pFANGS: Parallel High Speed Sequence Mapping for Next Generation 454-Roche Sequencing Reads

Sanchit Misra[1], Ramanathan Narayanan[2], Wei-keng Liao[3], Alok Choudhary[4]
Electrical Engineering and Computer Science
Northwestern University
Evanston, IL, USA 60208
Email: [1]*smi539@eecs.northwestern.edu,*
[2]*ran310@eecs.northwestern.edu,*
[3]*wkliao@eecs.northwestern.edu,*
[4]*choudhar@eecs.northwestern.edu*
Simon Lin
Robert H. Lurie Comprehensive Cancer Center
Northwestern University
Chicago, IL, USA 60611
Email: S-Lin2@northwestern.edu

***Abstract*—Millions of DNA sequences (reads) are generated by Next Generation Sequencing machines everyday. There is a need for high performance algorithms to map these sequences to the reference genome to identify single nucleotide polymorphisms or rare transcripts to fulfill the dream of personalized medicine. In this paper, we present a high-throughput parallel sequence mapping program pFANGS. pFANGS is designed to find all the matches of a query sequence in the reference genome tolerating a large number of mismatches or insertions/deletions. pFANGS partitions the computational workload and data among all the processes and employs load-balancing mechanisms to ensure better process efficiency. Our experiments show that, with 512 processors, we are able to map approximately 31 million 454/Roche queries of length 500 each to a reference human genome per hour allowing 5 mismatches or insertion/deletions at full sensitivity. We also report and compare the performance results of two alternative parallel implementations of pFANGS: a shared memory OpenMP implementation and a MPI-OpenMP hybrid implementation.**

***Keywords*-sequence mapping; next generation sequencers; 454 sequencers; parallel computing;**

I. Introduction

DNA sequencing is used in a variety of applications in medicine, such as SNP discovery, comparative genomics, gene expression, genotyping, metagenomics and personal genomics. Recent developments in Next Generation Sequencing (NGS) technology have resulted in affordable desktop-sized sequencers with low running costs and high throughput. These sequencers produce small fragments (reads) of the genome being sequenced as a result of the sequencing process. For example, the Illumina-Solexa system can generate 50 million sequences of length 30-50 nucleotides in just 3 days [10]. The Roche-454 system can generate $400,000$ sequences of length 250-500 nucleotides in a 7.5 hour run [20]. The ABI-SOLiD system can also generate data at a similar rate [10]. NGS is a rapidly advancing field with a very high rate of increase in throughput. It is speculated that eventually the running costs of sequencing a genome will be as low as $1000 [24]. This will trigger the use of such systems in laboratories around the world. The computational demands for processing NGS data are tremendous and far exceed current capabilities. In fact, without substantial advances in high-performance, scalable algorithms, very little progress would be made to extract knowledge from such a rich set of data. Therefore, there is a need to design powerful algorithms and systems which can efficiently handle the computational challenges posed by NGSs.

An important step in many of the applications mentioned above is mapping a set of read sequences to a canonical genomic database. A typical genomic database, for instance, the human genome, can be 3 billion nucleotides in length. The length of the read sequences depends on the sequencing technology. In this article, we focus on the mapping of the longer reads produced by Roche-454 system. A 454 sequencer was recently used for sequencing the DNA sequence of James D. Watson to 7.4 fold redundancy in just two months [27]. The authors used BLAT [11] to map the 454 reads to a reference genome, which is not at par with the sequencing speed. Moreover, BLAT is designed for local alignment, while sequence mapping requires the entire length of a query to be mapped. There have been considerable efforts to develop faster sequence mapping tools which can match the speed of Next Generation Sequencers, but most of them have been for reads generated by Illumina-Solexa machines (for example, ELAND, MAQ [13], SOAP [14] and BowTie [12]). Even though 454 sequencers are

widely used by researchers, there has not been sufficient research to develop faster tools for mapping 454 reads. To the best of our knowledge, the only algorithms which are specifically designed for 454 data are BWA and FANGS [17]. BWA is an unpublished package written by the authors of Maq. BWA is based on Burrows-Wheeler Transform (BWT). It supports gapped global alignment with respect to queries and is one of the fastest short read alignment algorithms while also finding suboptimal matches. However, [17] demostrates that BWA suffers from low sensitivity. FANGS dynamically reduces the search space by using q-gram filtering and the pigeonhole principle, to rapidly map 454 reads onto a reference genome. FANGS allows a large number of mismatches and insertion/deletions. It tries to find all matches of a read in the reference genome and maps nearly 100% of the reads. FANGS is shown to be upto an order of magnitude faster than the state-of-the-art techniques for 454 reads as long as the number of mismatches and insertion/deletions allowed is small. However, the execution time of FANGS increases dramatically with the increase in number of mismatches and insertion/deletions allowed. Therefore, there is a need to design powerful high throughput parallel programs and systems which can efficiently and accurately map 454 reads.

To the best of our knowledge, very little work has been done to parallelize sequence mapping algorithms. The approaches to parallelize high throughput sequence mapping tools typically include running a separate instance of the tool on each compute node and dividing the queries equally among these nodes. If the genome database occupies only a small amount of memory, this approach can give close to linear speedups. However, for large databases, like the human genome, the amount of memory required may not be available on one node. Moreover, the large memory requirement is also prone to having cache-misses and page-faults. MPIBLAST [9], one of the most prominent parallel sequence alignment tools, tries to solve this problem by distributing the database across processors. It uses a master-slave paradigm in which the master assigns each slave a batch of queries to process. Once a slave finishes its batch of queries, it requests the master for more queries. If there are more queries to be processed, the master sends another batch of queries to the worker. More recently, [6] focusses on the short read sequence mapping problem and discusses six different strategies of parallelization of hashing and indexing based algorithms. While the sequential and parallel tools mentioned above demonstrate a significant performance improvement over earlier sequence mapping tools, the throughput requirement of NGSs is also increasing rapidly and developing faster tools is constantly needed.

In this article, we describe our high-throughput parallel sequence mapping program pFANGS, a **p**arallel **F**ast **A**lgorithm for **N**ext **G**eneration **S**equencers. pFANGS is a parallel implementation of FANGS. We discuss three parallel implementations of FANGS: (a) a shared memory task-parallel implementation using OpenMP, (b) an MPI-OpenMP task-parallel hybrid implementation, and (c) pFANGS: a fully data- and task-parallel MPI implementation (Section VI). The first two implementations are based on query segmentation principle. The third implementation fully distributes the computational workload and data among all the processes and employs load-balancing mechanisms to ensure better process efficiency. We present the performance results in Section VII.

In comparison with existing tools, the most significant features of pFANGS are:

- High flexibility. It allows a large number of mismatches and insertions/deletions in mapping.
- High Sensitivity. It tries to find all the matches for each query and maps nearly 100% of the queries.
- Ability to handle large datasets. Using pFANGS, we have mapped approximately 31 Million queries of length 500 each to a reference human genome per hour allowing 5 mismatch or insertion/deletion at full sensitivity.
- Nearly linear scalability. With 512 processors, pFANGS achieves a speedup of upto 225 over the the time taken with 2 processors.

The remainder of the paper is organized as follows. We give a formal definition of the problem in Section II followed by a background in Section III. Section IV describes our key idea. Section V describes the sequential FANGS algorithm in detail. We describe our parallel implementations in Section VI followed by results in Section VII and conclusion in Section VIII.

II. Problem Definition

The sequence alignment problem has been studied in great detail in literature. However, it has become even more significant in the wake of the new sequencing technologies in the form of Next Generation Sequencers. Consider, for example, using a 454 sequencer [20] to sequence a human genome. It produces a collection of small DNA fragments called reads. These reads are about 250-500 bases in length. Now, we need to search a read, Q, in the database consisting of a reference human genome, G. The database and the reads are from the genomes of different human beings. Moreover, there can be sequencing errors also. Hence, we may not be able to find an exact match of the read Q in the database. However, since both G and Q are from the genomes of the same species, we should be able to find a near-exact match of Q in G. Hence, while searching for Q in G, we only look for alignments which have less than a certain number of mismatches and insertion/deletions.

Given a string S over a finite alphabet Σ, we use $|S|$ to refer to the length of S, $S[i]$ to denote the i^{th} character of S and $S[i, j]$ to denote the substring of S which starts at position i and ends at position j. A *q-gram* of S is defined

as a substring of S of length $q > 0$. The *unit cost edit distance* between two strings S_1 and S_2 is defined as the minimum number of substitutions, insertions and deletions required to convert S_1 to S_2 [23]. We will use $edist(S_1, S_2)$ to refer to the *unit cost edit distance* between S_1 and S_2. It can be calculated by using Needleman-Wunsch algorithm in $O(|S_1||S_2|)$ time [18]. For a string S, we will refer to the natural decimal representation of S over Σ as $dec(S, \Sigma)$. For example, for $\Sigma = \{A, C, G, T\}$, the nucleotides A, C, G, T are mapped to the numbers $0, 1, 2, 3$ respectively. Therefore:

$$f(A) = 0, f(C) = 1, f(G) = 2, f(T) = 3,$$

And, $dec(S, \{A, C, G, T\}) = \sum_{i=0}^{|S|-1} 4^i f(S[i])$

This brings us to the formal definition of the sequence mapping problem. We can represent every genomic sequence as a string over the alphabet $\Sigma = \{A, C, G, T\}$. Given a genomic database G of subject sequences $\{S_1, S_2, \cdots, S_l\}$, a query sequence (read) Q of length m and an integer n, we are required to find all substrings from G, such that for each substring B, $edist(B, Q) \leq n$. We will denote the integer n as the *maxEditDist* parameter.

III. Background

The classical approach to sequence alignment involves several variants of dynamic programming, the most prominent of which are the algorithms of Needleman-Wunsch [18] and Smith-Waterman [26]. Dynamic programming is excessively expensive in terms of time and space for larger databases like the human genome, and this has led to the development of faster hash-table based heuristic methods like FASTA [21], BLAST [4], BLAT [11] and SSAHA [19] and greedy algorithm based methods like MegaBLAST [29]. BLAST has been the most popular tool for sequence alignment. However, it usually takes several hundreds of days for the data generated by the latest sequencers in just a few hours and hence is not a feasible option.

Recently, the advent of Next Generation Sequencers has inpired the researchers to develop high-speed sequence mapping tools. Some of the most prominent recent tools for sequence mapping include ELAND, SHRiMP [1], RMAP [25], SOAP [14], MAQ [13], SWIFT [23], SeqMap [10], BowTie [12], GMAP [28], Mosaik [2], BWA [3] and SSAHA2 [19]. The primary idea behind these algorithms is the following lemma from [22].

Lemma 1: If two strings $A[1..m]$ and $B[1..m]$ have at most n mismatches and $p = \lfloor \frac{m}{n+1} \rfloor$, then there must be an integer x such that $A[x : x + p - 1] = B[x : x + p - 1]$. In other words, A and B share a common substring of length p.

We can easily derive the following corollary from Lemma 1.

Corollary 1: Given a genome $G[1..L]$ and a query $Q[1..m]$ $(L > m)$, if there is a substring α of G, such that α and Q match with an edit distance of at most n and $p = \lfloor \frac{m}{n+1} \rfloor$, then there must exist x, y such that $G[x : x + p - 1] = Q[y : y + p - 1]$.

The substring α is called a homologous region of Q in G. Most sequence mapping algorithms first find regions in the database that share a substring of length p with the query in order to get the locations of all the candidate homologous regions of G which can potentially have an edit distance of less than the *maxEditDist*. These candidate regions are then checked using an accurate algorithm to verify if the edit distance is indeed less than *maxEditDist*.

FANGS also uses the above corollary to filter out non-homologous regions. A *q-hit* between two strings S_1 and S_2 is defined as the tuple (x, y) such that $S_1[x : x + q - 1] = S_2[y : y + q - 1]$. FANGS creates an index of all *q-grams* in the database, called *q-gram index*, and uses it to find all *q-hits* of Q and G. Using the *q-hits* and corollary 1, it identifies regions in the reference genome that can potentially be homologous to the query. FANGS further analyzes these regions to check if the edit distance is within limits. We explain FANGS in complete detail in Section V.

Since BLAST is the most popular sequence alignment tool, several attempts have been made to parallelize it. Early attempts at parallelization have used query segmentation approach [7], [8], where individual compute nodes independently search disjoint sets of queries against the whole database. This technique works well when the database can fit in the memory of a compute node. However, this approach suffers from caching and paging overheads when the database requires large amount of memory as the database is randomly accessed. This led to the development of database segmentation [5], [9], [15], [16], where the genomic database is evenly distributed across compute nodes. This reduces the caching and paging overheads as each compute node uses a small amount of memory for its part of the database. Database segmentation divides the database into mutually exclusive parts and assigns one part to each node. Every node searches for the query in its own part of the database and results from all processes are merged in the end. In particular, mpiBLAST [9] uses a master-worker paradigm in which the master gives each worker a batch of queries to process. Once a worker finishes its batch of queries, it notifies the master. If there are more queries to be processed, the master sends another batch of queries to the worker.

Six parallelization methods for short sequence mapping algorithms are proposed in [6]. The methods are general and should work for most hashing and indexing based algorithms. The first three methods are: (i) Partition Read Only (PRO) partitions the reads into equal parts and sends each part to one processing node. Each node keeps its own copy of the index of the whole genome. This method is useful to match very large number of reads to a relatively short reference genome. If the genome is large, the index may not fit in the memory available on one node. (ii) Partition Genome Only (PGO) partitions the genome equally

amongst all processing nodes. Each node creates the index of only the assigned part of the genome and processes all reads against it. PGO performs well when the genome size is large and the number of reads is small but does not scale well if the number of reads is large. (iii) Suffix Based Assignment (SBA) assigns a set of suffixes to each processing node and makes them only responsible for genome and read sequences that end with the corresponding suffixes. The other methods are combinations of the first three methods. The authors compare the scalability of the proposed methods using theoretical analysis and experimentation using SOLiD System Color Space Mapping Tool.

IV. Key Idea

All the parallelization methods mentioned above are coarse-grained in the sense that they treat the sequential algorithm as one application and run separate instances of the sequential application on different parts of the queryset and the database. Many of the hashing and indexing based algorithms have essentially four stages:

- Creating the index of the database.
- Finding hits in the database based on the index.
- Reducing the number of hits to be processed using a filtering criteria.
- Processing the list of hits remaining to get the final mapping.

In this paper, we exploit this generic structure to achieve a more fine-grained parallelization. We parallelize each stage separately and perform load balancing between stages to achieve very high-throughputs. We also explore the possibility of using shared memory OpenMP programming for parallelization of hashing based sequence mapping algorithms. We discuss the implementation and compare the performance results of two alternative parallel implementations of pFANGS: a shared memory OpenMP implementation and a MPI-OpenMP hybrid implementation. Although we demonstrate our parallelization techniques using FANGS, they should be applicable to most hashing and indexing based algorithms.

V. FANGS

The FANGS algorithm can be divided into the following stages: 1) Creation of the q-gram index, 2) Finding q-hits using the q-gram index, 3) Finding potential holomologous regions and 4) Verifying the potential homologous regions to check if the edit distance is within limits.

A. Preprocessing step: Creation of the q-gram index

Here we describe the construction of the *q-gram index*. We preprocess the sequences in the database by breaking them into non-overlapping *q-grams* and store the location of each *q-gram* in the *q-gram index*. We will refer to the *q-gram index* as the *index-table*. We refer to the size of these non-overlapping *q-grams*, q, as *tileSize*. Each *q-gram* t can be uniquely mapped to a corresponding integer $dec(t, \Sigma)$ as defined in Section II.

Algorithm $GetHits(seq, dbIndex, q)$
Input:
seq : the query sequence.
$dbIndex$: The *index-table* of the database obtained after preprocessing
q : *q-gram* size used for the creation of *q-gram index*.
Output:
$hitList$: List of all *q-hits*.
$wildCardList$: List of query indices for which the *q-gram* starting at that index is more frequent than *maxFreq*

```
1:  hitList ⇐ φ
2:  wildCardList ⇐ φ
3:  for i in 0 to len(seq) − q + 1 do
4:    locations, numLocations ⇐ getDBLocations(
      dbIndex, seq[i : i + q − 1])
5:    if numLocations = −1 then
6:      addToList(wildCardList, i)
7:    else
8:      for j in 1 to numLocations do
9:        hit.dStart = locations[j]
10:       hit.qStart = i
11:       addToList(hitList, hit)
```

Figure 1. Algorithm for obtaining *q-hits*

For each *q-gram* t, we calculate two values: (1) $tileHead(t) = dec(t[1:12], \Sigma)$ and (2) $tileTail(t) = dec(t[13:q], \Sigma)$. The *index-table* consists of two arrays. The first array *occurrenceTable* stores (i) the location of $t[1:12]$ in the database G and (ii) $tileTail(t)$ for each *q-gram*. Hence, *occurrenceTable* contains the concatenation of lists $L(t[1:12]) = \{i, tileTail(G[i : i+q-1]) | G[i : i+11] = t[1:12]\}$, where t is a *q-gram*, that is $t \in \Sigma^q$. For each *q-gram* $t \in \Sigma^q$, the position $tileHead(t)$ in the second array *lookupTable* contains the pointer $p(t)$, which points to the beginning of the correponding list $L(t[1:12])$ in the *occurrenceTable*; and the count $c(t)$ of the number of occurences of $t[1:12]$ in G. Hence the length of the *lookupTable* is $|\Sigma|^{12}$. In order to find hits for a *q-gram* t, it first indexes the *lookupTable* with $tileHead(t)$. Let $L(t[1:12])$ be the corresponding list. The *q-hits* can be found by traversing through the list and outputting those locations for which $tileTail(t)$ matches.

The *index-table* is created in two passes. In the first pass, we find the number of non-overlapping occurrences of each *q-gram* in the database, so that we can allocate appropriate amount of memory to the *occurrenceTable* and calculate the pointer positions for *lookupTable*. In the second pass, we fill the array *occurrenceTable* with appropriate values for the *q-grams*. Note that creation of index need to be done only once for a given value of q. After that, we can process any

Algorithm $CheckRegions(seq, n, db, regionList)$
INPUT:
seq : the query sequence.
n : maximum edit distance allowed in the mapping (*maxEditDist*).
db : genomic database.
$regionList$: list of candidate homologous regions.
Output:
All mappings of the query sequence seq in the database db

1: **for all** $region$ in $regionList$ **do**
2: **if** $edist(region, seq) \leq n$ **then**
3: $outputmapping(region, seq)$

Figure 3. Algorithm for checking the candidate homologous regions to see if they are within the *maxEditDist*

number of queries.

The above structure of q-gram index is similar to the one used by [11]. The main difference is that [11] stores the locations of all the q-grams for $q > 12$. But this greatly reduces the speed of the algorithm. We ignore all the *q-grams* with number of occurrences in the database greater than a certain threshold frequency, say *maxFreq*. Hence for such highly frequent *q-grams*, FANGS makes the locations corresponding to frequently occurring *q-grams* in the *occurrenceTable* as -1. This filtering step helps us in two ways. First, it reduces the index table size. Second, it avoids unnecessary false hits due to repetition of DNA thereby improving efficiency. Due to this technique, we need only 1GB memory to store the index of the human genome that is 3GB in size.

B. Using index-table to map sequences

A substring of size p has at least $\lfloor \frac{p-(q-1)}{q} \rfloor$ *q-grams*. Substituting p from corollary 1, we get:

Corollary 2: Given a query $Q[1..m]$ and database $G[1..L]$ ($m < L$). For all substrings α of G such that $edist(Q, \alpha) < n$, $\exists x, y$ such that $Q[x : x+q-1] = \alpha[y : y+q-1], Q[x+q : x+2q-1] = \alpha[y+q : y+2q-1], \cdots, Q[x+(T-1)q : x+Tq-1] = \alpha[y+(T-1)q : y+Tq-1]$, where T is given by:

$$T = \lfloor \frac{\lfloor \frac{m}{n+1} \rfloor - (q-1)}{q} \rfloor$$

We use the above corollary to dynamically reduce the search space and map query sequences at a very high speed. For each query, we first find the list of candidate homologous regions and then verify each region to check whether the edit distance is within *maxEditDist*. The complete algorithm can be divided in three steps:

- *GetHits:* First we find all the *q-hits* of the query in the database. Each *q-hit* consists of two values - starting position of the *q-gram* in the query ($qStart$) and in the database ($dStart$). The algorithm is given in Figure 1. The algorithm takes each overlapping *q-gram* in the database and finds the locations of all occurrence of the *q-gram* in the database using the *index-table*. The algorithm creates a hit with each location and adds it to the hitList. For some of the *q-grams* which are more frequent than the *maxFreq* parameter, the number of locations $numLocations$ is returned as -1. We add all such query indices to the *wildCardList*.
- *FindRegions:* In the next step, we stitch together *q-hits* which are adjacent to each other both in the query and the database to create maximal *matchingBlocks*. We define a *block* as a contiguous sequence of *q-grams* in a sequence. Also, a *matchingBlock* is defined as a *block* in the query that perfectly matches a *block* of same length in the database. We represent a *matchingBlock* as a tuple $(qStart, dStart, len)$, where $qStart$ and $dStart$ are the starting locations of the first *q-gram* of the *matchingBlock* in the query and the database respectively and *len* is the number of *q-grams* in the *matchingBlock*. A maximal *matchingBlock* is a *matchingBlock* which will result in a mismatch if extended any further on either side. The algorithm is given in Figure 2.
As the hits are obtained in the order of increasing

Algorithm $MapSequences(seqList, seqCount, db, dbIndex, n, q, outFile)$
INPUT:
$seqList$: list of query sequences.
$seqCount$: total number of query sequences
db : genomic database.
$dbIndex$: The *index-table* of the database obtained after preprocessing
n : maximum edit distance allowed in the mapping (*maxEditDist*).
q : *q-gram* size used for the creation of *index-table*.
$outFile$: file where output has to be written.
Output:
All mappings of each query sequence in $seqList$ in the database db

1: **for all** seq in $seqList$ **do**
2: $hitList \Leftarrow GetHits(seq, dbIndex, q)$
3: $T = \lfloor \frac{\lfloor \frac{m}{n+1} \rfloor - (q-1)}{q} \rfloor$
4: $matchingBlockList \Leftarrow GetMatchingBlocks(hitList, wildCardList, q, T)$
5: $regionList \Leftarrow GetRegionList(matchingBlockList, q, size)$
6: $CheckRegions(seq, n, db, regionList)$

Figure 4. The sequence mapping algorithm

Algorithm $FindRegions(hitList, wildCardList, q, size$,n,$T(>1))$
Input:
$hitList$: List of all *q-hits* ¡d_j, r_k¿, where d_j is the $dStart$ value and r_k is the $qStart$ value.
$wildCardList$: List of query indices for which the *q-gram* starting at that index is more frequent than *maxFreq*
q : *q-gram* size used for the creation of *index-table*.
$size$: size of the query sequence.
n : maximum edit distance allowed in the mapping (*maxEditDist*).
T : T as given in corollary 2
Output:
$regionList$: list of candidate homologous regions.

1: $sortList(hitList, dStart)$
2: Let $d_1, d_2, ..., d_x$ be the distinct $dStart$ values in ascending order.
3: **for all** i **do**
4: Let $< d_i, r_{i,1} >$,$< d_i, r_{i,2} >$, $\cdots$, $< d_i, r_{i,y} >$ be the hits containing d_i
5: Let $< d_{i+1}, r_{i+1,1} >$,$< d_{i+1}, r_{i+1,2} >$, $\cdots$, $< d_{i+1}, r_{i+1,z} >$ be the hits containing d_{i+1}
6: **if** there exists a, b such that $r_{i+1,b} - r_{i,a} = d_{i+1} - d_i$ **then**
7: **if** d_{i+1} and d_i are either adjacent in the database or are separated only by highly frequent *q-grams* **then**
8: add $< d_{i+1}, r_{i+1,b} >$ to the *matchingBlock* containing $< d_i, r_{i,a} >$
9: **if** size of *matchingBlock* $\geq T$ **then**
10: add the *matchingBlock* to the *matchingBlockList*
11: $regionList \Leftarrow \phi$
12: **for all** $blockHit$ in *matchingBlockList* **do**
13: $region.dBegin \Leftarrow blockHit.dStart - blockHit.qStart - n$
14: $region.dEnd \Leftarrow blockHit.dStart - blockHit.qStart + size - 1 + n$
15: $addToRegionList(regionList, region)$

Figure 2. Algorithm for stitching together q-hits to find candidate homologous regions

$qStart$ values, the hitList is already sorted according to the $qStart$ values. Now we sort the list according to database positions ($dStart$ values). As a result, the list is now sorted according to the $dStart$ values and for each $dStart$ value, it is sorted according to the $qStart$ values. Let $d_1, d_2, ..., d_x$ be the distinct $dStart$ values in ascending order. For each i, we check if $d_{i+1} - d_i$ is equal to q; i.e.; they are the neighboring *q-grams* in the database. Let $< d_i, r_{i,1} >$,$< d_i, r_{i,2} >$, $\cdots$, $< d_i, r_{i,y} >$ be the hits containing d_i and $< d_{i+1}, r_{i+1,1} >$,$< d_{i+1}, r_{i+1,2} >$, $\cdots$, $< d_{i+1}, r_{i+1,z} >$ be the hits containing d_{i+1}. If $d_{i+1} - d_i = q$, then we search for a pair of hits $< d_i, r_{i,a} >$ and $< d_{i+1}, r_{i+1,b} >$ such that $r_{i+1,b} - r_{i,a} = q$. This means the pair of hits has neighboring tuples both in the query and the database. Hence, we have a *matchingBlock* of length 2 *q-grams*. This way we keep on combining hits to form *matchingBlocks*. If the length of a *matchingBlock* $\geq T$, we store it in the *matchingBlockList*.
This technique does not capture all the *matchingBlocks* with length greater than or equal to T because we do not store the database locations of very frequently occuring *q-grams*. Since multiple *matchingBlocks* may contain the frequently occuring *q-grams*, this leads to some of them not being detected. In order to solve this problem, we give a wildcard to all the frequently occuring *q-grams*. According to this wildcard, the frequently occuring *q-grams* can be part of any *matchingBlock*. Therefore, if two *matchingBlocks* are separated only by frequently occuring *q-grams*, the two of them together with the frequently occuring *q-grams* are combined to form one big *matchingBlock*.
Once we have the *matchingBlockList*, we extend each *matchingBlock* in the list to create a candidate homologous region. We also keep a buffer of size n on either side to account for gaps in the alignment. Each region consists of two values - beginning location, $dBegin$ and end location, $dEnd$ of the region in the database. Hence the beginning of the *matchingBlock* would be:

$$dBegin = dStart - qStart - n$$

and the end would be:

$$dEnd = dStart - qStart + size - 1 + n$$

Thus the homologous region is created by extending the *matchingBlock* on either side to cover the whole query and adding a buffer of n bases on either side. The function *addToRegionList* ensures that we do not

add two regions which have a huge overlap as they will result in the same mapping. If one region completely covers another region, we only include the former. Moreover, if two regions have an overlap of more than a certain value, then we merge them together into one region. This is done to avoid multiple outputs for the same homologous region.

- *CheckRegions:* The potential homologous region is further processed by using an adaptation of the Needleman-Wunsch algorithm to check if the homologous region actually has an edit distance $\leq n$ as given in Figure 3.

The complete algorithm is as given in Figure 4. The novelty of FANGS lies in the fact that it stitches the hits obtained into contiguous blocks of query which exactly match a contiguous block in the database. Another important contribution is that it gives a wildcard to all highly frequent *q-grams*. Hence, even though we do not store the highly occurring *q-grams* in the *index-table*, we can still map queries with 100% sensitivity. The above algorithm, though very fast, is still not at par with the current sequencing speeds. Hence there is a need to parallelize the algorithm. For the human genome, the algorithm needs 1GB memory for the *index-table*. Moreover, it also needs to keep the database G in memory as it needs the database to create the *index-table* and also to examine the candidate homologous regions in the end. Hence, the algorithm requires about 4.5GB memory for mapping reads to a reference human genome. This large amount of memory usage can potentially lead to a number of cache-misses and page-faults due to random access and hence slows down the execution. Hence, we need to distribute both the index and the database across processor nodes in order to run it efficiently on a cluster.

VI. Parallel Approaches

In this section, we investigate three parallelization approaches for FANGS: (a) a shared memory task-parallel implementation using OpenMP, (b) an MPI-OpenMP task-parallel hybrid implementation, and (c) a fully data- and task-parallel MPI implementation called pFANGS.

A. Shared Memory Parallel Implementation

Since the human genome database occupies significant amount of memory, a shared memory parallel implementation seems like a natural choice as we can load both index and database in the shared memory. The target platform is the parallel machine equipped with multiple CPU cores sharing a large sized main memory. We adopt the query segmentation strategy in which each thread takes a subset of the queries and processes them independently. In other words, we parallelize the outermost loop of the sequential algorithm. The algorithm divides the queries equally amongst all threads. All the threads access the same copy of the database and the *index-table* stored in shared memory. Each thread uses FANGS to perform the alignments and stores the results in the *localOutputList* data structure. The *globalOutputList* is shared across all threads. Once a thread finishes processing all its queries it acquires exclusive access to the *globalOutputList* and concatenates its *localOutputList* to it. After all the children threads have merged their results to the global list, the parent thread writes all the outputs to the output file.

Accessing the shared data structures, such as genome database, *index table*, and *globalOutputList*, must be serialized in order to achieve data atomicity and cache coherence. This can become a major performance bottleneck as the number of threads increase.

B. MPI-OpenMP Hybrid Implementation

In order to overcome the drawbacks of the shared memory approach, we have also designed an MPI-OpenMP hybrid approach. This approach targets the parallel computers equipped with multiple SMP compute nodes interconnected with a high speed communication network and the memory in each node is not directly accessible to a remote node. In this hybrid approach, the *index-table* is built independently in each compute node. All processes running on the same node share the *index-table* by accessing the shared memory. The queries are evenly assigned to the MPI processes across all compute nodes. The alignment outputs produced at each node are saved locally, which are later sent to the root process. The root process concatenates all the partial results and writes to the output file.

There is a single MPI process running on each compute node and OpenMP is used to enable thread parallelism using all cores in each node. Even though the memory size per processing core is small, the combined shared memory of all cores on a node is sufficient to hold both the database and the *index-table*. Compared to the shared-memory method, this approach alleviates the congestion problem by reducing the number of processes accessing the shared memory. However, since we are using more than 4GB of the memory on each node, the problem of cache misses is still unsolved.

C. pFANGS: Fully Data and Task Distributed MPI Implementation

The above hybrid implementation may not be very scalable as it requires about 4.5GB memory per node. In this section we describe a completely task and data parallel MPI implementation, named pFANGS.

The idea is to distribute the entire database and the *index-table* equally among all MPI processes. Recall that the genomic database is available as a set of sequences $\{S_1, S_2, \cdots, S_l\}$. As a preprocessing step, for each sequence S_i,we remove $|S_i| mod\ q$ nucleotides from the end so that the length of each sequence is a multiple of q. Then we store all the sequences in a file named *genomeFile* by concatenating the sequences. To keep track of the positions

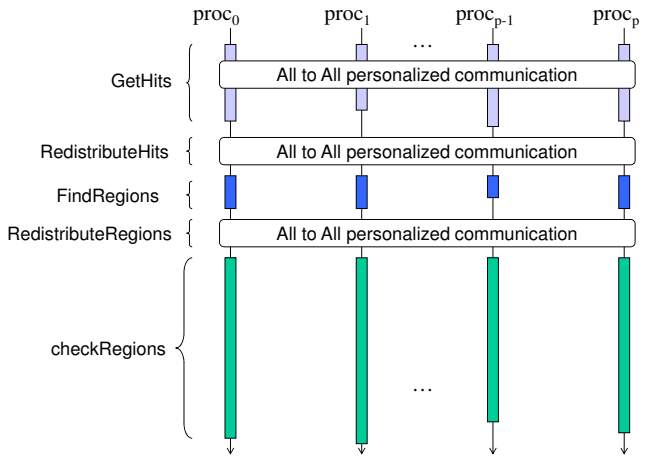

Figure 5. Execution and flow of data through various stages of the distributed MPI implementation

of these sequences, we also maintain a metadata file that stores the name and length of each sequence.

PFANGS starts by having each process read an equal contiguous portion of the *genomeFile*. For example, if the length of the file is L and there are p processes, process 0 will read the first $\frac{L}{p}$ nucleotides, process 1 will read the second $\frac{L}{p}$ nucleotides and so on. Recall that the length of the *lookupTable* is 4^q. In order to create the *index-table* in a distributed manner, process 0 is responsible for the first $\frac{4^q}{p}$ entries of the *lookupTable* and the corresponding part of the *occurrenceTable*, process 1 is responsible for the second $\frac{4^q}{p}$ entries of the *lookupTable* and so on. Each process creates non-overlapping *q-grams* from its chunk of the database and sends each *q-gram* to the process responsible for the corresponding part of the *index-table*. After receiving all the *q-grams*, each process creates its part of the *index-table*. This way we create the *index-table* in a distributed manner. Each process discards its chunk of the database read after the creation of the *index-table*.

Figure 5 shows the execution stages and the communication patterns for the query processing phase of the distributed MPI implementation with p processes. The queries are equally assigned to all processes. The query processing phase is divided into five stages.

- *GetHits:* Each MPI process takes its assigned queries and finds all the overlapping *q-grams*. Each *q-gram* is represented using three numbers: $tileHead$, $tileTail$ and $queryId$; and stored in an array. The array is sorted according to the $tileHead$ value. Hence, the *q-grams* that whose corresponding *index-table* entries are on one process are located in contiguous locations in the array. The process then sends each *q-gram* to the process which has the corresponding part of the *index-table*. It also receives *q-grams* which correspond to its part of the *index-table*. For all these *q-grams*, it hashes the corresponding hits using the *index-table*. Each hit consists of three values: the database location ($dStart$) the query location ($qStart$) and the $queryId$. It also finds all the wildcard hits. Each wildcard hit consists of the query location ($qStart$) and the $queryId$.
- *RedistributeHits:* We redistribute the hits and wildcards across all processes such that after redistribution, all hits and wildcard hits corresponding to one query are on one process and hits are approximately evenly divided across all processes.
- *FindRegions:* Every process processes all the hits for each assigned query to obtain the candidate homologous regions using the algorithm given in Figure 2. A candidate homologous regions consists of three numbers: (1) $dBegin$, (2) $dEnd$ and (3) $queryId$, where $dBegin$ and $dEnd$ are the start and end positions of the candidate region in the database.
- *RedistributeRegions:* The candidate homologous regions are redistributed across all processes such that the number of regions on each process is approximately equal and regions with close $dBegin$ values are on the same process. In order to do this, we perform a global bucket sort on all the regions across all processes based on the $dBegin$ value. Then we divide the sorted list of regions into equal parts and assign one part to each process.
- *CheckRegions:* Each process takes the list of regions (*regionList*) assigned to it. The regions are already sorted according to $dBegin$ values as a result of the global bucket sort. Each process reads the genome database from minimum of $dBegin$ values to maximum of $dEnd$ values of all the assigned regions. Then it checks each candidate region one by one to see if the edit distance is indeed less than *maxEditDist* using the algorithm given in Figure 4. All the homologous regions, which satisfy the criteria, are sent to the root process. The root process concatenates the results from all processes and writes them to the output file. Since the regions are processed in increasing order of dBegin values and regions with close $dBegin$ values are on the same process, the disk IO cost due to random access is minimized.

VII. Experiments and Performance Analysis

In our experiments, the human genome database is used. The queries to be used to search against the database were randomly sampled from the human genome into reads of length 500. The number of queries is set to 10000 per process. In this section, we will use the word "process" to refer to MPI process as well as OpenMP threads, so that the presentation is consistent. The comparison of the speed and the sensitivity and accuracy of FANGS with the other existing tools is given in [17]. The paper shows that

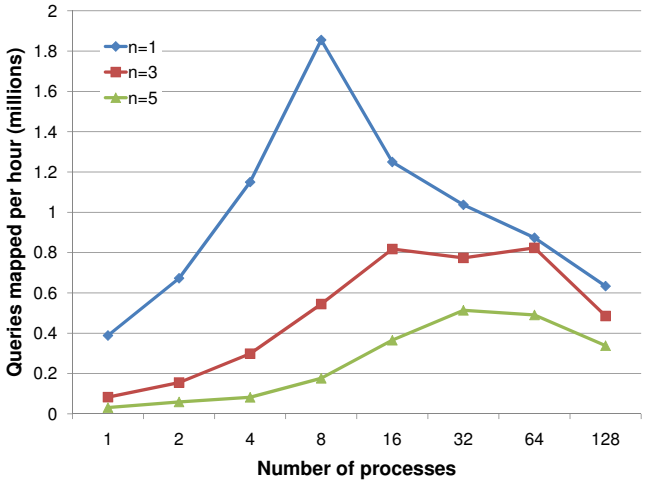

Figure 6. Number of queries mapped per hour to a reference human genome using the shared memory OpenMP implementation for different number of processes and different values of *maxEditDist*, n. Each query is 500 nucleotides long.

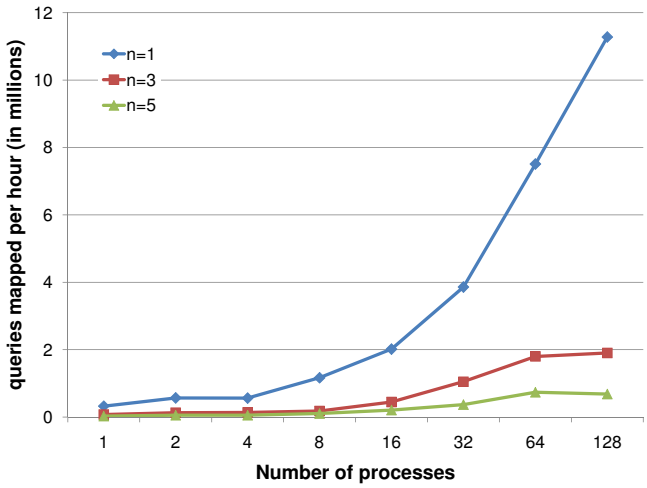

Figure 7. Number of queries mapped per hour to a reference human genome using the MPI-OpenMP hybrid implementation for different number of processes and different values of *maxEditDist*, n. We have used 2 shared memory cores per node. Each query is 500 nucleotides long.

sequential FANGS is upto an order of magnitude faster than the state-of-the-art techniques for 454-Roche reads of length 500 allowing 5 mismatches or insertion/deletions. To the best of our knowledge, there is no other published parallel implementation for mapping 454-Roche sequencing data. Hence, in this paper, we compare the parallel implementation with the sequential implementation of FANGS. All the parallel implemantations retain the sensitivity and accuracy of FANGS (data not shown here).

The experiments of using the shared-memory approach were performed on the NCSA SGI Altix SMP machine (Cobalt). Cobalt has two SMP nodes with 512 1.6 GHz Intel Itanium 2 processors each. The machine has 4GB of memory

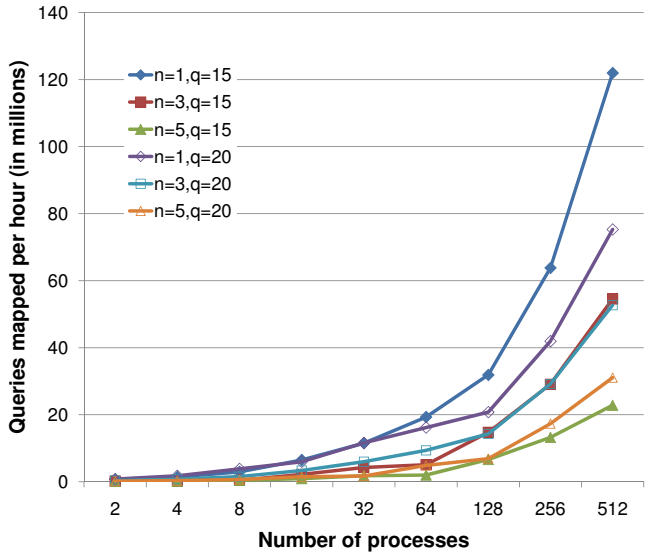

Figure 8. Number of queries mapped per hour to a reference human genome using the data and task parallel distributed memory MPI implementation for different number of processes and different values of *maxEditDist*, n and tileSize, q. Each query is 500 nucleotides long.

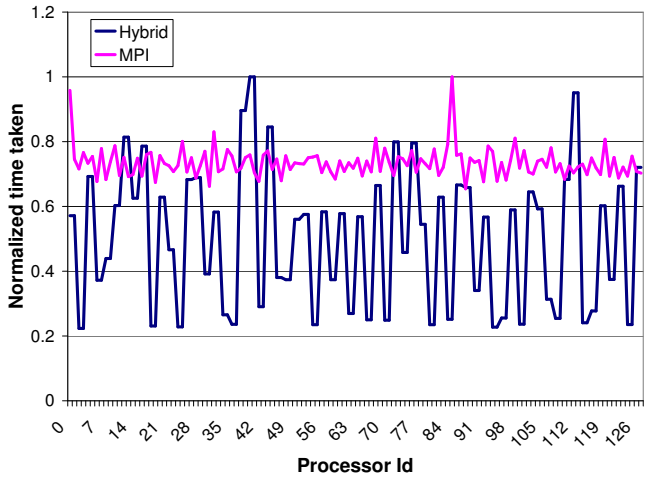

Figure 9. Comparison of load on each process for hybrid and MPI implementations. In order to obtain this plot, we normalized the time taken by each process by the maximum time taken by a process. The variances of load for hybrid and MPI implementations are 0.046 and 0.0021 respectively.

per processor. The SMP machine is running SGI ProPack 5 and Intel 10.1 C compiler. Figure 6 shows the number of queries mapped per hour by using the shared memory parallel implementation. Since all the processes are accessing the shared database and *index-table*, memory IO becomes a major bottleneck when the number of processes increases. Hence the number of queries mapped per hour increases first and then starts decreasing when such bottleneck becomes significant. It is clear that the shared memory approach does not scale. It improves the performance up to a certain

number of threads beyond which the performance starts to come down. Such performance saturation and degradation are commonly seen on the SMP parallel machines, due to the contention on the system bus as well as the system overhead of cache coherence control.

The performance results of the MPI-OpenMP hybrid implementation were collected from the NCSA IA-64 Teragrid cluster (Mercury). Mercury consists of 887 IBM cluster nodes: 256 nodes have dual 1.3 GHz Intel Itanium 2 processors and 631 nodes have dual 1.5 GHz Intel Itanium 2 processors. All the 1.5 GHz nodes and half of the 1.3 GHz nodes have 4GB of memory per node. The other half of the 1.3 GHz nodes are large-memory processors with 12GB of memory per node, making them ideal for running applications which require large memory. The cluster runs SuSE Linux and uses Myricom's Myrinet cluster interconnect network. We ran our experiments on the half of the 1.3 GHz nodes which have 12GB of memory per node. Figure 7 presents the performance results. The hybrid approach scales much better than the pure shared-memory approach, as each node has its own copy of the database and the *index-table*. Only a limited number of threads share each copy of the hash table using OpenMP. However, even though there are a small number of OpenMP threads on each node, they still have to contest for memory access thereby resulting in a sub-optimal speedup. Recall that our algorithm needs about 4.5GB memory to execute. Such large memory requirement with random data accesses can cause significant cache misses.

The distributed memory implementation was also evaluated on Mercury. Figure 8 and Table II show the results. It can be clearly seen that the distributed memory implementation scales very well. As the database and *index-table* (See Table I for performance results of creation of the *index-table*)are distributed across all processes, the memory requirement on one process is smaller, thereby reducing the number of cache-misses and page-faults. Another important thing to note is that, for the shared memory and hybrid implementations, we statically divided the queries equally across all processes assuming that the amount of load is equal for equal number of queries. Since our sequence mapping algorithm is heuristic based, the actual run times and result sizes for queries are highly irregular and difficult to predict. For the 128 process case, figure 9 displays the load on each process for the hybrid and MPI implementations. It is clear that there is significant load-imbalance for the hybrid implementation, while the load for the MPI implementation is much better balanced. For the MPI implementation, two of the processes (0 and 85) always take significantly more time in the *CreateIndex* and *GetHits* stage. Our initial investigations reveal that the imbalance in load is due to the irregular nature of the genomic databases.

Figure 10 shows the breakdown of time spent on each stage of the query processing phase for various values of *maxEditDist* and number of processes used. As the value of *maxEditDist* increases, more candidate homologous regions are generated by *FindRegions* algorithm since the value of T gets smaller. Moreover, *CheckRegions* stage has higher computational complexity as compared to other stages. Hence for larger values of *maxEditDist*, *CheckRegions* stage consumes more than 85% of the overall execution time. Note that each region can be examined independently of all other regions. We dynamically balance the load across processes by redistributing the candidate homologous regions evenly across processes to achieve better process efficiency. Also note that the percentage of time spent on the *RedistributeHits* stage increases as the communication time increases with the increase in the number of processes. This stage may become a bottleneck and hinder scalability as the number of processes increase. To avoid this, for larger number of processes, we divide them into disjoint subsets of 128 processes each. The queries are equally divided among these subsets. Each of these subsets work independently by creating their own copy of the index. As a result of this, the percentage of time spent on the *RedistributeHits* stage does not increase as the number of processes increase beyond 128. Notice from Figure 8 that we can process up to 31061118 queries per hour for $n = 5$ using 512 processors. Since each query is of length 500, this means we can map 454-Roche reads with a total of $31061118 * 500 = 15.53$ Billion nucleotides per hour against a reference human genome. Hence, with 512 processors, we are able to map 454/Roche reads of 5.17x coverage of a human genome to a reference human genome per hour allowing 5 mismatches or Indels at full sensitivity. In other words, we can map 5.17 human genomes per hour.

	Time taken (seconds)					
	$q = 15$			$q = 20$		
# proc	n=1	n=3	n=5	n=1	n=3	n=5
2	300.5	299.1	299.0	225.9	231.1	225.0
4	159.2	160.1	161.7	117.6	117.0	123.8
8	87.0	87.1	80.1	58.5	63.5	64.6
16	46.8	45.9	46.9	39.3	34.8	34.5
32	31.4	32.5	31.4	27.3	21.9	25.9
64	27.6	28.0	27.5	15.2	15.2	17.1
128	21.6	21.9	21.9	12.9	14.0	14.1
256	20.7	20.7	21.6	11.8	12.2	14.5
512	21.4	21.3	21.8	13.6	12.6	14.1

Table I
ABSOLUTE TIME TAKEN FOR THE CREATION OF INDEX TABLE FOR DIFFERENT VALUES OF n AND q, FOR DIFFERENT NUMBER OF PROCESSORS.

VIII. CONCLUSION

Advances in sequencing techniques necessitate the development of high performance, scalable algorithms to extract biologically relevant information from these datasets. In this paper, we investigate different parallel implementations of a fast sequence alignment tool FANGS. Firstly we develop

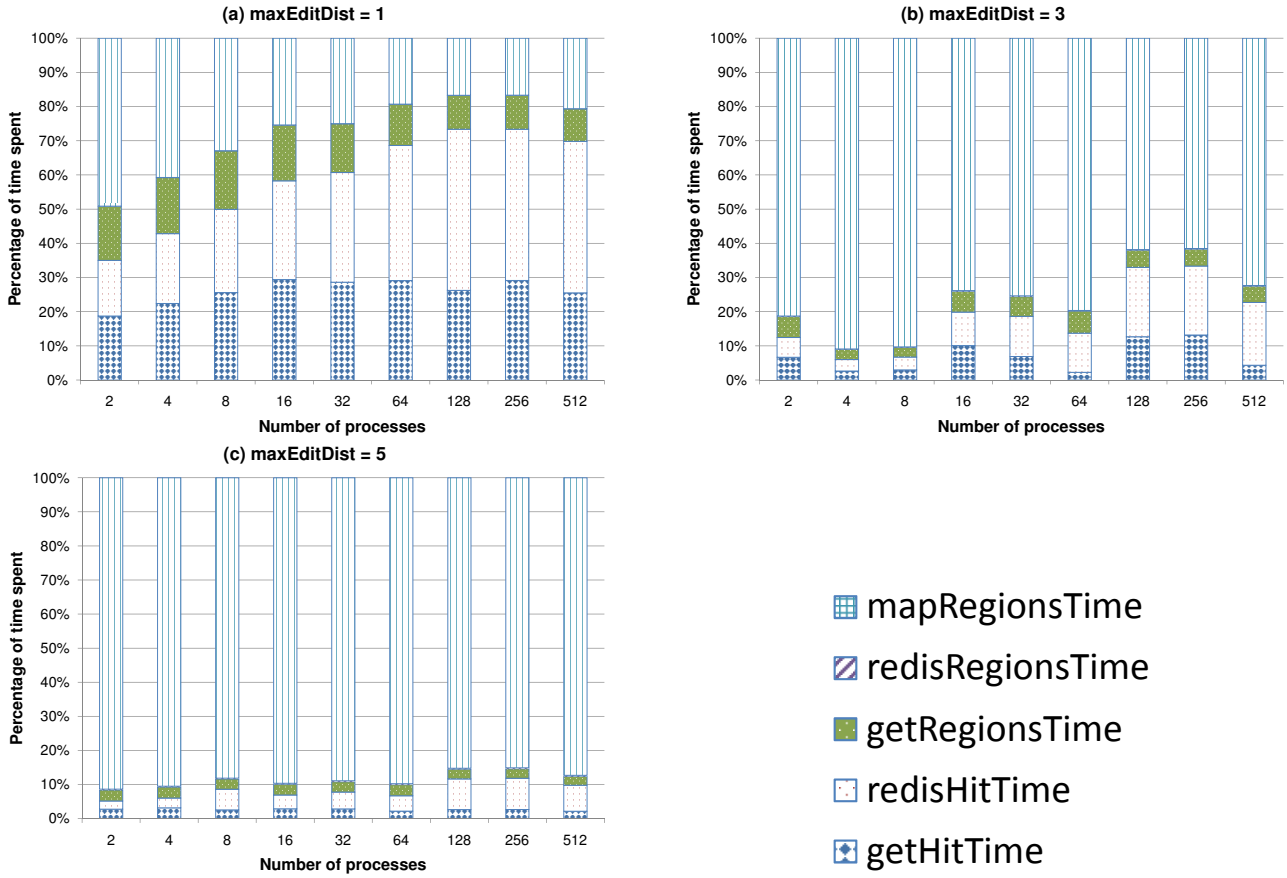

Figure 10. Percentage of time spent on each stage of the distributed memory MPI implemenation for different number of processes and different values of maxEditDist, n

	Speedup with respect to two processes					
	$q = 15$			$q = 20$		
# proc	n=1	n=3	n=5	n=1	n=3	n=5
2	1.0	1.0	1.0	1.0	1.0	1.0
4	2.1	1.0	2.0	2.2	2.2	2.0
8	4.3	1.7	4.4	4.8	4.5	3.9
16	9.5	9.1	9.0	7.5	10.1	9.1
32	16.8	17.5	17.5	14.5	17.9	10.2
64	28.1	21.0	19.6	20.3	28.1	29.4
128	46.4	60.8	65.5	26.0	42.8	41.9
256	92.9	119.6	130.7	52.5	87.5	105.1
512	177.6	224.2	225.4	94.3	158.1	188.5

Table II
SPEEDUP FOR PROCESSING STAGE OF PFANGS WITH RESPECT TO TIME TAKEN BY TWO PROCESSES FOR DIFFERENT VALUES OF n AND q, FOR DIFFERENT NUMBER OF PROCESSES.

query segmentation based OpenMP and MPI-OpenMP hybrid implementations and discuss their limitations. We then develop a highly optimized data- and task-distributed MPI implementation with intelligent load-balancing techniques that avoid problems of memory bandwidth and cache misses. Our experimental evaluation shows that this technique results in excellent load-balance and process efficiency and hence yield close to linear speedups.

With the advent of new technologies, we will need even faster sequence mapping tools to stay at par with the increasing sequencing speed. With the development of better parallel algorithms, we can setup huge processing centers which contain a large number of sequencers producing reads and huge clusters working in tandem to rapidly process them to extract a variety of information. The Next Generation Sequencers along with high-speed sequence processing systems will enable us to realize the dream of personal genomics. This can help us in using a patient's DNA in diagnosing a disease or even knowing in advance whether a person's DNA encodes a risk of a certain disease.

ACKNOWLEDGMENT

This work was supported in part by NSF CCF-0621443, NSF SDCI OCI-0724599, NSF CNS-0551639 and IIS-0536994, DOE SCIDAC-2: Scientific Data Management Center for Enabling Technologies (CET) grant DE-FC02-07ER25808, DOE SCiDAC award number DE-FC02-01ER25485.

REFERENCES

[1] SHRiMP - SHort Read Mapping Package http://compbio.cs.toronto.edu/shrimp/.

[2] Mosaik: http://bioinformatics.bc.edu/marthlab/Mosaik.

[3] BWA: http://maq.sourceforge.net/bwa-man.shtml.

[4] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *J Mol Biol*, 215(3):403–410, October 1990.

[5] R. Bjornson, A. Sherman, S. Weston, N. Willard, and J. Wing. Turboblast(r): A parallel implementation of blast built on the turbohub. *Parallel and Distributed Processing Symposium, International*, 2:0183, 2002.

[6] D. Bozdag, C. C. Barbacioru, and U. V. Catalyurek. Parallel short sequence mapping for high throughput genome sequencing. *Parallel and Distributed Processing Symposium, International*, 0:1–10, 2009.

[7] R. C. Braun, K. T. Pedretti, T. L. Casavant, T. E. Scheetz, C. L. Birkett, and C. A. Roberts. Parallelization of local blast service on workstation clusters. *Future Gener. Comput. Syst.*, 17(6):745–754, 2001.

[8] E. Chi, E. Shoop, J. Carlis, E. Retzel, and J. Riedl. Efficiency of shared-memory multiprocessors for a genetic sequence similarity search algorithm. *Technical Report TR97–005*, 1997.

[9] A. E. Darling, L. Carey, and W. C. Feng. The design, implementation, and evaluation of mpiblast.

[10] H. Jiang and W. H. Wong. Seqmap : mapping massive amount of oligonucleotides to the genome. *Bioinformatics*, pages btn429+, August 2008.

[11] W. J. Kent. Blat–the blast-like alignment tool. *Genome Res*, 12(4):656–664, April 2002.

[12] B. Langmead, C. Trapnell, M. Pop, and S. Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biology*, 10(3), 2009.

[13] H. Li, J. Ruan, and R. Durbin. Mapping short dna sequencing reads and calling variants using mapping quality scores. *Genome research*, August 2008.

[14] R. Li, Y. Li, K. Kristiansen, and J. Wang. Soap: short oligonucleotide alignment program. *Bioinformatics*, January 2008.

[15] H. Lin, X. Ma, P. Chandramohan, A. Geist, and N. Samatova. Efficient data access for parallel blast. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, page 72b, 2005.

[16] D. Mathog. Parallel blast on split databases. *Bioinformatics*, 19:1865–1866, 2003.

[17] S. Misra, R. Narayanan, S. Lin, and A. Choudhary. Fangs: High speed sequence mapping for next generation sequencers. In *Proceedings of ACM Symposium of Applied Computing (ACM SAC)*, Sierre, Switzerland, 2009.

[18] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, March 1970.

[19] Z. Ning, A. J. Cox, and J. C. Mullikin. Ssaha: A fast search method for large dna databases. *Genome Res.*, 11(10):1725–1729, 2001.

[20] K. L. Patrick. 454 life sciences: Illuminating the future of genome sequencing and personalized medicine. *Yale J Biol Med.*, 80(4):191–4, Dec 2007.

[21] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proc Natl Acad Sci U S A*, 85(8):2444–2448, April 1988.

[22] P. Pevzner and M. Waterman. Multiple filtration and approximate pattern matching. *Algorithmica*, 13:135–154, 1995.

[23] K. R. Rasmussen, J. Stoye, and E. W. Myers. Efficient q-gram filters for finding all epsilon-matches over a given length. *Journal of Computational Biology*, 13(2):296–308, 2006.

[24] C. Shaffer. Next-generation sequencing outpaces expectations. *Nature Biotechnology*, 25(2):149, February 2007.

[25] A. D. Smith, Z. Xuan, and M. Q. Zhang. Using quality scores and longer reads improves accuracy of solexa read mapping. *BMC Bioinformatics*, 9:128+, February 2008.

[26] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[27] D. A. Wheeler, M. Srinivasan, M. Egholm, Y. Shen, L. Chen, A. Mcguire, W. He, Y.-J. Chen, V. Makhijani, G. T. Roth, X. Gomes, K. Tartaro, F. Niazi, C. L. Turcotte, G. P. Irzyk, J. R. Lupski, C. Chinault, X.-Z. Song, Y. Liu, Y. Yuan, L. Nazareth, X. Qin, D. M. Muzny, M. Margulies, G. M. Weinstock, R. A. Gibbs, and J. M. Rothberg. The complete genome of an individual by massively parallel dna sequencing. *Nature*, 452(7189):872–876, April 2008.

[28] T. D. Wu and C. K. Watanabe. Gmap: a genomic mapping and alignment program for mrna and est sequences. *Bioinformatics*, 21(9):1859–1875, May 2005.

[29] Z. Zhang, S. Schwartz, L. Wagner, and W. Miller. A greedy algorithm for aligning dna sequences. *J. Comput. Biol*, 7:203–214, 2000.