Parallel Algorithms for String Matching Problem Baesd on Butterfly Model

[1]K K V V V S Reddy, [2]Dr. S. Viswanadha Raju, [3]Chinta Someswara Rao, [4]K Butchi Raju
[1]Rayalasheema University, AP, India
[2]School of IT, JNTUH, Hyderabad, AP, India
[3]Dept. of CSE, SRKR Engineering College, Bhimavaram, AP, India
[4]Dept. of CSE, GRIET, Hyderabad, AP, India

Abstract
The string matching problem is one of the most studied problems in computer science. While it is very easily stated and many of the simple algorithms perform very well in practice, numerous works have been published on the subject and research is still very active. In this paper we propose a butterfly parallel computing model for parallel string matching. Experimental results show that, on a multi-processor system, the butterfly model implementation of the proposed parallel string matching algorithm can reduce string matching time by more than 40%.

Keywords
String matching; Parallel string mathing; computing model;Butterfly.

I. Introduction

A. String Matching

String matching has been one of the most extensively studied problems in computer engineering since it performs important tasks in many applications like information retrieval (IRS), web search engines, error correction and several other fields [1]. Especially with the introduction of search engines dealing with tremendous amount of textual information presented on the World Wide Web, so this problem deserves special attention and any improvements to speed up the process will benefit these important applications [2].

As current free textual databases are growing almost exponentially with the time, the string matching problem becomes impractical to use the fastest sequential algorithms on a conventional sequential computer system [3-4]. To improve the performance of searching on large text collections, some researchers developed special purpose algorithms called parallel algorithms that parallelized the entire database comparison on general purpose parallel computers where each processor performs a number of comparisons independently. In Parallel processing the text string T and pattern P are assumed and that two input words have already been allocated in the processors in such a way that each processor stores a single text symbol, and some processors additionally a single pattern symbol. The input words are stored symbol-by-symbol in consecutive processors numbered according to the snake-like row-major indexing, that is, the processors in the odd-numbered rows 1, 3, 5, ... are numbered from left to right, and in the even-numbered rows from right to left. (The first symbols of T and P are in processor 1, the next in processor 2, and so on.) This allocation scheme places symbols adjacent in the text or pattern in adjacent processors. The output of the string matching algorithm is that each processor is to be marked as either being a starting position of an occurrence of P in T or not. In this paper we proposed a parallel string matching technique based on butterfly model.

The main contributions of this work are summarized as follows. This work offers a comprehensive study as well as the results of typical parallel string matching algorithms at various aspects and their application on butterfly computing models. This work suggests the most efficient algorithmic butterfly models and demonstrates the performance gain for both synthetic and real data. The rest of this work is organized as, review typical algorithms, algorithmic models and finally conclude the study.

II. Related work

The first optimal parallel string matching algorithm was proposed by Galil [5]. On SIMD-CRCW model, this algorithm required n / log n processors, and the time complexity is O(log n) ; on SIMD-CREW model, it required n / log2 n processors and the time complexity is O(log2 n). Vishkin [6] improved this algorithm to ensure it is still optimal when the alphabet size is not fixed. In [7], an algorithm used O (n × m) processors was presented, and the computation time is O(log log n) . A parallel KMP string matching algorithm on distributed memory machine was proposed by CHEN [8]. The algorithm is efficient and scalable in the distributed memory environment. Its computation complexity is O(n / p + m) , and p is the number of the processors.

SV Raju et.al [9] presents new method for exact string matching algorithm based on layered architecture and two-dimensional array. This has applications such as string databases and computational biology. The main use of this method is to reduce the time spent on comparisons of string matching by distributing the data among processors which achieves a linear speedup and requires layered architecture and additionally p*# processors.

Bi Kun et.al [10] proposed the improved distributed string matching algorithm. And also an improved single string matching algorithm based on a variant Boyer-Moore algorithm is presented. In this they implement algorithm on the above architecture and the experiments prove that it is really practical and efficient on distributed memory machine. Its computation complexity is O(n/p + m), where n is the length of the text, and m is the length of the pattern, and p is the number of the processors. They show that this distributed architecture is suitable for paralleling the multipattern string matching algorithms and approximate string matching algorithms.

Hsi-Chieh Le [11] et.al presents three algorithms for string matching on reconfigurable mesh architectures. Given a text T of length n and a pattern P of length m, the first algorithm finds the exact matching between T and P in O(1) time on a 2-dimensional RMESH of size (n - m+ 1) x m. The second algorithm finds the approximate matching between T and P in O(k) time on a 2D RMESH, where k is the maximum edit distance between T and P. The third algorithm allows only the replacement operation in the calculation of the edit distance and finds an approximate matching between T and P in constant-time on a 3D RMESH. By this paper we state that this is simpler model would be sufficient to run the proposed algorithms without increasing the reported time complexities.

S V Raju [12] et.al considers the problem of string matching

algorithm based on a two-dimensional mesh. This has applications such as string databases, cellular automata and computational biology. The main use of this method is to reduce the time spent on comparisons in string matching by using mesh connected network which achieves a constant time for mismatch a text string and. This is the first known optimal-time algorithm for pattern matching on meshes. The proposed strategy uses the knowledge from the given algorithm and mesh structure.

Its'hak Dinstein [13] et.al propose a parallel computation approach to two dimensional shape recognition. This approach uses parallel techniques for contour extraction, parallel computation of normalized contour-based feature strings independent of scale and orientation, and parallel string matching algorithms. The implementation on the EREW PRAM architecture is discussed, but it can be adapted to other parallel architectures.

Jin Hwan Park [14] et.al presents efficient dataflow schemes for parallel string matching. In this they consider two sub problems known as the exact matching and the k-mismatches problems are covered. Three parallel algorithms based on multiple input (and output) streams are presented. Time complexities of these parallel algorithms are O((n/d)+a), 0 £ a £ m, where n and m represent lengths of reference and pattern strings (n >> m) and d represents the number of streams used (the degree of parallelism). They show, they can control the degree of parallelism by using variable number (d) of input (and output) streams. They show their approaches solve the exact matching and the k-mismatches problems with time complexities of O((n / d) + a), where a = 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).

S V Raju [15] et.al considers the problem of exact string matching algorithm based on a two-dimensional array. This has applications such as string databases, cellular automata and computational biology. The main use of this method is to reduce the time spent on comparisons in string matching by finding common characters in pattern string which achieves a constant time O(1) for pattern string in a text string. This reduces many calls across backend interface.

Chuanpeng Chen [16] et.al propose a high throughput configurable string matching architecture based on Aho-Corasick algorithm. The architecture can be realized by random-access memory (RAM) and basic logic elements instead of designing new dedicated chips. The bit-split technique is used to reduce the RAM size, and the byte-parallel technique is used to boost the throughput of the architecture. By the particular design and comprehensive experiments with 100MHz RAM chips, one piece of the architecture can achieve a throughput of up to 1.6Gbps by 2-byte-parallel input, and we can further boost the throughput by using multiple parallel architectures.

Prasanna [17] et.al propose a multi-core architecture on FPGA to address these challenges. They adopt the popular Aho-Corasick (AC-opt) algorithm for our string matching engine. Utilizing the data access feature in this algorithm, they design a specialized BRAM buffer for the cores to exploit a data reuse existing in such applications. Several design optimizations techniques are utilized to realize a simple design with high clock rate for the string matching engine. An implementation of a 2-core system with one shared BRAM buffer on a Virtex-5 LX155 achieves up to 3.2 GBPS throughput on a 64 MB state transition table stored in DRAM. Performance of systems with more cores is also evaluated for this architecture, and a throughput of over 5.5 Gbps can be obtained for some application scenarios.

S. Muthukrishnan et.al [18] present an algorithm on the CRCW PRAM that checks if there exists a false match in O(1) time using O(n) processors. This algorithm does not require preprocessing the pattern. Therefore, checking for false matches is provably simpler than string matching since string matching takes O(log(log m)) time on the CRCW PRAM. In this hey use this simple algorithm to convert the Karp–Rabin Monte Carlo type string-matching algorithm into a Las Vegas type algorithm without asymptotic loss in complexity. Finally they present an efficient algorithm for identifying all the false matches and, as a consequence, show that string-matching algorithms take A.log log m/ time even given the flexibility to output a few false matches.

S V Raju [19] et.al present new approach for parallel string matching. Some known parallel string matching algorithms are considered based on duels by witness who focuses on the strengths and weaknesses of the currently known methods. The new 'divide and conquer' approach has been introduced for parallel string matching, called the W-period, which is used for parallel preprocessing of the pattern and has optimal implementations in a various models of computation. The idea, common for every parallel string matching algorithm is slightly different from sequential ones as Knuth-Morris-Pratt or Boyer-Moore algorithm.

III. Computing Model

A computational model defines the behavior of a theoretic machine. The goal of a model is to ease the design and analysis of algorithms to be executed in a wide range of architectures with the performance predicted by the model [20]. Generally it divides into two categories sequential and parallel models. Traditionally, software has been written for serial computation as shown in fig 1.

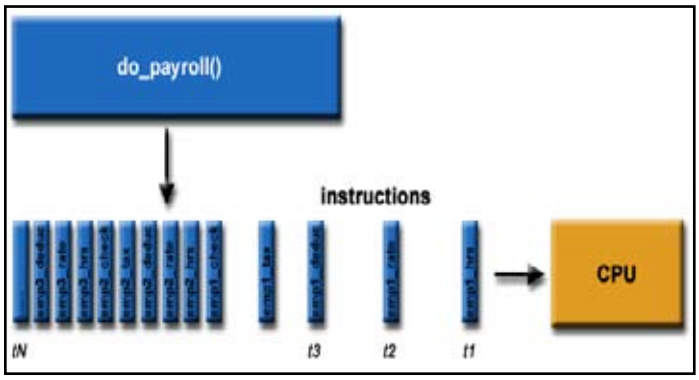

Fig. 1: Serial Computational Model

The main objectives are

- To be run on a single computer having a single Central Processing Unit (CPU);
- A problem is broken into a discrete series of instructions.
- Instructions are executed one after another.
- Only one instruction may execute at any moment in time.

A. Parallel Computational Model

In the simplest sense, parallel computing is the simultaneous use of multiple compute resources to solve a computational problem [21-22]. Parallel processing has emerged as a key enabling technology in modern computers, driven by the ever-increasing demand for higher performance, lower costs, and sustained productivity

in real-life applications. Concurrent events are taking place in today's high performance computers due to the common practice of multiprogramming, multiprocessing, or multi computing as shown in fig. 2.

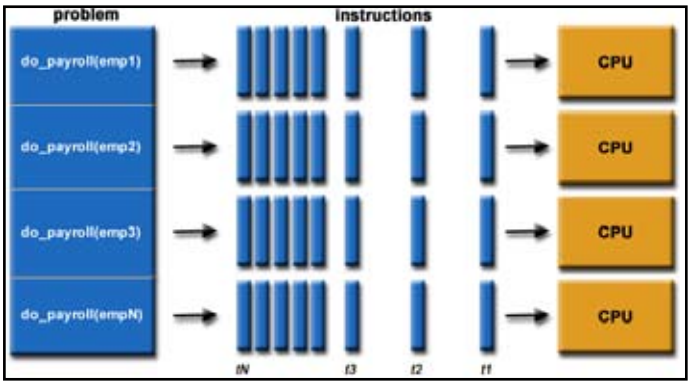

Fig. 2: Parallel Computational Model

The main objectives are

- To be run using multiple CPUs
- A problem is broken into discrete parts that can be solved concurrently
- Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different CPUs

B. Parallel Computer Models

Parallelism appears in various forms, such as look ahead, pipelining Vectorization concurrency, simultaneity, data parallelism, partitioning, interleaving, overlapping, multiplicity, replication, time sharing, space sharing, multitasking, multiprogramming, multithreading, and distributed computing at different processing levels [21-22].

1. Flynn's Classification

Michael Flynn introduced a classification of various computer architectures based on notions of instruction and data streams [8]. Conventional sequential machines are called SISD Computers. Vectors are equipped with scalar and vector hardware or appear as SIMD machines. Parallel computers are reserved for MIMD machines. An MISD machines are modeled. The same data stream flows through a linear array of processors executing different instruction streams. This architecture is also known as systolic arrays for pipelined execution of specific algorithms.

2. Parallel/Vector Computers

Intrinsic parallel computers are those that execute programs in MIMD mode. There are two major classes of parallel computers, namely shared-memory multiprocessors and message-passing multicomputer [22]. The major distinction between multiprocessors and multicomputer lies in memory sharing and the mechanisms used for interprocesssor communication.

The processors in a multiprocessor system communicate with each other through shared variables in a common memory. Each computer node in a multicomputer system has a local memory, unshared with other nodes. Inter processor communication is done through message passing among the nodes.

Explicit vector instructions were introduced with the appearance of vector processors. A vector processor is equipped with multiple vector pipelines that can be concurrently used under hardware or firmware control. There are two families of pipelined vector processors.

C. Emulation Among Models

Although it may appear that a different algorithm must be designed for each of the many parallel models, there are often automatic and efficient techniques for translating algorithms designed for one model into algorithms designed for another. These translations are work-preserving in the sense that the work performed by both algorithms is the same, to within a constant factor. For example, the following theorem, known as Brent's Theorem [23], shows that an algorithm designed for the circuit model can be translated in a work-preserving fashion to a PRAM model algorithm.

Theorem 3.1 (Brent's Theorem)

Any algorithm that can be expressed as a circuit of size (i.e., work) W, depth D and with constant fan-in nodes in the circuit model can be executed in O(W/P + D) steps in the CREW PRAM model.

Proof: The basic idea is to have the PRAM emulate the computation specified by the circuit in a level-by-level fashion. The level of a node is defined as follows. A node is on level 1 if all of its inputs are also inputs to the circuit. Inductively, the level of any other node is one greater than the maximum of the level of the nodes that provide its inputs. Let li denote the number of nodes on level i. Then, by assigning [li/P] operations to each of the P processors in the PRAM, the operations for level i can be performed in O([li/P]) steps. Concurrent reads might be required since many operations on one level might read the same result from a previous level. Summing the time over all D levels, we have

$$T_{PRAM}(W,D,P) = O(\sum_{i=1}^{D}[\frac{l_i}{p}])$$
$$= O(\sum_{i=1}^{D}(\frac{l_i}{p}+1)$$
$$= O(\frac{1}{P}(\sum_{i=1}^{D} l_i) + D)$$
$$= O(W/P + D)$$

The last step is derived by observing that $w = \sum_{i=1}^{D} l_i$, i.e., that the work is equal to the total number of nodes on all of the levels of the circuit.

Theorem 3.2

Any algorithm that takes time T on a P-processor PRAM model can be translated into an algorithm that takes time O(T(P/P′ + log P′)), with high probability, on a P′-processor butterfly machine model.

Sketch of proof: Each of the P′ processors in the butterfly emulates a set of P/P′ PRAM processors.

The butterfly emulates the PRAM in a step-by-step fashion. First, each butterfly processor emulates one step of each of its P/P′ PRAM processors. Some of the PRAM processors may wish to perform memory accesses. For each memory access, the butterfly processor hashes the memory address to a physical memory bank and an address within the bank. Ranade proved that if each processor in the P-processor butterfly sends at most P/P′ messages, and if the destinations of the messages are determined by a sufficiently powerful random hash function, then the it can deliver all of the messages, along with responses, in O(P/P′ + log P′) time.

D. Communication Network Relation to Computer System Components

A typical distributed system is shown in Figure 3. Each computer has a memory-processing unit and the computers are connected by a communication network. Figure 4 shows the relationships of the software components that run on each of the computers and use the local operating system and network protocol stack for functioning. The distributed software is also termed as middleware. A distributed execution is the execution of processes across the distributed system to collaboratively achieve a common goal. An execution is also sometimes termed a computation or a run. The distributed system uses a layered architecture to break down the complexity of system design. The middleware is the distributed software that drives the distributed system, while providing transparency of heterogeneity at the platform level [24]. Fig. 4 schematically shows the interaction of this software with these system components at each processor.

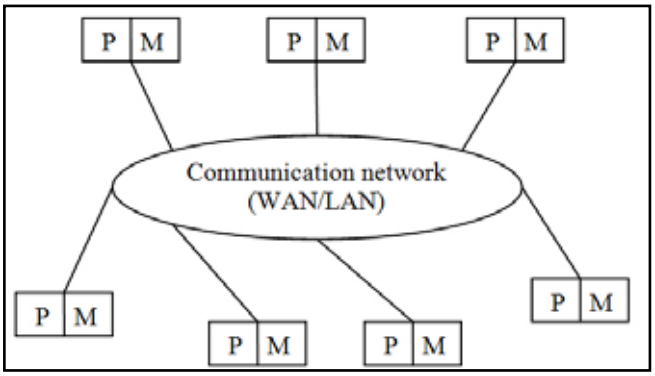

Fig. 3: A Distributed Systems Connects Processors by a Communication Network

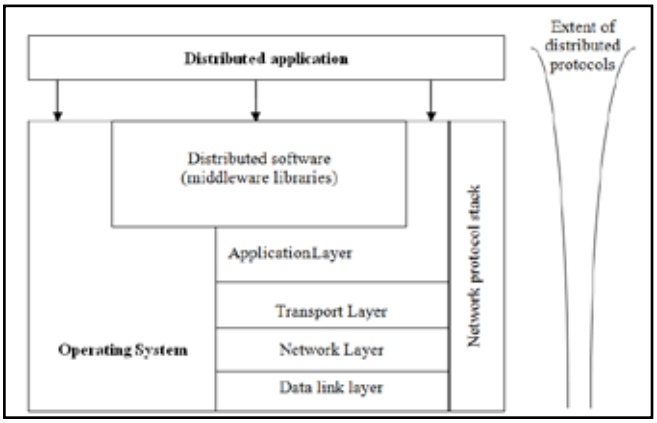

Fig. 4: Interaction of the Software Components at Each Processor

IV. Text Partitioning

The exact string-matching problem can achieve data parallelism with data partitioning technique. We decompose the text into r subtexts, where each subtext contains (T/p)+m-1 successive characters of the complete text. There is an overlap of m-1 string characters between successive subtexts, i.e, a redundancy of r(m-1) characters. Alternatively it could be assumed that the database of an information retrieval system contains r independent documents. Therefore, in both the cases all the above partitions yield a number of independent tasks each comprising some data (i.e. a string and a large subtext) and a sequential string matching procedure that operates on that data. Further, each task completes its string matching operation on its local data and returns the number of occurrences [24-26]. Finally, we can observe that there are no communication requirements among the tasks but only global (or collective) communication is required.

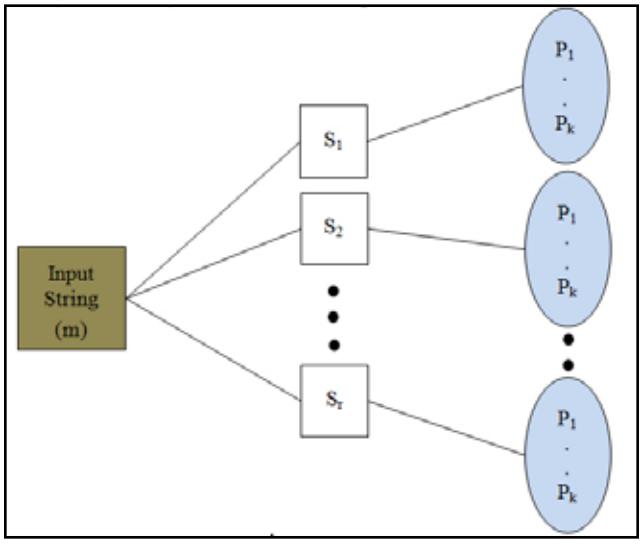

Fig. 5: Framework for Pool of Processors

The main issue to be addressed is how the several tasks (or r subtexts) can be mapped or distributed to multiple processors for concurrent execution. In [24-26] different ways of distributing the database across a multi computer network were discussed. Let p be the number of processors in network and r be the number of subtext in the whole collection then the text partition is defined as, if r=p then each subtext contains T/p+m-1 characters. This is called static allocation of subtext as shown in fig 5. In the next section we present the parallel algorithm that is based on static allocation of subtext using MPI library. A significant contribution of this paper is a demonstration of the maximum size buffer with 2k processors for implementation of string matching and capable of accepting a character from r subtexts where k=8bits. This architecture enables a buffered string matching system implementing a KMP like pre computation algorithm. In the above mapping {a1, a2…ar} is the input string where r represents subtext and {p1, p2…pk} are the number of processors for the given input string (k=8). In the above mapping the given input string will be allocated to each processor as shown in Fig. 5.

V. Butterfly Model Prelimanaries

A. Introduction

The butterfly network is an interconnection system most frequently associated with Fast Fourier Transform. In general, it consists of $p = (q + 1)2^q$ processors, organized as $q + 1$ ranks of 2^q processors each (Fig. 1). Optionally we shall identify the rightmost and the leftmost ranks, so there is no rank q, and the processors on ranks 0 and $q - 1$ are connected directly.

Let us denote the processor i on the rank r by $P_{ir,r}$, $0 \le i < 2^q$, $0 \le r \le q$. Then processor $P_{i,r+1}$ is connected to the two processors $P_{i,r,}$ and $P_{ir,r}$ and processor $P_{ir,r+1}$ is connected to the two processors Pi,r. and $P_{ir,r}$. Recall that $i^r = I_{q-1} .. .. i_r..i_0$ These four connections form a "butterfly" pattern, from which the name of the network is derived. The hypercube is actually the butterfly with the rows collapsed. The communication link in the hypercube between processors P_i, and P_{ir} is identified with the communication links in the butterfly between $P_{i,r+l}$ and $P_{ir,r+1}$, and between $P_{ir,r+l}$ and $P_{i,r}$.

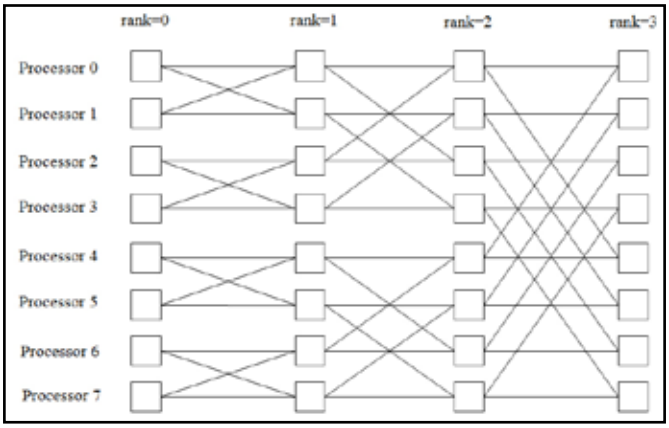

Fig. 6: Butterfly Network for q= 3 and p= 4:23

B. Prelimanaries

The network is a directed acyclic graph N with unit edge capacities, a single source node s and several sinks tϵT

The transfer matrix is $F = (I - K)^{-1} \epsilon GF(q)^{n \times n}$,where K and has zero coefficients whenever the adjacency matrix of the line graph of N does [Koetter, Medard 03].

If x is transmitted from the source node s and edges of the network are corrupted by an error vector e then the network transmission is y = (x + e)F:

The transfer matrix for the Buttery Network(shown in fig. 7) is given by $F = I + K + K_2 + K_3 = (I - K)_{-1}$

$$F = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Fig. 7: Transfer Matrix for the Buttery Network

The Transfer Matrix for each Receiver

F_t is the $n \times n_t$ sub matrix of F whose columns correspond to the nt edges connected to t.

Node t receives $y = (x + e)F_t$:

The code C_t is a subset of $\{zF_t : z \epsilon GF(q)^n\}$.

Messages z; $z_1 \epsilon GF(q)^n$ are identified if $z - z_1 \epsilon \ker F_t =: K_t$

The transfer matrix for the Buttery Network(shown in fig. 8) is given by $F = I + K + K_2 + K_3 = (I - K)^{-1}$

$$F = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Fig. 8: Transfer Matrix at Receiver

D. Distance Function

K_t induces a distance function on $GF(q)n_t$ by $d_t(u; v) := \min\{d(x; y) : xF_t = u; yF_t = v\}$; where d is a distance function on $GF(q)^n$. Then $d_t(u; v) = 0$ if and only if $x - y \epsilon K_t$ for some $x; y \epsilon GF(q)^n$ satisfying $xF_t = u$; $yF_t = v$. For the Hamming distance, $w_t(u) = d_t(u; 0)$ counts the minimum number of linearly independent rows of F_t required to obtain a representation of $u = xF_t$.

E. Interconnection Function

Unlike the Omega network, the generation of the interconnection pattern between a pair of adjacent stages depends not only on n but also on the stage number s. The recursive expression is as follows. Let there be M =n/2 switches per stage, and let a switch be denoted by the tuple <x,s>, where $x \in [0,M-1]$ and stage $s \in [0, \log_2 n-1]$. The two outgoing edges from any switch <x,s> are as follows. There is an edge from switch <x,s> to switch <y,s+1> if (i) x = y or (ii) x XOR y has exactly one 1 bit, which is in the (s+1)th MSB. For stage s, apply the rule above for M/2s switches. Whether the two incoming connections go to the upper or the lower input port is not important because of the routing function, given below.

Example Consider the Butterfly network in Fig. 1.4(b), n = 8 and M = 4. There are three stages, s = 0,1, 2, and the interconnection pattern is defined between s = 0 and s = 1 and between s = 1 and s = 2. The switch number x varies from 0 to 3 in each stage, i.e., x is a 2-bit string.

Consider the first stage interconnection (s = 0) of a butterfly of size M, and hence having $\log_2 2M$ stages. For stage s = 0, as per rule (i), the first output line from switch 00 goes to the input line of switch 00 of stage s =1. As per rule (ii), the second output line of switch 00 goes to input line of switch 10 of stage s = 1. Similarly, x = 01 has one output line go to an input line of switch 11 in stage s = 1. The other connections in this stage can be determined similarly. For stage s = 1 connecting to stage s = 2, we apply the rules considering only $M/2^1 = M/2$ switches, i.e., we build two butterflies of size M/2 – the "upper half" and the "lower half" switches. The recursion terminates for $M/2^s = 1$, when there is a single switch.

VI. Proposed System Architecture

A. The Butterfly

The Butterfly is a coarse-grained, shared-memory, expandable MIMD parallel computer built by Bolt Beranek and Newman (BBN) [BBN 861]. The computer got its name from the Butterfly switch which it uses for interprocessor communication. The switch supports a processor-to-processor bandwidth of 32 Mbits j second. One processor (Motorola MC68000) and 1 to 4 MBytes of memory are located on a single board called a Processor Node. The processor is capable of executing 500,000 instructions per second.

Remote memory references (via the Butterfly switch) take about five times longer than local references. The speed of the processors, memories, and switch are balanced to ensure that none become a performance bottleneck.

The Butterfly's architecture scales in a flexible and cost-efficient fashion from 1 to 256 processors. When using 256 processors the computer has a raw processing power of 128 MIPS and a main memory of 256 to 1024 MBytes. For applications such as matrix multiplication, Gaussian elimination, convolution, and histograms of images nearly linear speed is achieved through the entire range of processors.

B. Butterfly Architecture

Each Butterfly node contains an 8Mhz Motorola MC68000 microprocessor with 24 bit virtual addresses, at least 1 M byte of main memory, a 2901-based bit-slice microcoded co-processor called the Processor Node Controller (PNC), memory management hardware, an I/O bus, and an interface to the Butterfly switch. The PNC interprets every memory reference issued by the 68000 and is used to communicate with other nodes across the switching network. It also provides, in microcode, efficient test-and-set and queuing operations, a process scheduler, and communication synchronization mechanisms. The memory management unit is used to translate virtual addresses (used by the 68000) into physical memory addresses. As a result, the memory of all Processor Nodes, taken together, appear as a large single global memory to application software.

The Butterfly switch is a collection of 4 x 4 switch elements (4-input 4-output crossbars) configured as a "serial decision" network, a topology similar to that of the Fast Fourier Transform Butterfly. An N-processor system uses (N log4 N)/4 switches arranged in log, N columns. Switch operation is similar to that of a packet switching network. Figure 9 illustrates a 16-input 16-output Butterfly switch. To reduce switch contention a large configuration (e.g., a 128-node Butterfly) contains extra switch nodes, used to provide alternate communication paths between processors. This also makes the switch more resilient to switching node failures. Machines are configured so that the probability of message collision within the switch is relatively low. The switch supports efficient transfer of blocks of data between any pair of Processor Nodes at full switch bandwidth. The Butterfly I/O system is distributed among processor Nodes. The I/O bus on each processor supports connections to a Multibus (for fast I/O devices such as disks, and external memory and processors) and serial RS-232 lines (for terminals).

C. Programming the Butterfly Parallel Processor

The Butterfly Parallel Processor is programmed exclusively in high-level programming languages. Searching, IRS, Editing, compiling and linking, downloading, running and debugging of programs are done from a UNIX front-end. A window manager enables rapid switching between the front-end and the Butterfly system environments. Two distinct approaches to programming the Butterfly have seen widespread use: message passing and shared memory. When using the message passing paradigm the programmer decomposes the application into a moderately sized collection of loosely coupled processes which from time to time exchange control signals or data. This approach is similar to programming a multiprocessor application for a uniprocessor. In the shared memory approach, a task is usually some small procedure to be applied to a subset of the shared memory. A task, therefore, can be represented simply as an index, or a range of indices, into the shared memory and an operation to be performed on that memory. This style is particularly effective for applications containing a few frequently repeated tasks. Memory and processor management are used to keep all memories and processors equally busy.

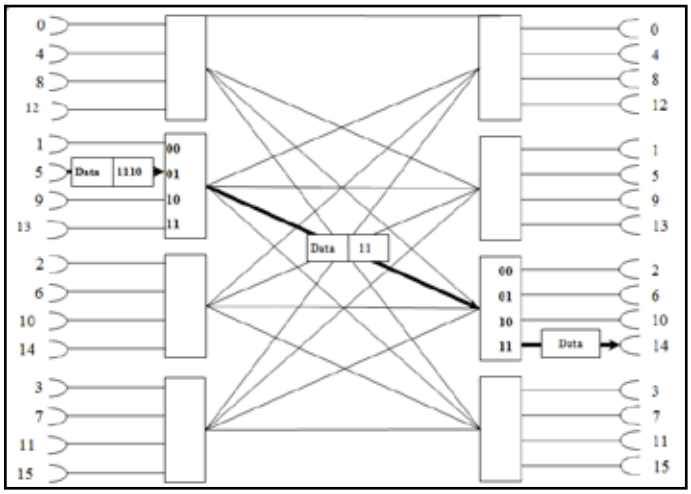

Fig. 9: 16-I/P 16-O/P Butterfly Computing Model

VII. Proposed Algorithm

In this paper, we designed an algorithm, which is designed based on the Hamming distance. Hamming distance error correction. So it is best of known for finding of differences between two strings. The Hamming distance is a measure of distance between two strings equal is the number of positions for which the corresponding symbols may be different, that need to be changed to obtain one from the other, as per the definition for the following two strings: "rama" and "rimu" have a Hamming distance of 2. Let $A=a_1a_2..a_m$ and $B=b_1b_2..b_m$ be two strings of length m , The Hamming distance of string A and string B, noted Ham(A,B) , is the number of locations where A[i]!=B[i], $1 \le i \le m$.How to use this definition in order to solve string matching problem as shown below: Let us consider text(T)length of n and pattern(P) length of m. suppose there is an occurrence of P in T, it means the text string ti,ti+1..ti+m-1 equal to P, so that H(ti,ti+1..ti+m-1,P)=0. It is used for exact string matching problem. Suppose that there is an occurrence of P in T. It means that a factor ti,ti+1..ti+m-1 of R is equal to P with at most k mismatches , hence Ham(ti,ti+1..ti+m-1,P) $\le$ k.

A. Constant Time String Matching Algorithm on Butterfly Model

In this model data on processors have been organized such that they represent the m sets of length of n-m+1 of the text string with m* n-m+1 matrix plus, the first processor of each row segment holding the first element of each set also carries an element of pattern. The process is similar as per above for the remaining m-1 rows. First show how to find the occurrences of pattern P in text string T on Butterfly model with m*(n-m+1) in constant time O (1).

Algorithm for string matching (pattern P, text T)

Begin

Initially: m elements of pattern initially distributed to the m processors on the first column, one processor and $L_{i,j}$ distributed to the m*(n-m+1) processors on the row and column. Where $1 \le i \le m$ and $m \le j \le m*(n-m+1)$

First processors broadcast elements on row buses: each first processor $p_{i,1}$, where $1 \le i \le m$, broadcasts the element ci,1 to every processor in the ith row using only row buses .After this multiple row broadcast communication operations each processor $P_{i,j}$ saves received element as $r_{i,j}$.

Compare and Set results: Each processor $P_{i,j}$ compares $r_{i,j}$ with $X_{i,j}$, if $r_{i,j} = X_{i,j}$ if sets result=1 otherwise, result=0

Sum up 1's: perform a one-dimensional binary prefix sum operation on each column simultaneously for the value of result. Each processor $P_{i,j}$ where $1 \le (i,j) \le m$ stores the binary prefix sums to $b_{i,j}$.

Based on Hamming Distance: If $P_{m,j} = 0$ then the string matching(i.e. exact string matching) otherwise approximate string matching with k mismatches.

End.

Conclusion

Each step in the above algorithm runs in constant time. Thus we have the following theorem.

Theorem 7.1

There is a constant time string matching algorithm on a Butterfly model that finds the occurrences of pattern in text using m*(n-m+1) processors

Example

Text string T(n)= mnonop and Pattern string P(m)= non

L1 ={mnon}, L2={nono} L3={onop}

n	m	n	o	n			<n,m>	<n,n>	<n,o>	<n,n>
o	n	o	n	o			<o,n>	<o,o>	<o,n>	<o,o>
n	o	n	o	p			<n,o>	<n,n>	<n,o>	<n,p>
Step 1							Step 2			

	0	1	0	1			0	1	0	1
	0	1	0	1			0	1	0	1
	0	1	0	0			0	1	0	0
	Step 3						3	0	3	1
							Step 4			

As per the given example, after step 4 in the matrix Mm+1,j values useful for deciding the matching is exact string matching or approximate string matching with the k mismatches.

Scalability:

Suppose the number of available processors is p, if $(n-m-1) \le p < m*(n-m+1)$, then modify algorithm in the following way to achieve good scalability. As per the above information, it is still want to split the processors into m sub-buses in step 1. Each group is responsible for one set of m sets and pattern string, matching array. Thus there x=p/(n-m+1) processors in each sub-bus .every processor has three arrays :pattern, reference(Li) and matching array each of which has size m*(n-m+1)/p. the algorithm takes (m*(n-m+1))/p bus cycle and there is no local computation for multicast and contains the (m*(nm+ 1))/p computation time. The time complexity for the modified string matching algorithm (m*(n-m+1))/p. in other words , the complexity T1(N)=O(1) , T2(P)=(m*(n-m+1))/p where N=m*(n-m+1) and P=p the scalability for our algorithm is

$g(N,P)=T2(P)/T1(N)* P/N = 1$.

Conclusion

So that the string matching is completely scalability and obtain the following theorem.

Theorem 7.2

The given two strings size of text n and size of pattern m. find the occurrences of pattern in text.

There is completely scalable on Butterfly model. The algorithm runs in O(m*(n-m+1))/P time, where P is the number of processors and $1 \le p \le m*(n-m+1)$.

VIII. Speed and Precision Comparisons

When the DFT is calculated by correlation, the program uses two nested loops, each running through N points. This means that the total number of operations is proportional to N times N. The time to complete the program is thus given by, the time required to calculate a DFT by correlation is proportional to the length of the DFT squared (k_{DFT} N2) where N is the number of points in the DFT and k_{DFT} is a constant of proportionality. For example, a 1024 point DFT will require about 25 seconds, or nearly 25 milliseconds per point. That's slow!

Using this same strategy we can derive the execution time for the FFT. The time required for the bit reversal is negligible. In each of the Log_2N stages there are N/2 butterfly computations. This means the execution time for the program is approximated by: $k_{FFT}N \log_2 N$. The value of k_{FFT} is about 10 microseconds on a 100 MHz Pentium system. A 1024 point FFT requires about 70 milliseconds to execute, or 70 microseconds per point. This is more than 300 times faster than the DFT calculated by correlation!. Not only is $NLog_2N$ less than N2, it increases much more slowly as N becomes larger. For example, a 32 point FFT is about tentimes faster than the correlation method. However, a 4096 point FFT is one-thousand times faster. For small values of N (say, 32 to 128), the FFT is important. For large values of N (1024 and above), the FFT is absolutely critical. Fig. 10, compares the execution times of the two algorithms in a graphical form.

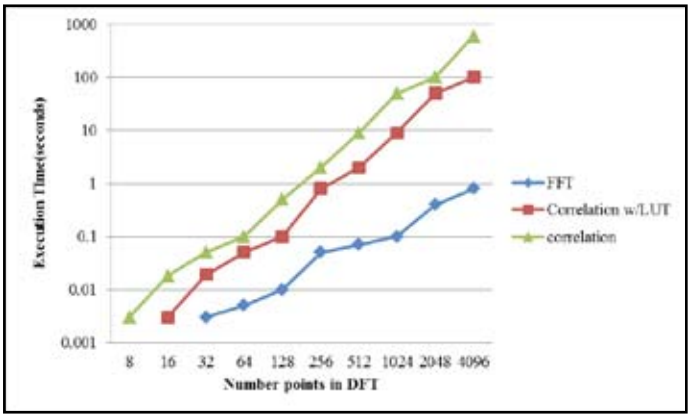

Fig. 10: Comparasion Among Different Procedures

IX. Conclusions

Parallel string matching is a ubiquitous problem that arises in a wide range of applications in computing, to full fill this we need efficient techniques. In this study we concentrate on parallel algorithms for string matching on computing models, especially in butterfly model. In this paper simulate the parallel algorithms for the implementation of high speed string matching; this uses fine-grained parallelism and performs matching of a search string by splitting the string into a set of substrings and then matching all of the substrings simultaneously. We also see that this implementation can be optimized in terms of resource utilization. Our future work is to apply omega network model to parallel string matching.

References

[1] K, Grabowski S,"Average-Optimal String Matching", Journal of Discrete Algorithms, pp. 579-594, 2009.

[2] Luis Russo L, Navarro G, Oliveira A, Morales P, "Approximate String Matching with Compressed Indexes Algorithm", pp. 1105-1136, 2009.

[3] Ilie L, Navarro G, Tinta L,"The Longest Common Extension Problem, Revisited and Applications to Approximate String Searching", Journal of Discrete Algorithms, pp-418-428, 2010.

[4] Fredriksson K, Grabowski S,"Average-Optimal String Matching, Journal of Discrete Algorithms", pp. 579-594, 2009.

[5] Z. Galil, "Optimal parallel algorithms for string matching", in Proc. 16th Annu. ACM symposium on Theory of computing, pp. 240-248, 1984.

[6] U. Vishkin,"Optimal parallel matching in strings", Information and control, Vol. 67, pp. 91-113, 1985.

[7] Y. Takefuji, T. Tanaka, K. C. Lee,"A parallel string search algorithm", IEEE Trans. Systems, Man and Cybernetics, Vol. 22, pp. 332-336, March-April 1992.

[8] CHEN Guo-liang, LIN-Jie, GU Nai-jie,"Design and analysis of string matching algorithm on distributed memory machine", Journal of Software, Vol. 11, pp. 771-778, 2000.

[9] Viswanadha Raju, S., Vinaya Babu, A., Mrudula, M.; "Backend Engine for Parallel String Matching Using Boolean Matrix", IEEE on PAR ELEC, pp-281-283,2006.

[10] Bi Kun, Gu Nai-jie, Tu Kun, Liu Xiao-hu, Liu Gang A Practical Distributed String Matching Algorithm Architecture and Implementation World Academy of Science", Engineering and Technology, 2005.

[11] Hsi-Chieh Leet, Fikret Ercalt,"RMESH Algorithms For Parallel String Matching", IEEE, 1997.

[12] S. Viswanadha Raju, A. Vinayababu,"Optimal Parallel algorithm for String Matching on Mesh Network Structure", 2006.

[13] Its'hak Dinstein, ad M. Landau,"Using Parallel String Matching Algorithms for Contour Based 2-D Shape Recognition", IEEE, 1990.

[14] Jin Hwan Park, K. M. George,"Parallel String Matching Algorithms Based on Dataflow", IEEE on System Sciences, 1999.

[15] S. Viswanadha Raju S R Mantena A.Vinaya Babu G V S Raju,"Efficient Parallel Pattern Matching using Partition Method", 2006.

[16] Chuanpeng Chen, Zhongping Qin,"A Bit-split Byte-parallel String Matching Architecture", IEEE, 2009.

[17] Qingbo Wang, Viktor K. Prasanna,"Multi-Core Architecture on FPGA for Large Dictionary String Matching", IEEE on Field Programmable Custom Computing Machines,2009.

[18] S. Muthukrishnan,"Detecting False Matches in String-Matching Algorithms", Algorithmica, Springer-Verlag New York Inc.1997.

[19] S.Viswanadha Raju, A.Vinaya Babu, G.V.S.Raju, K.R. Madhavi,"W-Period Technique for Parallel String Matching", 2007.

[20] D. P. Bertsekas, J. N. Tsitsiklis,"Parallel and Distributed Computation: Numerical Methods", Prentice-Hall.

[21] Bernd Mohr,"Introduction to Parallel Computing", John von Neumann Institute for Computing, 2006.

[22] Kai Hwang,"Advanced computer architecture: Parallelism, scalability, programability", TMH.

[23] R. P. Brent,"The parallel evaluation of general arithmetic expressions", Journal of the Association for Computing Machinery, 21(2), pp. 201–206, Apr. 1974.

[24] Ajay D. Kshemkalyani, Mukesh Singhal,"Distributed Computing: Principles, Algorithms, and Systems",Cambridge.

[25] S.Viswanadha Raju, A.Vinayababu,"Performance in the design of Parallel Programming", Proc ObComAPC-2004, Allied Publications, pp. 380-392, 2004.

[26] S.Viswanadha Raju, A.Vinayababu, S.P.Yanaiah, GVSRaju, 2006,"Parallel Approach for K String Matching", Proc NCIMDiL-2006, Indian Institute Of Technology, Kharagpur , 5-10.

[27] S. Viswanadha Raju, A.Vinaya Babu, 2006,"Optimal Parallel String Matching Algorithm on Body Centered Hypercube", International Journal of Mathematics and Computer Science,1 No. 4, pp. 473-484.