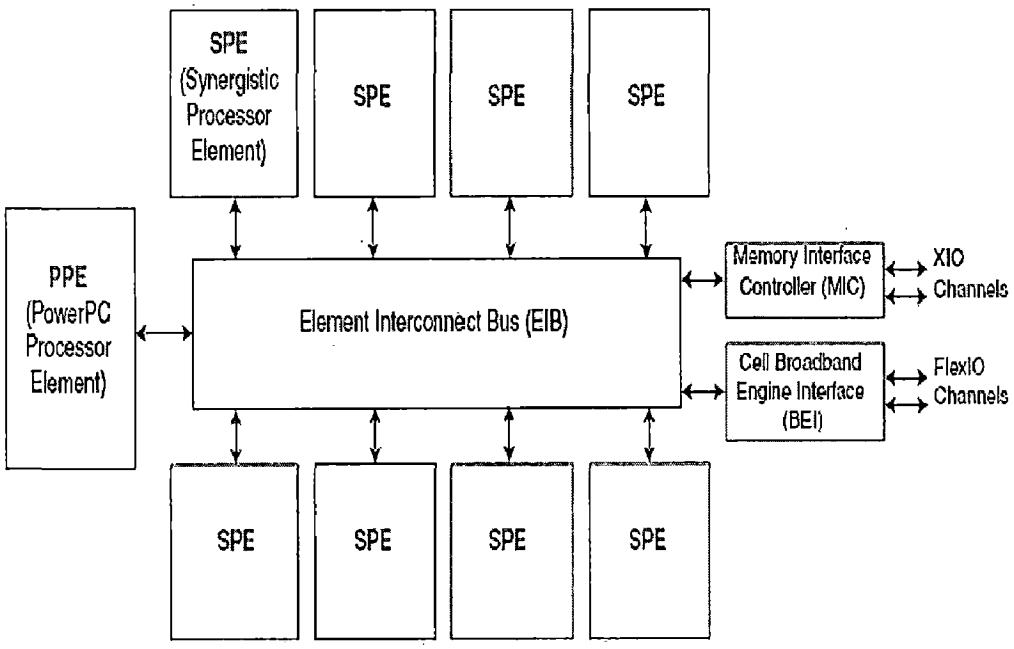


Figure 3.1 A Schematic diagram of CBEA

compute-transfer parallelism using programmable data transfer engines.

A key optimization is to deliver the best combination of parallelism degrees at each level, to ensure good utilization efficiency of the available resources by applications and to optimize system performance across the hardware and software stack, under area and power constraints. Data-level parallelism offers an efficient method to

increase the amount of computation at very little cost over a scalar computation. This is possible because the control complexity – which typically scales with number of instructions in flight – remains unchanged, i.e., the number of instructions fetched, decoded, analyzed for dependencies, the number of register file accesses and write backs, and the number of instructions committed remain unchanged.

Sharing execution units for both scalar and SIMD computation reduces the marginal power consumption of SIMD computation even further by eliminating control and datapath duplication. When using shared scalar/SIMD execution units, the only additional power dissipated for providing SIMD execution resources is dynamic power for operations performed, and static power for the area added to support SIMD processing, as the control and instruction handling logic is shared between scalar and SIMD data paths.

When sufficient data-parallelism is available, SIMD computation is also the most power efficient solution, because increase in power dissipation is for actual operations performed. Thus, SIMD power/performance efficiency greater than what can be achieved by multiple scalar execution units. Adding multiple scalar execution units duplicates control logic for each execution unit, and leads to increased processor complexity. This increased processor complexity is necessary to route the larger number of instructions (i.e., wider issue logic), to discover data-parallelism from a sequential instruction stream, plus potential data management (e.g., register renaming) and miss-speculation penalties incurred to rediscover and exploit data parallelism in a sequential instruction stream. Using a short 128b SIMD vector increases the likelihood of using a large fraction of computation units, and thus represents an attractive power/performance tradeoff.

Sharing of execution units for scalar and SIMD processing can be accomplished either architecturally, as in the Cell SPE, or microarchitecturally, as in the Cell PPE. Architectural sharing further increases efficiency of SIMD software exploitation by reducing data sharing cost. The Cell Broadband Engine also exploits instruction level parallelism with a statically scheduled power-aware multi-issue

microarchitecture. We provide statically scheduled parallelism between execution units to allow instruction dual-issue. Dual-issue is limited to instruction sequences which match the provisioned execution units of a comparable single-issue microprocessor. This limits in two respects [21]: (1) Instructions must be scheduled to match the resource profile as no instruction re-ordering is provided to increase the potential for multi-issue. (2) Execution units are not duplicated to increase multi-issue potential. While these decisions represent a limitation on dual issue, they imply that parallel execution is inherently power-aware. No additional reorder buffers, register rename units, commit buffers and similar structures are necessary, reducing core power dissipation.

Because the resource profile is known, a compiler can statically schedule instruction to the resource profile. Instruction level parallelism as used in the Cell Broadband Engine avoids the power inefficiency of wide issue architectures, because no execution units (and their inherent static and dynamic power dissipation) are added for marginal performance increase. Instead, parallel execution becomes energy-efficient because the efficiency of the core is increased by dual-issuing instructions: instead of incurring static power for an idle unit, the execution is performed in parallel, leading directly to a desirable reduction in energy-delay product.

As a first order approximation, let us consider energy to consist of the sum of energy per operation to execute all operations of a program $e$compute and a leakage power component dissipated over the entire execution time of the program $e$leakage . For normalized execution time $t = 1$, this gives a normalized energy delay metric of ($e$compute + $e$leakage).

By speeding up execution time using parallel execution, but without without adding hardware mechanisms or increasing the level of speculation, the energy-delay product is reduced. The new reduced execution time s, $s < 1$, is a fraction of the original (normalized) execution time t. The energy-delay product of power-aware parallel execution is ($e$compute +$e$leakage $\times s)\times s$. Note that both the energy and delay factors of the energy-delay product are reduced compared to non-parallel execution.

The total energy is reduced by scaling the leakage power to reflect the reduced execution time, whereas the energy *e*compute remains constant, as the total number of executed operations remains unchanged. In addition to speeding execution time by enabling parallel computation, ILP also can improve average memory latency by concurrently servicing multiple outstanding cache misses. In this use of ILP, a processor continues execution across a cache miss to encounter clusters of cache misses. This allows to concurrently initiate the cache reload for several accesses and overlap a sequence of memory accesses. The Cell BE cores support a stall on use policy which allows applications to initiate multiple data cache reload operations. While ILP provides a good vehicle to discover cache misses which can be serviced in parallel, it only has limited success in overlapping computation with the actual data cache miss service. Intuitively, instruction level parallelism can only cover a limited amount of the total cache miss service delay, a result confirmed by Karkhanis and Smith.

Thread-level parallelism (TLP) [20] is supported with a multithreaded PPE core and multiple SPE cores on a single Cell Broadband Engine chip. TLP delivers a significant boost in performance by providing ten independent execution contexts to multithreaded applications, with a total performance exceeding 200 GFLOPS. TLP is a key to deliver high performance with high power/performance efficiency, as described by Salapura *et al.*. To ensure performance of a single thread, we also exploit a new form a parallelism which we refer to as compute-transfer parallelism(CTP). To exploit memory more efficiently, compute-transfer parallelism considers data movement as an explicitly scheduled operation which can be controlled by the program to improve data delivery efficiency. Using application-level knowledge, explicit data transfer operations are inserted into the instruction stream sufficiently ahead of their use to ensure data availability and reduce program idle time. In the Cell Broadband Engine, bulk data transfers are performed by eight Synergistic Memory Flow controllers coupled to the eight Synergistic Processor Units.

Finally, to deliver a balanced CMP system, addressing the memory bottleneck is of prime importance to sustain application performance. Today, memory

performance is already limiting performance of a single thread. Increasing per-thread performance becomes only possible by addressing the memory wall head-on [19]. To deliver a balanced system design point with a chip multiprocessor, the memory interface utilization must be improved even more because memory interface bandwidth is growing more slowly than aggregate chip computational performance.

3.2 Cell Software Development Kit:

An SDK is available for the Cell Broadband Engine. The SDK contains the essential tools required for developing programs for the Cell Broadband Engine. The SDK consists of numerous components including the following:

- The IBM Full System Simulator for the Cell Broadband Engine, systemsim.
- system root image containing Linux execution environment for use within systemsim.
- GNU tools including c and c++ compilers, linkers, assemblers and binary utilities for both PPU and SPU.
- IBM xlc (c and c++) compiler for both PPU and SPU.
- newlib for the SPU. newlib is a C standard library designed for use on embedded systems.
- gdb debuggers for both PPU and SPU with support for remote gdb server debugging. The PPU debugger also provides combined, PPU and SPU, debugging.
- PPC64 Linux with CBE enhancements.
- SPE Runtime management library supporting SPE thread services - libspe. A next generation prototype SPE Runtime management, libspe2, is also provided.
- Static timing analysis timing tool, spu_timing, that instruments assembly source (either compiler or programmer generated) with expected instruction timing details.
- System wide profiler for Linux call oprofile.
- An Eclipse based Integrated Development Environment (IDE) to improve programmer productivity and integration of development tools.
- Standardized SIMD math libraries for the PPU's Vector/SIMD Multimedia Extension and the SPU.
- Example source code containing samples, libraries, workloads, and prototype tools. See the following section for more details.

Chapter 4

DESIGN OF PARALLELIZED PATTERN MATCHING FOR CELL BE

In this chapter we explain the workings of SSABS and TVSBS and how we derived our algorithm by modifying and parallelizing the above two.

4.1 SSABS

After a careful analysis of the existing algorithms, recently Sheik–Sumit–Anindya–Balakrishnan–Sekar (SSABS) proposed a new algorithm. The algorithm, SSABS, blends the advantages of QS and RAITA. In this algorithm, the order of character comparisons performed between the window and the search-string during each attempt is fixed. First, the rightmost characters of the window and the search string are compared. Secondly, the leftmost characters of the window and the search-string are compared, and then rests of the characters are compared in right to left order. In case of a mismatch in any one of the above-stated comparisons, the algorithm does not compare the remaining characters of the window. After either a match or a mismatch, the algorithm computes the shift of the window by finding the position of the bad character (character placed immediately after the window) in the search string. This shift value for all the characters in the alphabet are computed in the preprocessing phase and are used in the search phase. Hence, the algorithm SSABS is efficient and works well in most practical situations. We now deal with some simulated data to show the working process of this algorithm.

Preprocessing Phase (performed by qsBc)

Part of the sequence Considered for the Test Run.

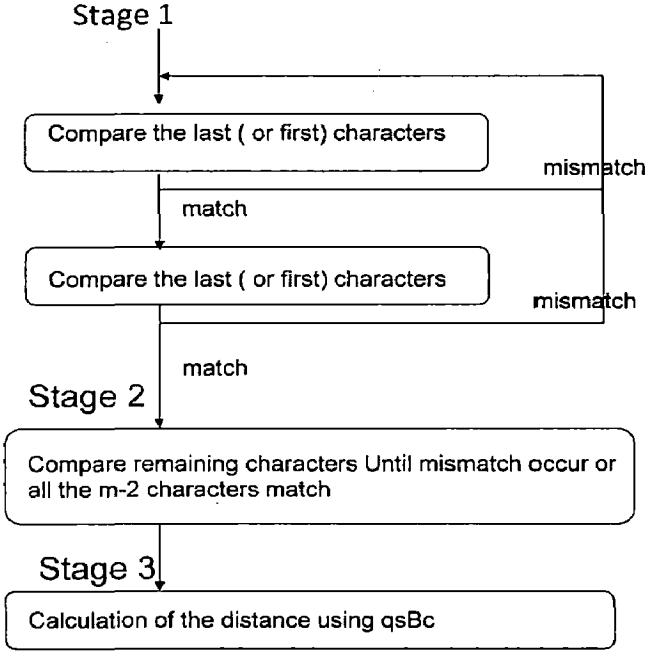
y (window) =
MARTKQTARKSTGGKAPRKQLATKAARKSAPSTGGVKKPHRYRPGTV

x (pattern) =
KAPRKQL

n = 47, m = 7, σ = 20

(SSABS)

Search Phase



Working Example (searching phase)

A	A	C	D	E	F	G	H	I	K	L	M	N	P	Q	R	S	T	V	W	Y
qsBc[a]	6	8	8	8	8	8	8	8	3	1	8	8	5	2	4	8	8	8	8	8

First attempt:

MARTKQTARKSTGGKAPRKQLATKAARKSAPSTGGVKKPHRYRPGTV

1

KAPRKQL

Shift = qsBc[A] = 6 // Stage 3- shifted by *j += qsBc[y [j + m]]*, j=0, m=7, y[7]

Second attempt:

6 13

MARTKQTARKSTGGKAPRKQLATKAARKSAPSTGGVKKPHRYRPGTV

1

KAPRKQL

Shift = qsBc[G] = 8 // Stage 3- shifted by *j += qsBc[y [j + m]]*, j=6, m=7, y[13]

Third attempt:

MARTKQTARKSTGGKAPRKQLATKAARKSAPSTGGVKKPHRYRPGTV

2 7 6 5 4 3 1

KAPRKQL

Shift = qsBc[A] = 6 // Stage 1 & 2 & 3 shifted by j += $qsBc[\, y\, [\, j + m\,]\,]$, j=14, m=7, y[21]

Fourth attempt:

MARTKQTARKSTGGKAPRKQLATKAARKSAPSTGGVKKPHRYRPGTV

1

KAPRKQL

Shift = qsBc[K] = 3 // Stage 3 shifted by j += $qsBc[\, y\, [\, j + m\,]\,]$, j=20, m=7, y[27]

Fifth attempt:

MARTKQTARKSTGGKAPRKQLATKAARKSAPSTGGVKKPHRYRPGTV

1

KAPRKQL

Shift = qsBc[P] = 5 // Stage 3 shifted by j += $qsBc[y[j + m]]$, j=23, m=7, y[30]

Sixth attempt:

MARTKQTARKSTGGKAPRKQLATKAARKSAPSTGGVKKPHRYRPGTV

1

KAPRKQL

Shift = qsBc[v] = 8 // stage 3 shifted by j += $qsBc[\, y[\, j + m]\,]$, j=28, m=7, y[35]

Seventh attempt:

MARTKQTARKSTGGKAPRKQLATKAARKSAPSTGGVKKPHRYRPGTV

1

KAPRKQL

Shift = qsBc[p] = 5 // stage 3 shifted by j += $qsBc[\, y\, [\, j + m]\,]$, j=36, m=7,
Compare until j< n - m, so stop

Total number of attempts: 7

Total number of character comparisons: 13

Figure 4.1 Working example of SSABS

4.2 TVSBS

As pointed out earlier, for a better performance, one needs to implement an efficient way of pre-processing the pattern to get a better shift value. Secondly, good methodology should be employed in the searching phase. TVSBS is a blend of Berry–Ravindran, and SSABS algorithms. The Berry–Ravindran bad character (hereafter, brBc) function is found to be effective during the preprocessing phase and the same has been implemented in the proposed algorithm with suitable modifications. The searching phase of this algorithm is exactly similar to that of the SSABS algorithm. The order of comparisons is carried out by comparing the last character of the window and that of the pattern first and once they match, the algorithm further compares the first character of the window and that of the pattern. This establishes an initial resemblance between the pattern and the window. The remaining characters are then compared from right to left until a complete match or a mismatch occurs. After each attempt, the skip of the window is gained by brBc shift value for the two consecutive characters immediately next to the window. The brBc function has been exploited to obtain the maximal shift and this reduces the number of character comparisons. These factors are collectively responsible for the improved performance of this algorithm.

- Working Example (searching phase)

brBC	A	C	G	T	*
A	10	10	2	10	10
C	7	10	9	10	10
G	1	1	1	1	1
T	10	10	9	10	10
*	10	10	9	10	10

- First attempt:

ATCTAAC AATCATAACCCTAATTGGCAGAGAGAGAATCAATCGAATCA

1

GCAGAGA G

Shift = brBc[T][C] = 10

- Second attempt:

ATCTAAC AATCATAACCCTAATTGGCAGAGAGAGAATCAATCGAATCA

1

GCAGAGAG

Shift = brBc[A][A] = 10

- Third attempt:

ATCTAAC AATCATAACCCTAATTGGCAGAGAGAGAATCAATCGAATCA

1

GCAGAGAG

Shift = brBc[G][A] = 1

- Fourth attempt:

ATCTAAC AATCATAACCCTAATTGGCAGAGAGAGAATCAATCGAATCA

2 1

GCAGAGAG

Shift = brBc[G][T] = 1

- Fifth attempt:

ATCTAAC AATCATAACCCTAATTGGCAGAGAGAGAATCAATCGAATCA

2 8 7 6 5 4 3 1

GCAGAGAG

Shift = brBc[A][G] = 2

- Sixth attempt:

28 35

ATCTAAC AATCATAACCCTAATTGGCAGAGAGAGAATCAATCGAATCA

2 1

GCAGAGAG

Shift = brBc[A][A] = 10

- Seventh attempt:

36

ATCTAAC AATCATAACCCTAATTGGCAGAGAGAGAATCAATCGAATCA

1

GCAGAGAG

- Total number of attempts: 7
- Total number of character comparisons: 16

Figure 4.2 Working example of TVSBS

4.3 Parallel Pattern Matching Algorithm for Multi-core systems

Combining the above two approaches along with the data-parallelism approach to solve algorithms parallel, we derive a new parallel algorithm for pattern matching. Though the algorithm itself is applicable to any kind of parallel system, we tested the algorithm against heterogeneous multi-core system. In our algorithm the preprocessing phase is modified Berry–Ravindran bad character function and the searching phase is modified SSABS in parallel. In the proposed algorithm we consider brBc over qsBc (Quick Search Bad Character) and bmBc (Boyer-Moore Bad Character) for the following reasons:

1. In qsBc the shift value is assigned for a character immediately next to the window, say **a**, based on the rightmost occurrence of that character. However brBc calculates the shift value based on the rightmost occurrence of two consecutive character, say **ab**, where b is the character next to a in the pattern, outside the window. The probability of the rightmost occurrence of ab in the pattern as compared to that of a, is very less. Therefore brBc always provides a better shift than qsBc or utmost an equal shift is obtained.
2. brBc value is always defined to be ≥ 1, and hence this could work independently to implement a fast algorithm, while bmBc yields a shift value ≤ 0 in some cases which requires the use of bmGs (Boyer-Moore Good Suffix) to calculate the skip of the window.

4.3.1 Algorithm without DMA

Serial Pre-processing phase:

Bad character function:

$$brBc[a,b] = min\begin{cases} 1 \;\; if\; x[M-1] = a, \\ m - i + 1 \;\; if\; x[i]x[i+1] = ab, \\ m + 1 \; if\; x[0] = b, \\ m + 2 \; otherwise \end{cases}$$

Parallel searching phase:

1) Let us define the degree of parallelism as p - 1 or m – 1, whichever is less, where p is the number of cores in a multi-core processor and m is the pattern length.
2) Compare the last character of the pattern with the last character of the window, the first one with first. Now compare 2nd 3rd … (m-1)th character of the pattern with the corresponding characters of the window. All of these comparisons are done in parallel. Each core does one comparison.
3) In case of a mismatch (found by any core), the information is passed to the controlling core and the next window location is calculated from the skip value table.

- Working Example (searching phase)

brBC	A	C	G	T	*
A	10	10	2	10	10
C	7	10	9	10	10
G	1	1	1	1	1
T	10	10	9	10	10
*	10	10	9	10	10

- First attempt:

ATCTAAC AATCATAACCCTAATTGGCAGAGAGAGAATCAATCGAATCA
2 3 4 5 6 7 8 1
GCAGAGA G
Shift = brBc[T][C] = 10 *Numbers represent cores*

- Second attempt:

ATCTAAC AATCATAACCCTAATTGGCAGAGAGAGAATCAATCGAATCA
2 3 4 5 6 7 8 1
GCAGAGAG
Shift = brBc[A][A] = 10

- Third attempt:

ATCTAAC AATCATAACCCTAATTGGCAGAGAGAGAATCAATCGAATCA

2 3 4 5 6 7 8 1

GCAGAGAG

Shift = brBc[G][A] = 1

- Fourth attempt:

ATCTAAC AATCATAACCCTAATTGGCAGAGAGAGAATCAATCGAATCA

2 3 4 5 6 7 8 1

GCAGAGAG

Shift = brBc[G][T] = 1

- Fifth attempt:

ATCTAAC AATCATAACCCTAATTGGCAGAGAGAGAATCAATCGAATCA

2 8 7 6 5 4 3 1

GCAGAGAG

Shift = brBc[A][G] = 2

- Sixth attempt:

28 35

ATCTAAC AATCATAACCCTAATTGGCAGAGAGAGAATCAATCGAATCA

2 3 4 5 6 7 8 1

GCAGAGAG

Shift = brBc[A][A] = 10

- Seventh attempt:

36

ATCTAAC AATCATAACCCTAATTGGCAGAGAGAGAATCAATCGAATCA

2 3 4 5 6 7 8 1

GCAGAGAG

- Total number of attempts: 7
- Total number of time units spent for character comparisons: 7 or less
- Maximum degree of parallelism: 8

Figure 4.3 Working example of Parallel Pattern Matching without DMA

4.3.2 Algorithm using DMA

Serial Pre-processing phase:

Bad character function:

$$brBc[a,b] = min\begin{cases} 1 \;\; if\; x[M-1] = a, \\ m - i + 1 \;\; if\; x[i]x[i+1] = ab, \\ m + 1 \; if\; x[0] = b, \\ m + 2 \; otherwise \end{cases}$$

Parallel searching phase:

1) Let us define the degree of parallelism as p - 1 or m – 1, whichever is less, where p is the number of cores in a multi-core processor and m is the pattern length.
2) Pre-assume that mismatch occurs and use **DMA Double Buffering** to the available processors/cores. (This step increases performance because in any given biological string number of mismatches is far more than number of matches.)
3) In case of a match (found by any core), the information is passed to the controlling core and the location is printed as output.

Working Example (searching phase)

brBC	A	C	G	T	*
A	10	10	2	10	10
C	7	10	9	10	10
G	1	1	1	1	1
T	10	10	9	10	10
*	10	10	9	10	10

First attempt:

ATCTAAC AATCATAACCCTAATTGGCAGAGAGAGAATCAATCGAATCA

1 1 1 1 1 1 1

GCAGAGA G

Shift = brBc[T][C] = 10

Numbers represent cores

Second attempt:

ATCTAAC AATCATAACCCTAATTGGCAGAGAGAGAATCAATCGAATCA

2 2 2 2 2 2 2

GCAGAGAG

Shift = brBc[A][A] = 10

- Third attempt:

ATCTAAC AATCATAACCCTAATTGGCAGAGAGAGAATCAATCGAATCA

3 33 3 3 3 3 3

GCAGAGAG

Shift = brBc[G][A] = 1

- Fourth attempt:

ATCTAAC AATCATAACCCTAATTGGCAGAGAGAGAATCAATCGAATCA

4 44 4 4 4 4 4

GCAGAGAG

Shift = brBc[G][A] = 1

- Fifth attempt:

ATCTAAC AATCATAACCCTAATTGGCAGAGAGAGAATCAATCGAATCA

5 5 5 5 5 5 5

GCAGAGAG

Shift = brBc[A][G] = 2

- Sixth attempt:

ATCTAAC AATCATAACCCTAATTGGCAGAGAGAGAATCAATCGAATCA

6 6 6 6 6 6 6

GCAGAGAG

Shift = brBc[A][A] = 10

- Seventh attempt:

36

ATCTAAC AATCATAACCCTAATTGGCAGAGAGAGAATCAATCGAATCA

7 7 7 7 7 7 7

GCAGAGAG

- Total number of attempts: 7
- Total number of time units spent for character comparisons: 1
- Maximum degree of parallelism: 8

Figure 4.4 Working example of Parallel Pattern Matching with DMA

Chapter 5

IMPLEMENTATION AND RESULTS

Here, we will discuss some of the implementation issues and different designing parameters. In addition, we will do the performance analysis with the help of experimental results.

5.1 Performance Parameters

We use accuracy and running time as the performance parameters to test Parallel Pattern Matching in cell processor. For accuracy comparison we compare the results achieved by our algorithm with the BM algorithm results.

The running time is defined as the time required identifying all the patterns in the text.

5.2 Data Set

We have tested our algorithm on both, the actual and simulated data.

5.2.1 Simulated Data

We used two stings to test our programs before testing them on actual biological data. The strings we used were:

ATCTAACAATCATAACCCTAATTGGCAGAGAGAGAATCAATCGAATCA

and

MARTKQTARKSTGGKAPRKQLATKAARKSAPSTGGVKKPHRYRPGTV

5.2.2 Actual biological Data

I. UniProt

Until recently, the EBI/SIB Swiss-Prot + TrEMBL databases and the PIR Protein Sequence Database (PIR-PSD) coexisted as protein databases with differing protein sequence coverage and annotation priorities. In 2002, EBI, SIB, and PIR (at the

Georgetown University Medical Center and National Biomedical Research Foundation) joined forces as the UniProt consortium. The primary mission of the consortium is to support biological research by maintaining a high quality database that serves as a stable, comprehensive, fully classified, richly and accurately annotated protein sequence knowledgebase, with extensive cross-references and querying interfaces freely accessible to the scientific community.

The UniProt Knowledgebase (UniProtKB) provides the central database of protein sequences with accurate, consistent, rich sequence and functional annotation.

The UniProt Knowledgebase consists of two sections: Swiss-Prot - a section containing manually-annotated records with information extracted from literature and curator-evaluated computational analysis, and TrEMBL - a section with computationally analyzed records that await full manual annotation.

Swiss-Prot is an annotated protein sequence database. It was established in 1986 and maintained collaboratively, since 1987, by the group of Amos Bairoch first at the Department of Medical Biochemistry of the University of Geneva and now at the Swiss Institute of Bioinformatics (SIB) and the EMBL Data Library (now the EMBL Outstation - The European Bioinformatics Institute (EBI)). The Swiss-Prot Protein Knowledgebase consists of sequence entries. Sequence entries are composed of different line types, each with their own format. For standardization purposes the format of Swiss-Prot follows as closely as possible that of the EMBL Nucleotide Sequence Database.

Swiss-Prot distinguishes itself from protein sequence databases by four distinct criteria:

a) Annotation

In Swiss-Prot, as in many sequence databases, two classes of data can be distinguished: the core data and the annotation.

For each sequence entry the core data consists of:

- The sequence data;
- The citation information (bibliographical references);
- The taxonomic data (description of the biological source of the protein).

The annotation consists of the description of the following items:

- Function(s) of the protein;
- Posttranslational modification(s) such as carbohydrates, phosphorylation, acetylation and GPI-anchor;
- Domains and sites, for example, calcium-binding regions, ATP-binding sites, zinc fingers, homeoboxes, SH2 and SH3 domains and kringle;
- Secondary structure, e.g. alpha helix, beta sheet;
- Quaternary structure, i.g. homodimer, heterotrimer, etc.;
- Similarities to other proteins;
- Disease(s) associated with any number of deficiencies in the protein;
- Sequence conflicts, variants, etc.

b) Minimal redundancy

Many sequence databases contain, for a given protein sequence, separate entries which correspond to different literature reports. In Swiss-Prot we try as much as possible to merge all these data so as to minimize the redundancy of the database. If conflicts exist between various sequencing reports, they are indicated in the feature table of the corresponding entry.

c) Integration with other databases

It is important to provide the users of biomolecular databases with a degree of integration between the three types of sequence-related databases (nucleic acid sequences, protein sequences and protein tertiary structures) as well as with specialized data collections. Swiss-Prot is currently cross-referenced to more than 50 different databases. Cross-references are provided in the form of pointers to information related to Swiss-Prot entries and found in data collections other than Swiss-Prot. This extensive network of cross-references

allows Swiss-Prot to play a major role as a focal point of biomolecular database interconnectivity.

d) Documentation

Swiss-Prot is distributed with a large number of index files and specialized documentation files. Some of these files have been available for a long time (this user manual, the release notes, the various indices for authors, citations, keywords, etc.), but many have been created recently and new files are being added continuously.

II. Nucleotide Sequence

We also tested our program on nucleotide sequences and amino acid sequences. A total of 837 gene sequences (comprising of nucleotides, 826.31 MB size) have been used to test the power of the proposed algorithm as was done with TVSBS. The data set contains 4-characters (nucleotides) viz., A(Adenine - 239490165), C(Cytosine - 183940124), G(Guanine - 183818044) and T(Thymine - 239419854) and hence the alphabet size is equal to 4. In order to avoid bias in the result the calculation was carried out for different pattern lengths the execution time was significantly reduced as shown in table 5.5.

III. Amino Acid Sequence

The case study with amino acid residues had a larger alphabet size of 20. Again as with TVSBS, we used 453861 gene sequences (191.24 MB). In this case the alphabet set used is A(13100890), C(1839722), D(8295604), E(9841468), F(6335049), G(10713539), H(3349835), I(9562897), K(8668206), L(15356872), M(3715491), N(6697619), P(6900621), Q(5838973), R(8414478), S(10200603), T(8319861), V(10559951), W(1837371), Y(4820702).

5.3 Implementation

The algorithms were tested in a PC having Intel Core2Duo E4300 CPU (1.8 GHz), 2GB Ram @ 800 MHz. The algorithms were coded in C Programming language and were run on Fedora Core 5 Operating System running over Windows Vista using

VMware Server 1.0.4. We used the IBM Cell Broadband Engine simulator for simulating the algorithm in a multi-core environment.

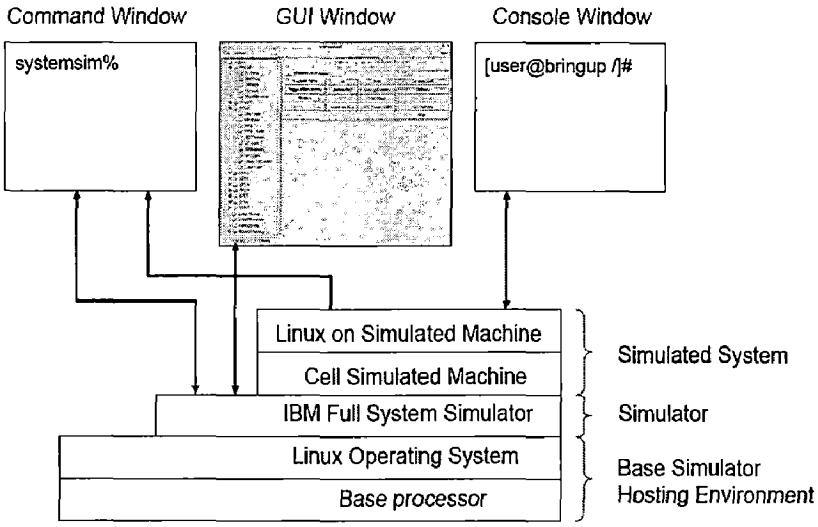


Figure 5.1: Implementation details

A multi-core system does not have much processing power/time wasted in communication. But still it is normally a significant factor. IBM cell processor is better than other processors in this aspect. We used DMA for communication among memory and synergistic processing cores' local stores inside the cell broadband engine. The programming language used was C and the compiler was GCC without any optimizations.

5.4 Accuracy Comparison

We compared the accuracy of our algorithm with the famous BM algorithm. For pattern length varying from 2 to 20 we applied both the algorithms to the same data sets. The results are shown in Table 5.1. The patterns found by our algorithm and BM was the same.

Table 5.1: Result of Accuracy comparison							
Sequence Length	4	8	12	16	20	30	40
Simulated data 1	100%	100%	100%	100%	100%	100%	100%
Simulated Data 2	100%	100%	100%	100%	100%	100%	100%
UniProt – Human*	100%	100%	100%	100%	100%	100%	100%
UniProt – Sprot*	100%	100%	100%	100%	100%	100%	100%
UniProt – tREmbl*	100%	100%	100%	100%	100%	100%	100%

*we tested with portions of these databases as they are very large and running BM on them usually takes much time.

5.5 Results for Actual Biological Data Set

Experiments were performed over the UniProt data set. UniProt data set consist of Swiss-Prot and tREmbl data sets. We removed the descriptions and documentations and performed out experiment on the sequence data. Table 5.1 shows the best results for UniProt dataset. Time units are in second. There was a 32 times improvement in terms of time units spent for the same dataset using our algorithm with DMA.

Table 5.2: Result of Our algorithms on UniProt Human Database					
Sequence Length	4	8	12	16	20
UniProt – Human* Algorithm with DMA	38	39	38.45	40.51	42.66
UniProt – Human* Algorithm without DMA	51	50.6	52	54.1	50

Table 5.3: Result of Our algorithms on Swiss-Prot Database									
Sequence Length	2	4	6	8	10	12	14	16	20
UniProt – Sprot* Algorithm with DMA	452	460	455	459	460	476	453	484	446
UniProt – Sprot* Algorithm w/o DMA	587	573	587	579	569	586	576	597	576

Sequence Length	2	4	6	8	10	12	14	16	20
UniProt – tREmbl* Algorithm with DMA	244	246	247	249	250	246	287	246	251
UniProt – tREmbl* Algorithm without DMA	321	316	324	302	321	310	319	317	320

Table 5.4: Result of Our algorithms on tREmbl Database

The following table shows the result obtained by our algorithm and TVSBS. It is clear from the above results and this that the proposed algorithm outperforms the others irrespective of the alphabet sizes.

Table 5.5: Result of Parallel Pattern Matching algorithm in nucleotide and amino acid databases

Sequence Length	4	8	12	16	20
Nucleotide sequence - TVSBS	1038	978	999	993	983
Nucleotide sequence – Algorithm with DMA	516	560	545	559	560
Nucleotide sequence – Algorithm without DMA	624	643	603	640	644
Amino Acid sequence - TVBSB	138	120	116	113	110
Amino Acid sequence – Algorithm with DMA	9	6.9	7.8	6.6	6.4
Amino Acid sequence – Algorithm without DMA	12	11.6	10	10.2	11

From the above two results we notice a slightly better speedup in case of the uniProt database than in the nucleotide and amino acid databases. The reason for this is the

inherent nature in the uniProt database. Our algorithm is based on bad character function and pre-assumption that a mismatch shall occur. As a result if a mismatch occurs then the algorithm works faster because of two reasons:

a) The comparison between the pattern and window took less time than a match.
b) The next window is already available because of DMA double buffering.

But as with the nucleotide and amino acid sequences the number of matches is higher than that of uniProt knowledgebase. As a result the running time increases because of the following reasons:

a) More matches means more time spent for comparison purposes.
b) The SPU has to flush the next DMA operation and output the result to the console. So one DMA cycle gets lost.

As a result of above factors the speedup achieved was varying between 20 times to maximum 32 times (as can be seen from the worked example in section 4.3). In worst case scenario also our algorithm performs better than any other sequential pattern matching algorithm on these biological datasets.

We also plotted the different execution time of our algorithm by disabling some vector processor cores and compared it with TVSBS in an uniprocessor machine. Also the difference in execution time due to DMA and DMA double buffering was noted. The following plots reflect our findings.

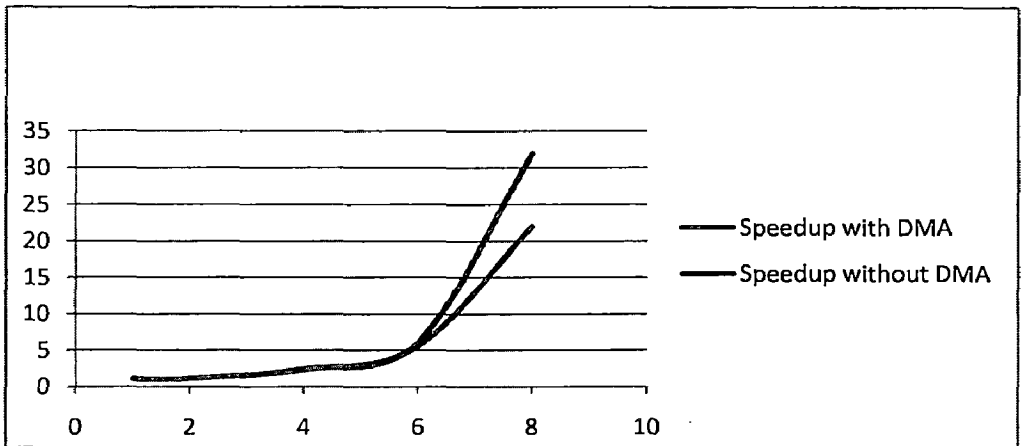


Figure 5.2 Graphical representation of speedup with respect to number of cores

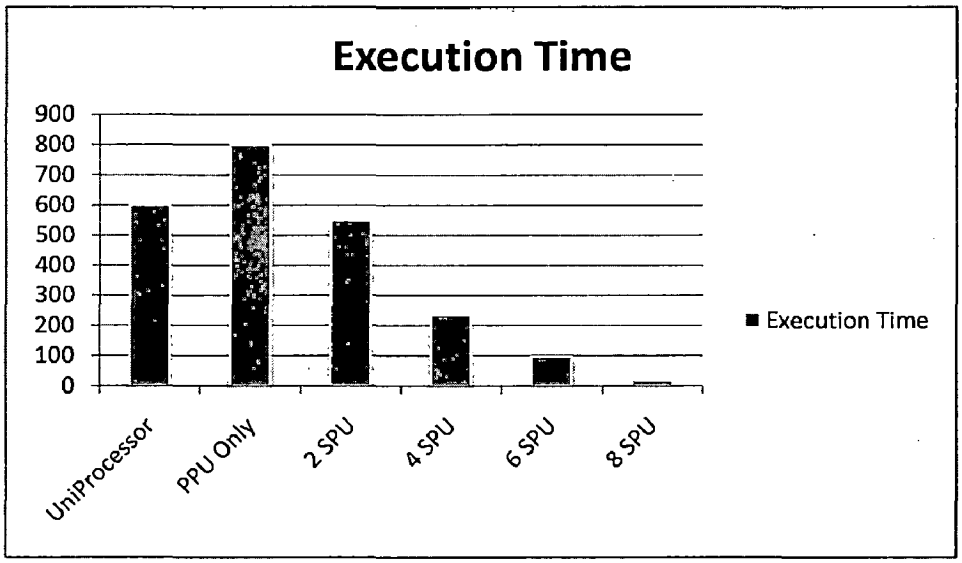


Figure 5.3: Result of execution of parallel pattern matching algorithm on CBEA in different situations. Y axis represents time in milliseconds

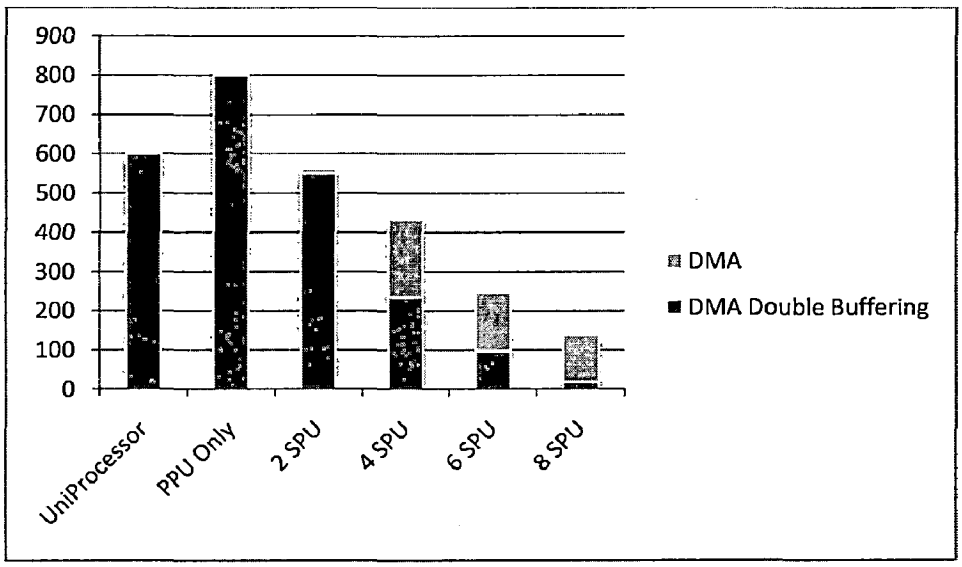


Figure 5.4: Comparative results of execution of parallel pattern matching algorithm using DMA and DMA double buffering on CBEA in different situations. Y axis represents time in milliseconds

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

In this thesis first we have shown through the statistics that although there is no theoretical lower bound in the number of comparisons to be performed for serial primitive pattern matching algorithms, any new algorithm will not be capable of improving the time complexity drastically. But new age pattern matching applications demand faster processing. The only way is to harness the strength of parallelization. We implemented the parallel version of one of the best pattern matching algorithm i.e., TVSBS in a heterogeneous multi-core processing environment i.e., the IBM Cell Broadband Engine. A significant speedup was achieved. The strategy proposed in the thesis can be extended to several other pattern matching algorithm to harness the computation gain by parallelization on Cell BE architecture.

6.2 Future Work

The following are some areas of future work that ensue from the thesis:

1. Repeat the experiment in other than bioinformatics datasets that are available.
2. Modify the DMA portion of the algorithm to make it suitable for other homogeneous and heterogeneous multi-core processors.
3. Rewrite the program in STAPL or similar language to test its effect on multi-processor environment.
4. Scale the program to be applicable to cell blade server or similar having 16 or more SPUs.

REFERENCES

[1] David, G. – Industry Trends: Chip Makers turn to multi-core processors. *Computer, 38(5):* pp.11-13,2005.

[2] Thathoo, R., Virmani, A., Sai Lakshmi, S., Balakrishnan, N., Sekar, K. TVSBS: A fast exact pattern matching algorithm for biological sequences. *Current Science*, 91(1), pp.47-53, 2006.

[3] Breslauer, D., Galil, Z. A Lower Bound for Parallel String Matching. *Proceedings of the twenty-third annual ACM symposium on Theory of computing,* pp. 439-443, 1991.

[4] Mongelli, H., Song, S. Efficient Two-Dimensional Parallel Pattern Matching With Scaling. *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms,* pp. 344-357, 1990.

[5] Muthukrishnan, S. Simple Optimal Parallel Multiple Pattern Matching. *Journal of Algorithms* 34, pp. 1-13, 2000.

[6] Lu, H., Zheng, K., Liu, B., Zhang, X., Liu, Y. A Memory-Efficient Parallel String Matching Architecture For High Speed Intrusion Detection, *IEEE Journal in selected areas in communication,* 24(10), pp.1793-1805, 2006.

[7] Mishra, J. Derivation of a Parallel String Matching Algorithm. *Information Processing Letters,* 85(5), pp. 255-260, 2003.

[8] Boyer, R., Moore, S. A Fast String Searching Algorithm. *Comm. ACM,* 10, pp. 762-772, 1977.

[9] Dany Breslauer, Zvi Galil. An Optimal O(log log n) Time Parallel String Matching Algorithm. *Siam J. Comput.*, 19, pp. 1051-1058, 1990.

[10] Coremen, T., Leiserson, C. Rivest, R. Stein, C. Introduction to Algorithms, 2nd Edition. *Prentice-Hall of India Pvt. Ltd.* pp.906-932, 2006.

[11] Lecroq, T. Experimental Results on String Matching Algorithms. *Software-Practice and Experience*. Vol. 25, pp. 727-765, 1995.

[12] Tarhio, J., Ukkonen, E. Approximate Boyer-Moore String Matching. *Siam J. Comput.*, Vol. 22, pp. 243-260, 1993.

[13] Mukherjee, A. Hardware Algorithms for Determining Similarity Between Two Strings. *IEEE Trans. On Computers*, 38(4), pp. 600-603, 1989.

[14] Yates, B., Gonnet, R., A New Approach to Text Searching. *Communications of the ACM.* 35(10), pp. 74-82, 1992.

[15] Apostolico A., Crochemore M. Optimal Canonization of all substrings of a string. *Information and Computation.* 95(1), pp. 76-95, 1991.

[16] Simon, I. String Matching Algorithms and Automata. *Results and Trends in Theoretical Computer Science*, pp. 386-395, 1994.

[17] Hwan Park, J., George, K. Parallel String Matching Algorithms Based on Dataflow. *Proceedings of the 32nd Hawaii International Conference on System Sciences,* pp.1-10, 1999.

[18] Christian Charras, Thierry Lecroq. Handbook of Exact String-Matching Algorithms, 2004 available at www-igm.univ-mlv.fr/~lecroq/string/string.ps

[19] Khale, J., Day, M., Hofstee, H., Johns, C., Maeurer, T., Shippy, D. Introduction to the Cell Microprocessor. *IBM Journal of Research and Development* [DOI: 10.1147/rd.494.0589]

[20] Gschwind, M., IBM T.J. Watson Research Centre, Erb, D., Sid Manning, and Nutter, M., IBM Austin. An Open-source Environment for Cell Broadband Engine System Software. *IEEE Computer Society,* pp.37-47, *2007.*

[21] Samuel W., Shalf, J., Oliker, L., Husbands, P., Kamil, S., Yelick K. The Potential of the Cell Processor for Scientific Computing. *Lawrence Berkeley National Laboratory, University of California* [Paper LBNL-59071], pp.9-20, 2006

[22] Crochemore, M. Off-line serial exact string searching, in *Pattern Matching Algorithms*, ed. A. Apostolico and Z. Galil, Chapter 1, pp 1-53, Oxford University Press,1997

[23] Crochemore, M., Hancart, C.Pattern Matching in Strings, in *Algorithms and Theory of Computation Handbook*, M.J. Atallah ed., Chapter 11, pp 11-1--11-28, CRC Press Inc., Boca Raton, FL,1999

PUBLICATIONS

1. Chowdhury, R., Niyogi, R., Mittal, A. A Performance Analysis of Sequential and Parallel Pattern Matching Algorithms – *Proceedings of National Conference on Algorithms (NCA'08)*, May 16-17, 2008

2. Chowdhury, R. Optimizing Performance of Pattern Matching Algorithms for Multi-Core Systems – *Proceedings of International Conference on Challenges and Developments in IT (ICCDIT'08)*, May 30, 2008

APPENDIX A

The following is the analysis and comparison of several pattern matching algorithms highlighting their limitations and their advantages.

1. Brute Force algorithm:

   - no preprocessing phase;
   - constant extra space needed;
   - always shifts the window by exactly 1 position to the right;
   - comparisons can be done in any order;
   - searching phase in $O(mn)$ time complexity;
   - $2n$ expected text characters comparisons.

2. DFA algorithm:

   - builds the minimal deterministic automaton recognizing the language $\Sigma^* x$;
   - extra space in $O(m\Pi)$ if the automaton is stored in a direct access table;
   - preprocessing phase in $O(m\Pi)$ time complexity;
   - searching phase in $O(n)$ time complexity if the automaton is stored in a direct access table, $O(n\log(\Pi))$ otherwise.

3. Karp-Rabin algorithm:

   - uses an hashing function;
   - preprocessing phase in $O(m)$ time complexity and constant space;
   - searching phase in $O(mn)$ time complexity;
   - $O(n+m)$ expected running time.

4. Shift Or algorithm:

   - uses bitwise techniques;
   - efficient if the pattern length is no longer than the memory-word size of the machine;
   - preprocessing phase in $O(m + \Pi)$ time and space complexity;
   - searching phase in $O(n)$ time complexity (independent from the alphabet size and the pattern length);
   - adapts easily to approximate string matching.

5. Morris-Pratt algorithm:

   - performs the comparisons from left to right;

- preprocessing phase in $O(m)$ space and time complexity;
- searching phase in $O(n+m)$ time complexity (independent from the alphabet size);
- performs at most $2n$-1 information gathered during the scan of the text;
- delay bounded by m.

6. Knuth-Morris-Pratt algorithm:

- performs the comparisons from left to right;
- preprocessing phase in $O(m)$ space and time complexity;
- searching phase in $O(n+m)$ time complexity (independent from the alphabet size);
- delay bounded by $\log \Phi(m)$ where Φ is the golden ratio ($\Phi = \frac{1+\sqrt{(5)}}{2}$).

7. Simon algorithm:

- economical implementation of $A(x)$ the minimal Deterministic Finite Automaton recognizing $\Sigma^* x$;
- preprocessing phase in $O(m)$ time and space complexity;
- searching phase in $O(m+n)$ time complexity (independent from the alphabet size);
- at most $2n$-1 text character comparisons during the searching phase;
- delay bounded by $min\{1 + \log_2 m, \Pi\}$.

8. Colussi algorithm:

- refinement of the Knuth, Morris and Pratt algorithm;
- partitions the set of pattern positions into two disjoint subsets; the positions in the first set are scanned from left to right and when no mismatch occurs the positions of the second subset are scanned from right to left;
- preprocessing phase in $O(m)$ time and space complexity;
- searching phase in $O(n)$ time complexity;
- performs $\frac{3}{2}n$ text character comparisons in the worst case.

9. Galil-Giancarlo algorithm:

- refinement of Colussi algorithm;
- preprocessing phase in $O(m)$ time and space complexity;
- searching phase in $O(n)$ time complexity;
- performs $\frac{4}{3}n$ text character comparisons in the worst case.

10. Apostolico-Crochemore algorithm:

- preprocessing phase in $O(m)$ time and space complexity;
- searching phase in $O(n)$ time complexity;

- performs $\frac{3}{2}n$ text character comparisons in the worst case.

11. Not So Naive algorithm:

- preprocessing phase in constant time and space;
- searching phase in $O(nm)$ time complexity;
- slightly (*by coefficient*) sub-linear in the average case.

12. Boyer-Moore algorithm:

- performs the comparisons from right to left;
- preprocessing phase in $O(m+\Pi)$ time and space complexity;
- searching phase in $O(mn)$ time complexity;
- $3n$ text character comparisons in the worst case when searching for a non periodic pattern;
- $O(n / m)$ best performance;

13. Turbo-BM algorithm:

- variant of the Boyer-Moore;
- no extra preprocessing needed with respect to the Boyer-Moore algorithm;
- constant extra space needed with respect to the Boyer-Moore algorithm;
- preprocessing phase in $O(m+\Pi)$ time and space complexity;
- searching phase in $O(n)$ time complexity;
- $2n$ text character comparisons in the worst case.

14. Apostolico-Giancarlo algorithm:

- variant of the Boyer-Moore algorithm;
- preprocessing phase in $O(m+\Pi)$ time and space complexity;
- searching phase in $O(n)$ time complexity;
- $\frac{3}{2}n$ comparisons in the worst case.

15. Reverse Colussi algorithm:

- refinement of the Boyer-Moore algorithm;
- partitions the set of pattern positions into two disjoint subsets;
- preprocessing phase in $O(m^2)$ time and $O(m\Pi)$ space;
- searching phase in $O(n)$ time complexity;
- $2n$ text character comparisons in the worst case.

16. Horspool algorithm:

- simplification of the Boyer-Moore algorithm;
- easy to implement;

- preprocessing phase in $O(m+\Pi)$ time and $O(\Pi)$ space complexity;
- searching phase in $O(mn)$ time complexity;
- the average number of comparisons for one text character is between $1/\Pi$ and $2/(\Pi+1)$.

17. Quick Search algorithm:

- simplification of the Boyer-Moore algorithm;
- uses only the bad-character shift;
- easy to implement;
- preprocessing phase in $O(m+\Pi)$ time and $O(\Pi)$ space complexity;
- searching phase in $O(mn)$ time complexity;
- very fast in practice for short patterns and large alphabets.

18. Tuned Boyer-Moore algorithm:

- simplification of the Boyer-Moore algorithm;
- easy to implement;
- very fast in practice.

19. Zhu-Takaoka algorithm:

- variant of the Boyer-Moore algorithm;
- uses two consecutive text characters to compute the bad-character shift;
- preprocessing phase in $O(m+\Pi^2)$ time and space complexity;
- searching phase in $O(mn)$ time complexity.

20. Berry-Ravindran algorithm:

- hybrid of the Quick Search and Zhu and Takaoka algorithms;
- preprocessing phase in $O(m+\Pi^2)$ space and time complexity;
- searching phase in $O(mn)$ time complexity.

21. Smith algorithm:

- takes the maximum of the Horspool bad-character shift function and the Quick Search bad-character shift function;
- preprocessing phase in $O(m+\Pi)$ time and $O(\Pi)$ space complexity;
- searching phase in $O(mn)$ time complexity.

22. Raita algorithm:

- first compares the last pattern character, then the first and finally the middle one before actually comparing the others;
- performs the shifts like the Horspool algorithm;
- preprocessing phase in $O(m+\Pi)$ time and $O(\Pi)$ space complexity;

- searching phase in $O(mn)$ time complexity.

23. Reverse Factor algorithm:

- uses the suffix automaton of x^R;
- fast on practice for long pattens and small alphabets;
- preprocessing phase in $O(m)$ time and space complexity;
- searching phase in $O(mn)$ time complexity;
- optimal in the average.

24. Turbo Reverse Factor algorithm:

- refinement of the Reverse Factor algorithm;
- preprocessing phase in $O(m)$ time and space complexity;
- searching phase in $O(n)$ time complexity;
- performs $2n$ text characters inspections in the worst case;
- optimal in the average.

25. Forward Dawg Matching algorithm:

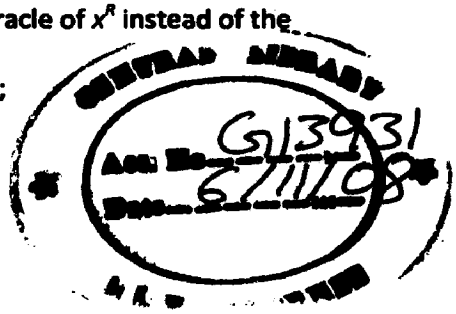
- uses the suffix automaton of x;
- $O(n)$ worst case time complexity;
- performs exactly n text character inspections.

26. Backward Nondeterministic Dawg Matching algorithm:

- variant of the Reverse Factor algorithm;
- uses bit-parallelism simulation of the suffix automaton of x^R;
- efficient if the pattern length is no longer than the memory-word size of the machine;

27. Backward Oracle Matching algorithm:

- version of the Reverse Factor algorithm using the suffix oracle of x^R instead of the suffix automaton of x^R;
- fast in practice for very long patterns and small alphabets;
- preprocessing phase in $O(m)$ time and space complexity;
- searching phase in $O(mn)$ time complexity;
- optimal in the average.



28. Galil-Seiferas algorithm:

- constant extra space complexity;
- preprocessing phase in $O(m)$ time and constant space complexity;
- searching phase in $O(n)$ time complexity;
- performs $5n$ text character comparisons in the worst case.

29. Two Way algorithm:

- requires an ordered alphabet;
- preprocessing phase in $O(m)$ time and constant space complexity;
- constant space complexity for the preprocessing phase;
- searching phase in $O(n)$ time;
- performs $2n-m$ text character comparisons in the worst case.

30. String Matching on Ordered Alphabets:

- no preprocessing phase;
- requires an ordered alphabet;
- constant extra space complexity;
- searching phase in $O(n)$ time;
- performs $6n+5$ text character comparisons in the worst case.

31. Optimal Mismatch algorithm:

- variant of the Quick Search algorithm;
- requires the frequencies of the characters;
- preprocessing phase in $O(m^2+\Pi)$ time and $O(m+\Pi)$ space complexity;
- searching phase in $O(mn)$ time complexity.

32. Maximal Shift algorithm:

- variant of the Quick Search algorithm;
- quadratic worst case time complexity;
- preprocessing phase in $O(m^2+\Pi)$ time and $O(m+\Pi)$ space complexity;
- searching phase in $O(mn)$ time complexity.

33. Skip Search algorithm:

- uses buckets of positions for each character of the alphabet;
- preprocessing phase in $O(m+\Pi)$ time and space complexity;
- searching phase in $O(mn)$ time complexity;
- $O(n)$ expected text character comparisons.

34. KMP Skip Search algorithm:

- improvement of the Skip Search algorithm;
- uses buckets of positions for each character of the alphabet;
- preprocessing phase in $O(m+\Pi)$ time and space complexity;
- searching phase in $O(n)$ time complexity.