Parallel Pattern Identification in Biological Sequences on Clusters [*]

Chun-Hsi Huang, Ratnabali Biswas
Department of Computer Science and Engineering
University of Connecticut
Storrs, CT 06269
huang@cse.uconn.edu

Abstract

This paper presents a low communication-overhead parallel algorithm for pattern matching in biological sequences. Given such a sequence of length n and a pattern of length m, we conclude an algorithm with five computation/communication phases, each requiring $O(n)$ computation time and only $O(p)$ message units. The low communication overhead of the algorithm is essential to achieving reasonable speedups on clusters, where the interprocessor communication latency is usually higher. Previous parallel implementations use straightforward domain decomposition based on existing sequential algorithms and rely on parallel machines with low-latency interconnection network and fast hardware support for processor synchronization.

Keywords: Parallel Algorithm, Pattern Matching, Cluster Computing

1 Introduction

1.1 The Problem

The exact sequence matching problem is to find all occurrences of a given m-character pattern sequence, *PATTERN* $[1 \cdots m]$, in a given n-character target sequence, *TEXT* $[1 \cdots n]$. The biological sequences, such as nucleic acid sequences (DNA and RNA), and protein sequences, are over a fixed and limited alphabet. (The DNA alphabet has four characters, whereas the protein alphabet has twenty.) Therefore, pattern identification in biological sequences is considered a particular instance of the general string matching problem, where the characters in both pattern and target sequences are over a finite but arbitrary alphabet.

A few parallel algorithms for solving exact string matching problems have been proposed. Vishkin [10] presented a linear-time CRCW PRAM algorithm that runs in $O(n/p)$ time using $p = O(n/\log m)$ processors. Breslauer et al. [2] presented an optimal algorithm, also on the CRCW PRAM, that runs in $O(\log\log m)$ time using $n/\log\log m$ processors. String-search VLSI circuits have been investigated by Hirata et al. [5] and Foster et al. [4]. A neural network approach running in constant time using $m \times n$ processing units was investigated by Takefuji et al. [8]. These algorithms are considered impractical because of the extreme fine granularities, $O(n)$ or $O(mn)$, and the disregarding of practical concerns such as remote memory access latencies, synchronization, and inter-task communication costs.

The team led by Douglas Smith at UCSD developed a sequential algorithm for inexact pattern matching in biological sequences. They have also ported the sequential algorithm on the SDSC's Intel Paragon and Cray C-90, using the straightforward domain decomposition for parallelization. Their basic method is to divide the database of known protein sequences and assign each portion to a different Paragon processor. Each processor is also given the entire set of undetermined sequences. The processors then compare the set of undetermined sequences against the portions of the database they have been assigned. However, there are two-fold drawbacks in their approach: (1) Both

[*] Computational resources are provided by the Center for Computational Research (CCR) at the State University of New York at Buffalo.

Intel Paragon and Cray C-90 are equipped with low-latency (costly, too) interconnection network and high speed hardware synchronization support that are not available in the cluster environments, which are more popularly used and affordable by researchers, and (2) Each processor is assumed to be given the entire pattern sequence, which eliminates the scalability of the algorithm and oversimplifies the communication overhead when very long patterns, whose lengths exceed the local memory size of each single processing node, are processed.

In this paper, we present a scalable parallel algorithm for exact pattern matching in biological sequences. We demonstrate the communication efficiency by showing that only five computation/communication phases are needed and the message cost in each phase depends only on the number of processors, instead of either the pattern size m or the sequence size n. The algorithm can easily be adjusted to take care of longer patterns which exceed the local memory size.

1.2 Background

Most of the algorithmic techniques for exact string matching seem to revolve around the *periodic properties* of strings [2, 10]. We introduce some definitions first. The *prefix* of a string u is a substring of u starting at the beginning of u. The concatenation of two strings u and v are denoted as uv. A string u is called a *period* of a string w if w is a prefix of u^k for some positive integer k or equivalently if w is a prefix if uw. The shortest period of w is called the *period* of w. We say w is *periodic* if w is at least twice its period length. Some properties of string periodicity are summarized as follows [6]:

(1) If w has two periods of lengths p and q and $|w| \geq p + q$, then w has a period of length gcd(p, q).

(2) If w occurs in positions p and q of some string and $0 < q - p < |w|$, then w has a period of length $q - p$. Therefore we cannot have two occurrences of the pattern at positions p and q if $0 < q - p < |u|$ and u is the period of the pattern.

A method proposed in [10] efficiently eliminates many possible occurrences by computing array *WITNESS* $(1 \cdots m)$ defined below and using this information for text analysis.

Let u be the period of the pattern w and v be a prefix of w. It follows immediately from the periodicity property that if $|u|$ does not divide $|v|$ and $|v| <$ max$(|u|, |w| - |u|)$, then v is not a period, and hence w is not a prefix of vw. In that case, we can find an index k such that *PATTERN* $[k] \neq$ *PATTERN* $[k - |v|]$. We call this k *a witness to the mismatch of w and vw*, and define *WITNESS* $(|v| + 1) = k$. We are interested only in *WITNESS* (i) for $1 < i \leq |u|$, which by the periodicity properties mentioned above can be based only on the first $2|u| - 1$ characters of the pattern. See Figure 1 for determining the locations of the *WITNESS* array. (If some *WITNESS* (i) is greater than $2|u|$, it can be modified to be in the desired range: Let $r =$ *WITNESS* (i) mod $|u|$; then if $r < i$ set *WITNESS* (i) to $r + |u|$; otherwise we set *WITNESS* (i) to r.)

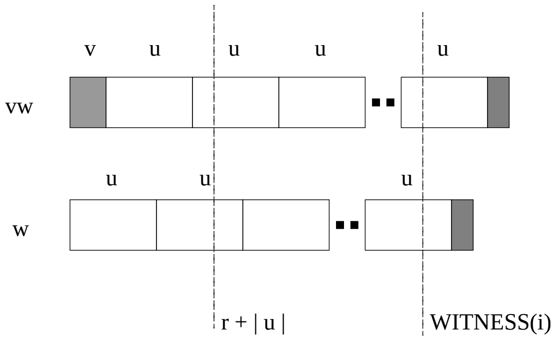

Figure 1. Determining witness locations

Having computed the *WITNESS* array, the following method was further suggested in [10] to eliminate close possible occurrences. Suppose we suspect that the pattern may start at positions i and j of the text where $0 < j - i < |u|$, thus, since $r =$ *WITNESS* $(j - i + 1)$ has been computed, we can find a character in the text in constant time that will eliminate at least one of the possible occurrences. More specifically, since *PATTERN* $[r] \neq$ *PATTERN* $[r - j + i]$, at most one of them can be equal to *TEXT* $[r + i - 1]$. (Refer to Figure 2.)

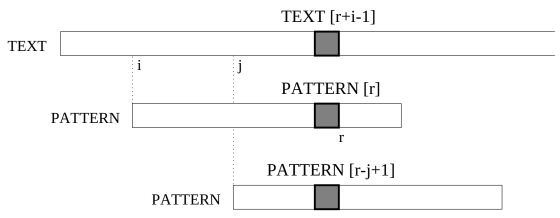

Figure 2. Eliminating close possible occurrences

2 Our Work

2.1 Observations

For the case when the pattern w is not periodic, the pattern can not occur at both positions i and j of the text if $|j-i| < m/2$. This implies that the pattern can occur in the text at most $2n/m$ times. Namely, if the text is evenly divided into blocks of size $m/2$, only one occurrence of the pattern might start in each block. The *WITNESS* array now of size $m/2$ comes in handy eliminating possible occurrences.

The cases with periodic patterns extend from the above approach. Now the pattern w is of the form $u^k v$ where $k > 1$ and v is the proper prefix of u. Consider the prefix *PATTERN* $[1 \cdots 2|u| - 1]$ as a new pattern, which is not periodic, whose occurrences in *TEXT* are to be found using the abovementioned procedure. The occurrences of u^2 hence are easily determined. We call an occurrence of u^2 at position i a *final occurrence* if there is no occurrence of u^2 at position $i + |u|$. For an occurrence of u^2, define its *right match* to be the nearest final occurrence to its right. If u^2 occurs at position i, its right match must be at position $i + l|u|$ for some integer $l \geq 0$. This implies that the number of consecutive occurrences of u starting at position i is $l + 2$. For each final occurrence we need to verify whether v occurs after it. Note that v occurs after each non-final occurrence since v is a prefix of u. This information also helps decide for each occurrence of u^2 whether it is the beginning of an occurrence of the pattern.

Note that identifying the right match for each u^2 is reducible to the problem of finding the closest smaller element for each element in a given sequence. More formally, let $A = (a_1, a_2, \cdots, a_n)$ be an array of elements from a totally ordered domain. For each a_j, $1 \leq j \leq n$, find the nearest element to the right of a_j that is smaller than a_j.

2.2 The Algorithm

Without loss of generality, we define a *p-processor coarse-grained parallel computer* to be a parallel system with p processors interconnected by any communication media (shared memory or any static/dynamic interconnection network), where each processor has $O(\frac{n}{p})$ local memory (n being the problem size). The algorithm performance is described in terms of several network parameters, including p: the number of processors, g: the ratio of communication throughput to processor throughput, and L: the time required to barrier synchronize all or part of the processors. The program proceeds as a series of *supersteps*. In each superstep, a processor may operate only on values stored in local memory. Values sent through the communication network are not guaranteed to arrive until the end of the current superstep. Parameters p, g, and L can be used to estimate the running time of the program whose communication behavior is known. Similarly, a program that has access to these parameters for the machine it is running on can use them to choose between different algorithms. Our purpose is to design a scalable algorithm, minimizing the number of communication supersteps as well as the local computation time. These performance characterizations are similar to the general parallel programming models proposed by Valiant [9], McColl [7] and Dehne [3]. However, our algorithm and analysis do not rely on any particular programming model.

Computing the *WITNESS* array and eliminating close possible occurrences can easily be finished within two supersteps, whether the given *PATTERN* is periodic or non-periodic. Here we focus on solving the problem of finding the closest smaller element for each element in a given sequence.

Given a sequence $A = (a_1, a_2, \cdots, a_n)$ and a p-processor coarse-grained parallel machine, each P_i $(1 \leq i \leq p)$ stores $(a_{\frac{n}{p}(i-1)+1}, a_{\frac{n}{p}(i-1)+2}, \cdots, a_{\frac{n}{p}i})$. For simplicity, we assume that the elements in the sequence are distinct. We define the nearest smaller value to the right of an element to be its *right match*. This problem can be solved sequentially with linear time using a straightforward stack approach. To find the right matches of the elements, we scan the input, keep the elements for which no right match has been found on a stack, and report the current element as a right match for those elements on the top of the stack that are larger than the current element. The left matches can be found similarly. For brevity and without loss of generality, we will focus on finding the right matches.

Some definitions are given below. Throughout the rest of this paper, we use i ($1 \le i \le p$) for processor related indexing and j ($1 \le j \le n$) for array element related indexing.

- For any j:
  - $\mathrm{rm}(j)$ ($\mathrm{lm}(j)$, resp.) $\overset{\text{def}}{=}$ the index of the right (left, resp.) match of a_j.
  - $\mathrm{rmp}(j)$ ($\mathrm{lmp}(j)$, resp.) $\overset{\text{def}}{=}$ the index of the processor containing $a_{\mathrm{rm}(j)}$ ($a_{\mathrm{lm}(j)}$, resp.).
- For any i:
  - $\min(i)$ $\overset{\text{def}}{=}$ the index (in A) of the smallest element in P_i.
  - $\mathrm{rm\_min}(i)$ ($\mathrm{lm\_min}(i)$, resp.) $\overset{\text{def}}{=}$ the index of the right (left, resp.) match of $a_{\min(i)}$ with respect to the array $A_{\min} = (a_{\min(1)}, \cdots, a_{\min(p)})$.
  - $\wp_i \overset{\text{def}}{=} \{P_x | \ \mathrm{rm\_min}(x) = i\}$; $\varphi_i \overset{\text{def}}{=} \{P_x | \ \mathrm{lm\_min}(x) = i\}$.

Based on the above definitions, we observe that $P_{\mathrm{rmp}(\min(i))} = P_{\mathrm{rm\_min}(i)}$ ($P_{\mathrm{lmp}(\min(i))} = P_{\mathrm{lm\_min}(i)}$, resp.) Next we prove a technical lemma used in our algorithm.

Lemma 2.1 *On a p-processor coarse-grained parallel computer, for any i, if $a_{\mathrm{rm}(\min(i))}$ exists and $\mathrm{rmp}(\min(i)) \ne i+1$, then there exists a unique processor $P_{k(i)}$, $i < k(i) < \mathrm{rmp}(\min(i))$, such that $\mathrm{lmp}(\min(k(i))) = i$ and $\mathrm{rmp}(\min(k(i))) = \mathrm{rmp}(\min(i))$. (Symmetrically, for any i, if $a_{\mathrm{lm}(\min(i))}$ exists and $\mathrm{lmp}(\min(i)) \ne i-1$, then there exists a unique processor $P_{k'(i)}$, $\mathrm{lmp}(\min(i)) < k'(i) < i$, such that $\mathrm{lmp}(\min(k'(i))) = \mathrm{lmp}(\min(i))$ and $\mathrm{rmp}(\min(k'(i))) = i$).*

Proof: We show that P_s, where $a_{\min(s)} = \min\{a_{\min(i+1)}, a_{\min(i+2)}, \cdots, a_{\min(\mathrm{rmp}(\min(i))-1)}\}$, is the unique processor described in the lemma. For any s' with $i+1 \le s' < s$, $\mathrm{rmp}(\min(s'))$ must be $\le s$. Similarly, for any s' with $s+1 \le s' < \mathrm{rmp}(\min(i))$, $\mathrm{lmp}(\min(s'))$ must be $\ge s$. This renders P_s the only candidate processor. We can easily infer that $a_{\min(s)} > a_{\min(i)} > a_{\min(\mathrm{rmp}(\min(i)))}$. Since $a_{\min(s)}$ is the smallest element among those in $P_{i+1}, \cdots, P_{\mathrm{rmp}(\min(i))-1}$, we conclude that P_s is the unique processor $P_{k(i)}$ specified in Lemma 2.1. (The symmetric part can be proved similarly.) □

We next outline our algorithm. To begin with, all processors sequentially find the right matches for their local elements, using the stack approach. Those matched elements require no interprocessor communication. We therefore focus on those elements which are not yet matched. The general idea is to find the right matches for those not-yet-matched elements by reducing the original problem to $2p$ smaller "special" instances, and solve them in parallel.

Next we compute the right and left matches for all $a_{\min(i)}$'s. To do this, we first solve this match-finding problem with respect to the array $A_{\min} = (a_{\min(1)}, \cdots, a_{\min(p)})$. Then, for each processor P_i, we define four sequences, $\mathrm{Seq1}_i$, $\mathrm{Seq2}_i$, $\mathrm{Seq3}_i$ and $\mathrm{Seq4}_i$ as follows:

- If $a_{\mathrm{rm}(\min(i))}$ does not exist, then $\mathrm{Seq1}_i$ and $\mathrm{Seq2}_i$ are undefined.
- If $a_{\mathrm{rm}(\min(i))}$ exists and $\mathrm{rmp}(\min(i)) = i+1$, then:

  $\mathrm{Seq1}_i = (a_{\min(i)}, \cdots, a_{\frac{n}{p}i})$, $\mathrm{Seq2}_i = (a_{\frac{n}{p}i+1}, \cdots, a_{\mathrm{rm}(\min(i))})$.
- If $a_{\mathrm{rm}(\min(i))}$ exists and $\mathrm{rmp}(\min(i)) > i+1$, let $P_{k(i)}$ be the unique processor specified in Lemma 2.1. Then: $\mathrm{Seq1}_i = (a_{\min(i)}, \cdots, a_{\mathrm{lm}(\min(k(i)))})$, $\mathrm{Seq2}_i = (a_{\mathrm{rm}(\min(k(i)))}, \cdots, a_{\mathrm{rm}(\min(i))})$.
- If $a_{\mathrm{lm}(\min(i))}$ does not exist, then $\mathrm{Seq3}_i$ and $\mathrm{Seq4}_i$ are undefined.
- If $a_{\mathrm{lm}(\min(i))}$ exists and $\mathrm{lmp}(\min(i)) = i-1$, then:

  $\mathrm{Seq3}_i = (a_{\frac{n}{p}(i-1)+1}, \cdots, a_{\min(i)})$, $\mathrm{Seq4}_i = (a_{\mathrm{lm}(\min(i))}, \cdots, a_{\frac{n}{p}(i-1)})$.

- If $a_{\mathrm{lm}(\min(i))}$ exists and $\mathrm{lmp}(\min(i)) < i-1$, let $P_{k'(i)}$ be the unique processor specified in Lemma 2.1. Then: $\mathrm{Seq3}_i = (a_{\mathrm{rm}(\min(k'(i)))}, \cdots, a_{\min(i)})$, $\mathrm{Seq4}_i = (a_{\mathrm{lm}(\min(i))}, \cdots, a_{\mathrm{lm}(\min(k'(i)))})$.

Note that $\mathrm{Seq1}_i$ and $\mathrm{Seq3}_i$, if they exist, always reside on P_i, $\mathrm{Seq2}_i$, if it exists, always resides on $P_{\mathrm{rmp}(\min(i))}$, and $\mathrm{Seq4}_i$, if it exists, always resides on $P_{\mathrm{lmp}(\min(i))}$. The following two lemmas 2.2 and 2.3 specify how to find the right matches for all unmatched elements. Detailed proofs can be found in [1].

Lemma 2.2 *The right matches of all not-yet-matched elements in* $\mathrm{Seq1}_i$ *lie in* $\mathrm{Seq2}_i$. *The right matches of all not-yet-matched elements in* $\mathrm{Seq4}_i$, *except its first element, lie in* $\mathrm{Seq3}_i$.

Each processor P_i therefore is responsible for identifying right matches for not-yet-matched elements in $\mathrm{Seq1}_i$ and $\mathrm{Seq4}_i$. Again, we apply the sequential algorithm at each processor P_i with respect to the two concatenated sequences, $\mathrm{Seq1}_i \| \mathrm{Seq2}_i$ and $\mathrm{Seq4}_i \| \mathrm{Seq3}_i$.

Lemma 2.3 *All elements will be right-matched after the above-mentioned 2p special problem instances are solved in parallel.*

The following technical lemma, Lemma 2.4, provides the foundation for the communication lower bound in the worst-case analysis.

Lemma 2.4 *1. Suppose that* $\wp_i = \{P_{x_1}, P_{x_2}, \cdots, P_{x_t}\}$ *where* $x_1 < x_2 < \cdots < x_t$. *Then:*

$\mathrm{Seq2}_{x_1} = (a_{\mathrm{rm}(\min(x_2))}, \cdots, a_{\mathrm{rm}(\min(x_1))})$, $\mathrm{Seq2}_{x_2} = (a_{\mathrm{rm}(\min(x_3))}, \cdots, a_{\mathrm{rm}(\min(x_2))})$, $\cdots$, $\mathrm{Seq2}_{x_t} = (a_{\frac{n}{p}(i-1)+1}, \cdots, a_{\mathrm{rm}(\min(x_t))})$.

2. Suppose that $\varphi_i = \{P_{y_1}, P_{y_2}, \cdots, P_{y_s}\}$ *where* $y_1 < y_2 < \cdots < y_s$. *Then:*

$\mathrm{Seq4}_{y_1} = (a_{\mathrm{lm}(\min(y_1))}, \cdots, a_{\mathrm{lm}(\min(y_2))})$, $\mathrm{Seq4}_{y_2} = (a_{\mathrm{lm}(\min(y_2))}, \cdots, a_{\mathrm{lm}(\min(y_3))})$, $\cdots$, $\mathrm{Seq4}_{y_s} = (a_{\mathrm{lm}(\min(y_s))}, \cdots, a_{\frac{n}{p}i})$.

Proof: We only prove Statement 1. The proof of Statement 2 is similar. First observe that, for any $P_x, P_y \in \wp_i$, $x < y$ implies $a_{\min(x)} < a_{\min(y)}$ and $k(x) \le y$. Based on these observations, we have $k(x_l) = x_{l+1}$ for $1 \le l < t$ and $x_t = i-1$. The lemma follows from the definition of Seq2. □

Below we outline this algorithm.

Input: A partitioned into p subsets of continuous elements. Each processor stores one subset.

Output: The right match of each a_i is computed and stored in the processor containing a_i.

1. Each P_i sequentially finds right matches for each element in its local subset.

2. (a) Each P_i computes its local minimum $a_{\min(i)}$.

   (b) All $a_{\min(i)}$'s are globally communicated. (Hence each P_i has the array $A_{\min}$.)

3. Each P_i finds the right match and left match, with respect to $A_{\min}$; and identify the sets $\wp_i$ and φ_i.

4. Each P_i computes $a_{\mathrm{rm}(\min(x))}$ for every $P_x \in \wp_i$ and $a_{\mathrm{lm}(\min(y))}$ for every $P_y \in \varphi_i$.

5. Each P_i determines $\mathrm{Seq1}_i$, $\mathrm{Seq3}_i$ and receives $\mathrm{Seq2}_i$, $\mathrm{Seq4}_i$ as follows:

   (a) Each P_i computes the unique $k(i)$ and $k'(i)$ (as in Lemma 2.1), if they exist, and determines $\mathrm{Seq1}_i$ and $\mathrm{Seq3}_i$.

   (b) Each P_i determines $\mathrm{Seq2}_x$ for every P_x in $\wp_i$, and $\mathrm{Seq4}_y$ for every P_y in φ_i (as in Lemma 2.4).

   (c) Each P_i sends $\mathrm{Seq2}_x$, for every P_x in $\wp_i$, to P_x and $\mathrm{Seq4}_y$, for every P_y in φ_i, to P_y.

6. (a) Each P_i finds the right matches for the unmatched elements in $\mathrm{Seq1}_i$ and $\mathrm{Seq4}_i$

   (b) Each P_i collects the matched $\mathrm{Seq4}_y$'s from all P_y's in φ_i.

2.3 Cost Analysis

We use the term *h-relation* to denote a routing problem where each processor has at most h words of data to send to other processors and each processor is also due to receive at most h words of data from other processors. In each superstep, if at most w arithmetic operations are performed by each processor and the data communicated forms an *h-relation*, then the cost of this superstep is $w + h * g + L$ (the parameters g and L are as defined in Section 1). The cost of an algorithm using S supersteps is simply the sum of the costs of all S supersteps:

$$Program\ cost = comp.\ cost + comm.\ cost + synch.\ cost = W + H * g + L * S,$$

where H is the sum of the maxima of the *h-relation*s in each superstep and W is the sum of the maxima of the local computations in each superstep.

Here we assume the sequential computation time for finding the closest smaller element of each element in a sequence of size n is $T_s(n)$, and the sequential time for finding the minimum of n elements is $T_\theta(n)$. Then the cost breakdown of the algorithm mentioned in Section 2.2 can be derived as in Table 1.

Table 1. Cost Breakdown

	Cost		
Step	comp.	comm.	synch.
1	$T_s(\frac{n}{p})$		
2(a)	$T_\theta(\frac{n}{p})$		
2(b)		pg	L
3	$2T_s(p)$		
4	$\max_i\{T_\theta(\frac{n}{p} + \lvert\varphi_i\rvert) + T_\theta(\frac{n}{p} + \lvert\wp_i\rvert)\}$		
5(a)	$\max_i\{T_\theta(\mathrm{rm\_min}(i) - \mathrm{lm\_min}(i))\}$		
5(b)	$O(\frac{n}{p})$		
5(c)		$\max_i\{(\Sigma_{P_x \in \wp_i}\lvert\mathrm{Seq2}_x\rvert + \Sigma_{P_y \in \varphi_i}\lvert\mathrm{Seq4}_y\rvert)g\}$	L
6(a)	$\max_i\{T_s(\lvert\mathrm{Seq1}_i\rvert + \lvert\mathrm{Seq2}_i\rvert) + T_s(\lvert\mathrm{Seq4}_i\rvert + \lvert\mathrm{Seq3}_i\rvert)\}$		
6(b)		$\max_i\{\Sigma_{P_x \in \varphi_i}\lvert\mathrm{Seq4}_x\rvert g\}$	L

Since $\frac{n}{p} = \Omega(p)$ and $T_s(n) = T_\theta(n) = O(n)$, the computation time in each step is obviously linear in the local data size, namely $O(\frac{n}{p})$. Steps 2(b), 5(c) and 6(b) involve communication. Thus the algorithm takes three supersteps. Based on Lemma 2.4 and the fact that $|\varphi_i| + |\wp_i| \le p$, the communication steps 5(c) and 6(b) can each be implemented by an $(\frac{n}{p} + p)$-relation.

Note that while using the above-mentioned algorithm to identify the right match for each u^2 in a given biological sequence (as described in Section 2.1), all $Seq1(2,3,4)_i$'s are of length $O(1)$, since the algorithm is now to be performed on a (0,1)-sequence, where 0 stands for a u^2 pattern, and 1 for non u^2-patterns. Therefore, the communication cost in each step depends only on the number of processors p. Besides, the two additional supersteps to compute the *WITNESS* array and eliminate close possible occurrences both use $O(p)$-relations in their communication phases. This concludes the following theorem:

Theorem 2.1 *Identifying exact patterns of length m in biological sequences of length n can be done on a p-processor coarse-grained parallel computer in five supersteps using $O(n)$ local computation time and an $O(p)$-relation in each communication phase, provided $p \le n/p$.*

3 Experiments

The experiments have been carried out on two cluster systems provided by the Center for Computational Research at the State University of New York at Buffalo, including (1) an SGI Intel Linux Cluster with 75 dual processor nodes, with each node equipped with 2 Pentium III processors running at 1 GHz, 1 GB of RAM per node, 30GB disk space, Myrinet 2000 - 2 Gb switch and the RedHat 6.2 (Kernel 2.4) as the operating system, and (2) a Sun cluster of 16 Sun Blade 1000 workstations and 48 Sun Ultra 5 workstations configured as a supercomputer, with Myrinet 1000 - 1.28 Gb switch and Solaris 8 as the operating system. Both systems use the Portable Batch System (PBS) for batch job submissions.

A portable implementation of MPI (Message Passing Interface), MPICH, is installed on both the Sun

Cluster and the SGI Intel Cluster. We use MPI_Bcast for broadcasting. Some collective communication routines, such as MPI_Allreduce, MPI_Allgather, are also used for global communication. MPI_Scatter and MPI_Gather are used to distribute input data and collect results. MPI_Send and MPI_Recv are used for two-sided message passing.

We looked up the nucleic acid sequences in the EMBL Nucleotide Sequence Database. Fragments of sequences are retrieved from different species, including archaea (mjannaschii), bacteria (spneumoniae), eukaryota (pfalciparum) and plasmids (atumefaciens). Processing time is measured on 1 to 32 processors on each cluster for primate DNA sequence data $35K$ to $0.5M$ long and patterns around $0.3K$ long, as shown in Figs 3, 4, 5 and 6.

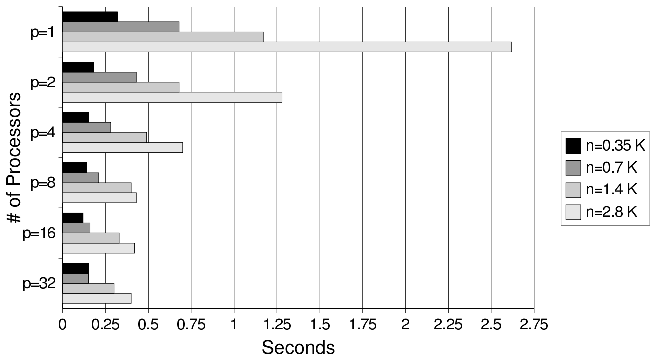

Figure 3. SGI Intel Linux Cluster Running Times (1)

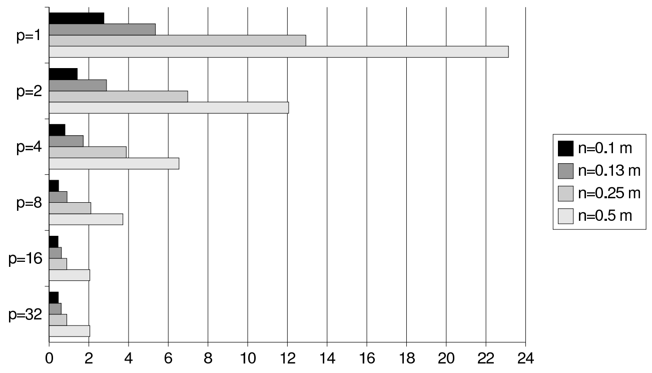

Figure 4. SGI Intel Linux Cluster Running Times (2)

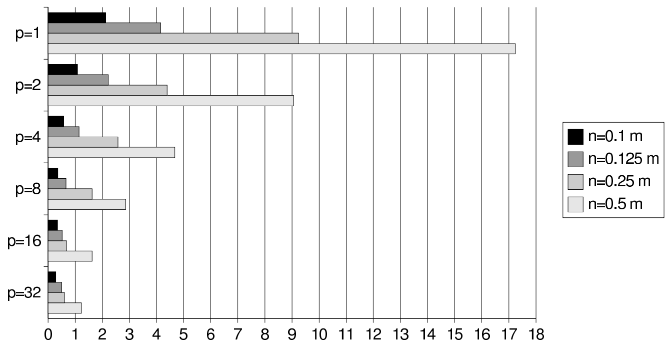

Figure 5. Sun Cluster Running Times (in seconds) (1)

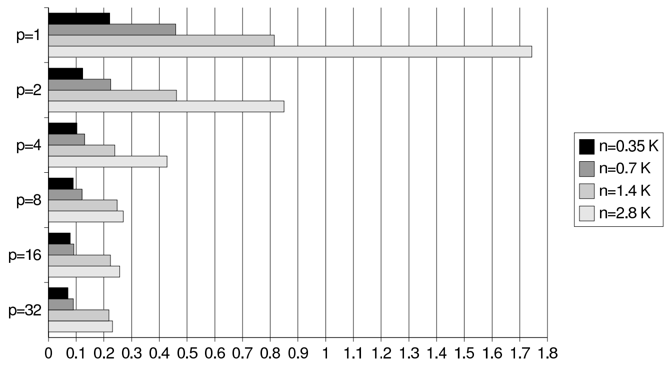

Figure 6. Sun Cluster Running Times (in seconds) (2)

4 Conclusions and Future Work

Interprocessor communication has been shown to become a major bottleneck for parallel algorithm performance. Parallel algorithms should seek to minimize both computation and communication time to be considered practical. Especially for clusters, the superior cost effectiveness and flexibility achieved through which have attracted and enabled (financially) more and more researchers to carry out parallel and distributed computing, the communication efficiency of the parallel algorithm plays a more critical role in algorithm performance. Traditional algorithms designed for high-cost supercomputers do not always perform well on clusters. New and more communication-efficient algorithms are often required while clusters are used. In this paper, we present a communication-efficient parallel algorithm for pattern identification in biological sequences and show reasonable speedups on clusters. This also shows that researchers need not rely on high-speed, high-cost supercomputers to perform computational research in biological fields.

Though exact pattern matching serves as the foundation of parallel pattern identification in biological sequences, identifying genes by comparing their protein sequences to those already identified in databases is more difficult since, in most cases, an exact match is not desired. Biologically, genes can contain insertions, deletions, and local changes while still carrying out the same (or a closely related) biological function. Therefore, providing a scalable parallel approximate pattern matching with predictable communication efficiency is of higher practical relevance. The authors have been developing such algorithms and will experiment on the low-cost cluster environments.

5 Acknowledgments

The authors appreciate the technical support and code optimization by the technical staff at the Center for Computational Research at SUNY Buffalo. We would also like to thank the anonymous referees for many useful suggestions.

References

[1] O. Berkman, B. Schieber, and U. Vishkin. Optimal Doubly Logarithmic Parallel Algorithms Based on Finding All Nearest Smaller Values. *Journal of Algorithms*, 14:344–370, 1993.

[2] D. Breslauer and Z. Galil. An Optimal $O(\log \log n)$ Time Parallel String Matching Algorithm. *SIAM J. Computing*, 19:1050–1058, 1990.

[3] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable Parallel Geometric Algorithms for Coarse Grained Multicomputers. In *Proc. 9th ACM Annual Computational Geometry*, pages 298–307, 1993.

[4] M. J. Foster and H. T. Kung. The Design of Special-Purpose VLSI Chips. *IEEE Computer*, 13:26–40, 1980.

[5] M. Hirata, H. Yamada, H. Nagai, and K. Takahashi. A Versatile Data String-Search VLSI. *IEEE Solid-State Circuits*, 23:329–335, 1988.

[6] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[7] W. F. McColl. Scalable Computing. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 46–61. Springer-Verlag, Berlin, 1995.

[8] Y. Takefuji, T. Tanaka, and K. Lee. A Parallel String Search Algorithm. *IEEE Transactions on System, Man, and Cybernetics*, 22(2):332–336, 1992.

[9] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, 1990.

[10] U. Vishkin. Optimal Parallel Pattern Matching in Strings. *Information and Control*, 67:91–113, 1985.