Eliminating Irregularities of Protein Sequence Search on Multicore Architectures

Jing Zhang[1], Sanchit Misra[2], Hao Wang[1], Wu-chun Feng[1]
[1]Dept. of Computer Science, Virginia Tech, {zjing14, hwang121, wfeng}@vt.edu
[2]Parallel Computing Lab, Intel Corporation, sanchit.misra@intel.com

***Abstract*—Finding regions of local similarity between biological sequences is a fundamental task in computational biology. BLAST is the most widely-used tool for this purpose, but it suffers from irregularities due to its heuristic nature. To achieve fast search, recent approaches construct the index from the database instead of the input query. However, database indexing introduces more challenges in the design of index structure and algorithm, especially for data access through the memory hierarchy on modern multicore processors.**

In this paper, based on existing heuristic algorithms, we design and develop a database indexed BLAST with the identical sensitivity as query indexed BLAST (i.e., NCBI-BLAST). Then, we identify that existing heuristic algorithms of BLAST can result in serious irregularities in database indexed search. To eliminate irregularities in BLAST algorithm, we propose muBLASTP, that uses multiple optimizations to improve data locality and parallel efficiency for multicore architectures and multi-node systems. Experiments on a single node demonstrate up to a 5.1-fold speedup over the multi-threaded NCBI BLAST. For the inter-node parallelism, we achieve nearly linear scaling on up to 128 nodes and gain up to 8.9-fold speedup over mpiBLAST.

***Keywords*-BLAST; Pairwise sequence alignment; Database index; Multicore; MPI**

I. Introduction

The Basic Local Alignment Search Tool (BLAST) [1] performs the fundamental task in life sciences to identify the most similar sequences from the database to a given query sequence. The similarities identified by BLAST can be used to infer functional and structural relationships between the corresponding biological entities. Although optimizing BLAST is a rich area of research using multi-core CPUs, many-core GPUs, and clusters and clouds [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], BLAST is still a major bottleneck in biological research. In fact, in a recent human microbiome study that consumed 180,000 core hours, BLAST consumed nearly half the time [15]. It still requires urgent attention in higher level applications.

BLAST adopts a heuristic method to identify the similarity between the query sequence and subject sequence from the database. Initially, the query sequence is decomposed into short words with fixed length; and the words are converted into the query index, i.e., a lookup table [16] or a deterministic finite automaton (DFA) [17], to store the positions of words in the query sequence. BLAST reads the words from the subject sequence and identifies high scoring short matches, i.e., hits, from the query index. Based on two or more hits near enough to each other, BLAST forms the local alignment without the insertions and deletions, i.e., gaps, (called two-hit ungapped extensions), and then generates the further extension based on the local alignments but allows the gaps. Although such a heuristic can efficiently eliminate unnecessary search space, it makes the execution of program unpredictable and the memory access irregular, leading to the limited scope of SIMD parallelism and the increase of trips to memory.

With the advent of next-generation sequencing (NGS), the exponential growth of sequence databases is arguably outstripping the ability to analyze the data. In order to deal with huge databases, a range of recent approaches of BLAST build the index based on the subject sequences of database instead of the input query [18], [19], [20], [21], [22]. Although these alternatives building the database index in advance and reusing it for multiple queries can improve the overall performance, there are more challenges in the parallel design on multi-core processors. In fact, most of the tools have to use longer, non-overlapping, or non-neighboring words to reduce the size of database index, and consequently reduce the number of hits and extensions, to fit in the memory. However, as reported by [23], [24], [25], these methods compromise the sensitivity and accuracy compared to the query indexed methods.

In this paper, following the existing heuristic algorithm, we first implement a database indexed BLAST that includes the overlapping and neighboring words, and provides exactly the same accuracy as query indexed BLAST, i.e., NCBI-BLAST. Then, we identify that directly using the existing heuristic algorithms on the database indexed BLAST will suffer further from irregularities: when it aligns a query to multiple subject sequences at the same time, the ungapped extension, which is the most time-consuming stage, will access the memory randomly across different subject sequences. Even worse is that the penalty from random memory access cannot be offset by the cache hierarchy even on the latest multi-core processors. To eliminate irregularities in BLAST algorithms, we propose muBLASTP, a multi-threaded and multi-node parallelism of BLAST algorithms for protein sequence search. It includes four major optimization techniques: (1) **decoupling** hit search and ungapped extension based on their different memory access patterns, (2) **sorting** hits to remove the irregular memory access and improve data locality in ungapped extension, (3) **pre-**

1530-2075/17 $31.00 © 2017 IEEE
DOI 10.1109/IPDPS.2017.120

filtering hits not near enough to reduce the overhead of hit sorting, (4) **refactoring** the data partitioning method to improve the load balance and remove the overhead of contention and synchronization for both intra-node and inter-node parallelisms. We carry out the experiments for single node on a dual socket Intel® Xeon® Haswell processor and multi-node on the Stampede supercomputer containing a dual socket Intel® Xeon® Sandy Bridge processor per node. Our experimental results show significant performance improvements. Compared to the query index based NCBI-BLAST and the database index based NCBI-BLAST, our method can deliver up to 5.1-fold and 3.9-fold speedups, respectively, on a single node. Using 128 nodes, our method can deliver nearly linear scaling and gain up to 8.9-fold speedup over mpiBLAST. To guarantee that every change on the algorithm does not affect outputs, we verify the outputs of every stage in muBLASTP are the same to NCBI-BLAST.

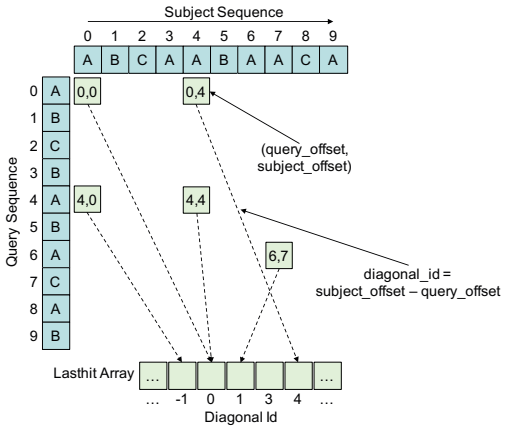
II. Background and Motivation

A. Preliminaries

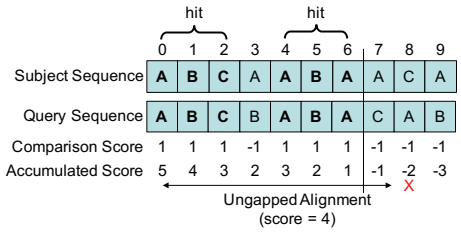
BLAST is a family of programs to approximate the results of the Smith-Waterman algorithm [26], [27], an optimal local-alignment algorithm. Instead of comparing the entire sequence, BLAST uses a heuristic method to reduce the search space. With only a slight loss in accuracy, BLAST executes significantly faster than the Smith-Waterman. In this paper, we focus on BLAST for protein sequence search, called BLASTP, which is more complicated than the other variants, e.g., BLASTN for nucleotide sequence search, because there are 24 possible characters (20 characters for amino acids, 4 additional ones for special protein states) in protein sequences while only 4 characters (A/C/T/G) in nucleotide sequences. The BLASTP algorithm consists of the four stages as below:

Hit detection finds high-scoring short matches (i.e., hits) between the query sequence and the subject sequence from database. The index, which is built on the query or subject sequences, records the positions of words with fixed length W. Typically, W is 3 in BLASTP and the words can be overlapped. For example, in Figure 1(a), *ABC* at the position 0 and *BCA* at the position 1 are overlapping words in the subject sequence. In order to improve the accuracy, the neighboring words, which contains the word itself and the similar words to the word, are also considered to be the hits. For example, the neighboring words *ABC* and *ABA* are treated as the hit to each other in Figure 1(a). For the query indexed search, the hit detection scans the subject sequence from left to right, and searches each word in the query index to find the hits. For the database indexed search, the hit detection scans the query sequence from top to bottom.

Two-hit ungapped extension finds the pairs of hits close together, and extends hit pairs to basic alignments without gaps. The ungapped extension algorithm uses an array, called last hit array *lasthit*, to update the position of last found hit for each diagonal. When a hit is found, the algorithm computes its distance to the last found hit of the diagonal. If the distance is less than a predefined threshold, the ungapped extension is triggered in both backward and forward directions. In Figure 1(a), when the hit (4,4) is found in the diagonal 0, the algorithm checks the distance to the last hit (0,0) in the same diagonal, and triggers the ungapped extension, which ends at the position (7,7). Then, the ending position will be written back to the position 0 of last hit array for diagonal 0. Figure 1(b) shows the details of the ungapped extension. The extension starts at the end of the second hit (6,6), and extends to both left and right. For the left extension starting at the position 6, where letters in both query and subject sequence are "A", the accumulated score increases by 1. The right extension starts at position 7, where the letters are mismatched, the accumulated score decreases by 1. The ungapped extension stops when the accumulated score drops a threshold -2 below the highest accumulated score during the extension.



(a) Hit detection



(b) Ungapped extension

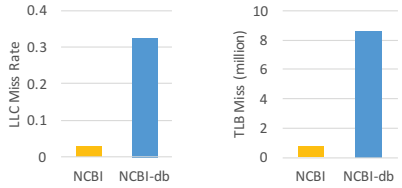
Figure 1: An example of BLAST algorithm for the most time-consuming stages — hit detection and ungapped extension.

The third stage ***Gapped extension*** performs a gapped alignment with dynamic programming on the high-scoring ungapped regions to determine if they can be part of a larger, higher-scoring alignment. The fourth stage ***Traceback*** re-aligns the top-scoring alignments from the gapped extension using a traceback algorithm, and produces the top scores.

The ranked results will be then returned back to the user. Because the third and fourth stages are not considered as the performance bottleneck, we focus on optimizing the hit detection and ungapped extension in this paper, and apply the optimizations proposed in the previous research [4] to improve their performance on multi-core processors.

B. Motivation

The existing BLAST algorithm executes the first three stages interactively: once a hit is detected, the algorithm immediately triggers the ungapped extension, if the distance to the last hit in the same diagonal is smaller than the threshold, and then triggers the gapped extension. For the query indexed BLAST, since the subject sequences are aligned one by one to the query, only one last hit array is needed for the query sequence. Moreover, because most of the protein sequences are short enough, e.g., no more than 2K characters, the query indexed BLAST can achieve good performance thanks to the cache systems on modern multi-core processors, even though the memory access pattern on those data structures are totally random.



(a) LLC miss rate (b) TLB miss

(c) Stalled cycle(%) (d) Execution time

Figure 2: Profiling numbers and execution time of the query indexed NCBI-BLAST (NCBI) and the database indexed NCBI-BLAST (NCBI-db) when searching a query of length 512 on *env_nr* database.

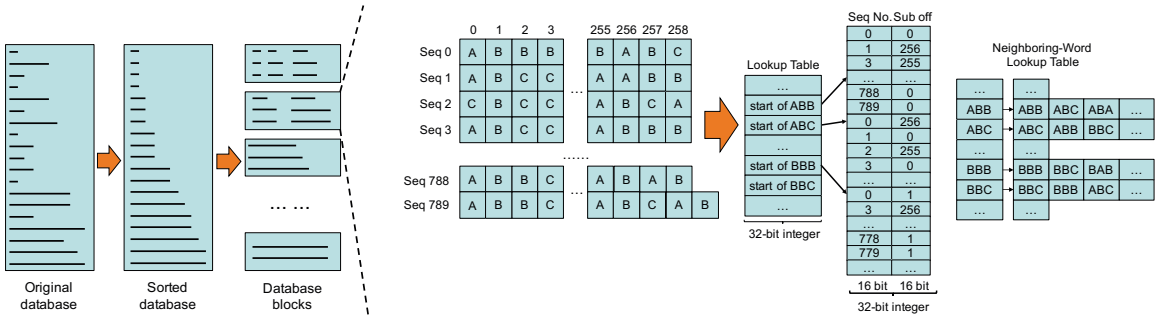
However, irregular memory access in the database indexed search can lead to a severe locality issue. Because each word in a database index is including positions from tons of different subject sequences, the algorithm has to keep multiple last hit arrays, one for each subject sequence. When the algorithm scans the query sequence successively, a new hit may occur in any last hit array, and the ungapped extension may be triggered for any subject sequence. As a consequence, the execution path of the program will jump back and forth across different subject sequences, leading to the cached data of last hit arrays and subject sequences flushed out before reuse. Figure 2(a) and 2(b) compares the LLC (Last-Level Cache) and TLB (Translation Lookaside Buffer) miss rate, respectively, between NCBI-BLAST with the query index (NCBI) and NCBI-BLAST with the database index (NCBI-db), when searching a real protein sequence having 512 letters on *env_nr* database. Note that NCBI-db (described in Section III) uses the database index with overlapping and neighboring words to provide the same results as the default NCBI-BLAST. It is clear the database indexed method has much higher LLC and TLB miss rate, which results in much higher stalled cycle percentage (Figure 2(c)). As a result, Figure 2(d) shows the overall performance of database indexed NCBI-BLAST is even worse than that using the query index, which is anti the motivation of database index design.

III. Building Database Index

In this section, we introduce how to build the database index. Different with previous studies, our database index includes the overlapping words and neighboring words, so that the database indexed BLAST can keep the same accuracy as query indexed BLAST, e.g., NCBI-BLAST. In the database index, we put the words ($W = 3$) as the key, and the subject sequence ID and the position (offset) in the subject sequence as the value. Because the size of database for protein sequences is increasing substantially and exceeding the capacity of main memory, we use the index blocking technique to partition the database index to multiple blocks.

Fig. 3(a) illustrates the details of index blocking. We sort the subject sequences of database by their lengths, divide the subject sequences into multiple blocks, each of which has similar number of characters, and build an index for each block. To avoid cutting a sequence in the middle, if a sequence exceeds the block boundary, we put it in the next block. With the index blocking, the BLAST algorithms can go through the blocks one by one, and then merge the top-ranking results of each block after the four stages (presented in Section II-A). The index blocking can make the database index of huge databases for protein sequences fit into the main memory, especially when we enable the overlapping words. Furthermore, by configuring the size of index block properly, we can fully utilize the cache hierarchy to achieve the best performance. We will evaluate the variable performances with different block sizes in Section V. To compress the size, we further optimize the index that records the local offset of sequences in each block instead of the absolute sequence IDs to save several bits.

Unlike BLAST for nucleotide sequences (i.e., BLASTN), which only cares about exact matches, BLASTP algorithms need to search the neighboring words. When building the query index, most BLASTP tools, e.g., NCBI-BLAST, also store the positions of neighboring words for each word. This method can get all positions of the original word and its neighboring words through contiguous memory accesses but result in heavy redundant positions. The redundant



(a) Index blocking (b) Index structure

Figure 3: An example of building a database index.

positions will extremely increase the size of database index. To overcome this problem, we build an additional lookup table that contains the neighboring words of each words. As shown in Figure 3(b), the neighboring word lookup table puts each word as the key (e.g., *ABB*), and put its neighboring words as the values, (e.g., *ABB*, *ABC*, etc.). In hit detection, we first get all neighboring words of a word, and then loads corresponding positions for each neighbor. Although this two-level structure of database index requires slightly extra memory accesses, it can dramatically reduce the total size of the index.

IV. Removing Irregularities of BLASTP

A. Decoupling First Three Stages

As discussed in Section II-B, with the database index, the BLAST algorithms have to operate on multiple last hit arrays simultaneously, because one word can induce multiple hits at different subject sequences. The interleaving execution of hit detection, ungapped extension, and gapped extension will lead to random memory accesses across different last hit arrays and subject sequences. In order to avoid the data swapped in and out of the cache without being fully reused, we decouple these three stages. That means after loading one index block, the hit detection will find all hits, and store hits in a temporal buffer. Because the hits for a single subject sequence may be distributed randomly in this buffer, we add an additional stage, i.e., hit reordering, before the ungapped extension and the following gapped extension.

A new data structure is introduced to record the hits for fast hit reordering. A hit should contain the subject sequence ID, diagonal ID, subject position (offset) and query position (offset). For the sequence ID and diagonal ID, we pack them into one 32-bit integer as the key, in which the sequence ID uses the higher bits and the diagonal ID uses the lower bits. With this packed key, sorting hits by the key once, hits are sorted in the order for both sequence IDs and diagonal IDs. For the subject offset and query offset, since a subject offset or query offset can be calculated with each other in a given diagonal as $diagonal\_id - query\_offset$ or $diagonal\_id - subject\_offset$, we only keep one of these two offsets, e.g., the query offset as shown in Figure 4, and calculate the other in the runtime of ungapped extension. We realize that today's protein database may contain very long sequences ($\sim 40k$ characters). We don't build the index for such extreme cases. Instead, we use a method proposed recently in [14] to divide the extremely long sequence into multiple short sequences with the overlapped boundaries and use an assembly stage to extend the ungapped extension and gapped extension after finishing the extension inside each short sequence.

B. Hit Reordering with Radix Sort

As shown in Figure 4, the hit detection algorithm will put hits for different subject sequences in successive memory locations in the temporal hit buffer. For the word *ABC*, the hit detection will put the hits (0,0) and (0,4) for the subject sequence 0 in the hit buffer, and then put the hits (0,0), (0,4), and (0,6) for the subject sequence 1 into the following memory locations of the hit buffer. Because the ungapped extension can only operate on hits in the same diagonal of a given subject sequence, we have to reorder the out of order hits in order.

The radix sort and merge sort are two candidates for the hit reordering. The radix sort has the $O(n)$ computational complexity, but it requires several passes over each element, leading to higher bandwidth utilization. The merge sort has the higher computational complexity, i.e., $O(n \log n)$, but it can be highly efficient due to the ease of vectorization and bandwidth friendly. Recent studies [28], [29], [30], [31], [32] compared the performance of radix sort and merge sort in modern multi-core processors, and revealed the best scenarios for each method. In our applications, we implement both methods with the optimizations proposed in previous research, and find the radix sort is better for hit reordering due to the following reasons. First, thanks to the index blocking technique, each block has hundreds of kilobytes to several megabytes of hits, which can fit into the LLC of multi-core processors in our evaluation. Therefore, the radix sort does not have severe memory bandwidth issue. Second, because we sort the subject sequences when building the database index, each block has the similar length of sequences and so does the length of diagonals. As a result, the radix sort operating on the sequence IDs

and diagonal IDs can finish sorting all hits by using similar passes. Thus, the fixed length of keys is friendly to the radix sort. Third, in the hit detection, the query sequence is scanned from the beginning to the end, and the hits are already in the order of query offsets. Because we need to keep such an order in the key-value sort, the radix sort is a better choice considering the merge sort may lose a little bit performance to achieve the stable sort. There are two ways to implemented the radix sort, one is beginning at the least significant digit, called LSD radix sort; and the other is beginning at the most significant digit, called MSD radix sort. Although MSD radix sort has less computational complexity because it may not need to examine all keys, MSD radix sort is too slow for small datasets, e.g., hundred kilobytes in our case. Therefore, we choose LSD radix sort to reorder the hits after the hit detection stage.

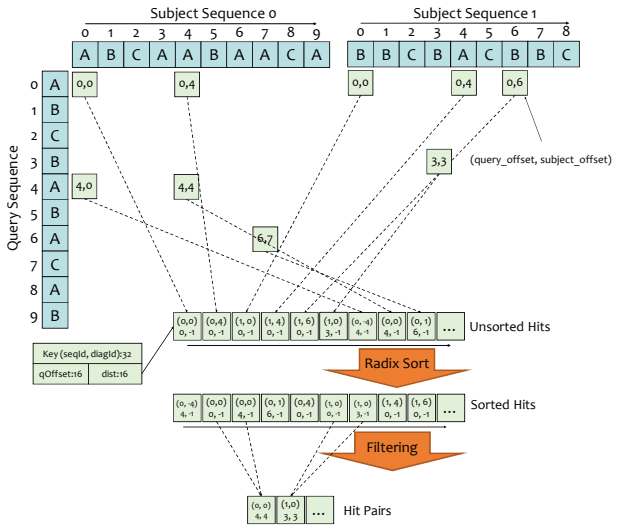


Figure 4: Hit-pair search with hit reordering

Algorithm 1 illustrates BLAST algorithms on the database index. To achieve better data locality, the algorithm loads the index blocks one by one (line 1), and go through all input queries for the index block in the inner loop (line 2). For each query in the inner loop, the hit detection function *hitDetect()* scans the current query, and find hits for all subject sequences in the index block (line 3). All hits are sorted with the key, including the sequence ID and diagonal ID, by LSD radix sort (line 4). After the hits are sorted, they are passed to the filtering stage (line 7) to pick up the hit pairs near enough along the same diagonal (line 9), and store into the internal buffer *HitPairs*. In the ungapped extension, the *for* loop starting from line 15, the hit pairs are extended one by one in the order of subject sequence IDs and diagonal IDs. Thus, this method can reuse the subject sequence during the ungapped extension, while the previous methods cannot, because they issue the ungapped extension immediately within the hit detection and have to jump from one subject sequence to another. Before doing the ungapped extension, the algorithm will also check if the current hit pair is covered by the extension of previous hit pair (line 16). If it is, the algorithm will skip this hit pair.

Algorithm 1: Database Indexed BLASTP Algorithms with Hit Reordering

input : DI: database index, Q: query sequences
output: U: high-scoring ungapped alignments

```
1  foreach database index block dIdxBlk_i in DI do
2      foreach sequence q_i in Q do
3          hits ← hitDetect(dIdxBlk_i, q_i);
4          sortedHits ← radixSort(hits);
5          reachedPos ← −1;
6          reachedKey ← −1;
7          foreach hit_i in sortedHits do
8              distance ← hit_i.qOffset − reachedPos;
9              if reachedKey == hit_i.key and
                distance < threshold then
10                 hit_i.dist = distance;
11                 HitPairs ← HitPairs + hit_i;
12             reachedPos ← hit_i.qOffset;
13             reachedKey ← hit_i.key;
14         extReached ← −1 reachedKey ← −1;
15         foreach hit_i in HitPairs do
16             if reachedKey == hit_i.key and
                extReached > hit_i.qOffset then
17                 skip this hit;
18             else
19                 ext ← ungappedExt(hit_i, lastHit, S, q_i);
20                 if ext.score > thresholdT then
21                     U ← U + ext;
22                     extReached ← ext.end;
23                 else
24                     extReached ← hit_i.qOffset;
25             reachedKey ← hit_i.key;
```

C. Hit Pre-filtering

Although we have applied the highly efficient radix sort in the hit reordering, the overhead to sort millions of hits per block are not negligible. We introduce a pre-filtering stage before the hit reordering to filter out hits that cannot trigger the ungapped extension. We use the similar idea of the last hit array: an array is created for a subject sequence to record the current hit in each diagonal; instead of triggering ungapped extension immediately when a hit pair is detected, the hit pair is put into the hit buffer. Because we only use these last hit arrays in the hit detection in which we don't access any subject sequence, we do not have the cache swapping issue in the last hit array method. Figure 5 illustrates the optimized BLAST algorithms with the hit pre-filtering.

Figure 6 illustrates the number of hits that will be sorted in the hit reordering stage with and without the hit pre-filtering. There are only less than 5 percent of hits left after pre-filtering. As a result, the overhead in radix sort can be reduced dramatically.

Algorithm 2 shows the optimized BLAST algorithms with pre-filtering. In the inner loop, the two-dimensional array

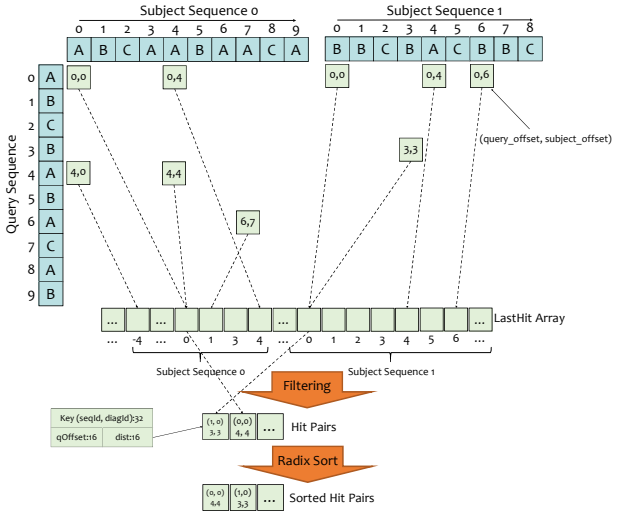


Figure 5: Hit reordering with pre-filtering

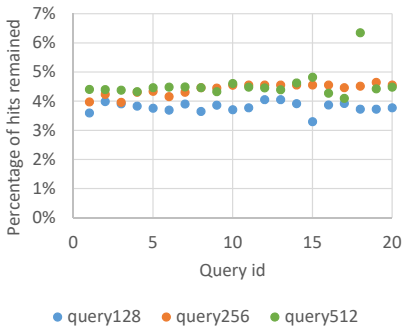


Figure 6: Percentage of hits remained after pre-filtering with different query length — 128, 256 and 512 from *uniprot_sprot* database

lastHitArr is used to record the last hits in diagonals of subject sequences. When a hit is detected (line 4), the algorithm calculates its diagonal ID and sequence ID (line 5), and accesses the last hit in this diagonal (line 7). If the distance is smaller than the threshold, this hit pair is stored into the hit pair buffer (line 10). The corresponding position of last hit array is also updated to the current hit (line 11). After pre-filtering, all hit pairs will be sorted using the radix sort (line 12). Note that Algorithm 1 also has a filtering stage after the hit reordering (post-filtering) to filter out the hit pairs that cannot trigger the ungapped extension. We apply the pre-filtering in our evaluations to reduce the overhead of hit reordering.

D. Intra-node and Inter-node Parallelisms

Because different query sequences can be aligned to subject sequences of the database independently, we can parallelize the BLAST algorithms by using the multi-threading programming models on multi-core processors in a compute node, and MPI across multiple nodes. However, there are still many challenges to get good performance, especially in the cache sharing among multiple threads of intra-node,

Algorithm 2: Database Indexed BLASTP Algorithms with Pre-filtering and Hit Reordering

input : DI: database index, Q: query sequences
output: U: high-scoring ungapped alignments

1 **foreach** *database index block* $dIdxBlk_i$ *in* DI **do**
2 **foreach** *sequence* q_i *in* Q **do**
3 $hits \leftarrow hitDetect(dIdxBlk_i, q_i)$;
4 **foreach** hit_j *in* $hits$ **do**
5 $diagId \leftarrow hit.subOff - hit.queryOff$;
6 $seqId \leftarrow hit.seqId$;
7 $lastHit \leftarrow lastHitArr[seqId][diagId]$;
8 $distance \leftarrow hit - lastHit$;
9 **if** $distance < thresholdA$ **then**
10 $hitPairs \leftarrow createHitPairs(hit, lastHit)$;
11 $lastHitArr[seqId][diagId] \leftarrow hit.subOff$;
12 $sortedHitPairs \leftarrow hitSort(hitPairs)$;
13 $extReached \leftarrow -1$;
14 **foreach** $hitPair_i$ *in* $sortedHitPairs$ **do**
15 **if** $hitPair_i.end.subOff > extReached$ **then**
16 $ext \leftarrow ungappedExt(hitPair_i, S, q_i)$;
17 **if** $ext.score > thresholdT$ **then**
18 $U \leftarrow U + ext$;
19 $extReached \leftarrow ext.end.subOff$;
20 **else**
21 $extReached \leftarrow hitPair_i.end.subOff$;

and communication and synchronization of MPI processes of inter-node.

1) Intra-node Parallelism: Algorithm 3 shows the design of multi-threaded BLAST algorithms with OpenMP. To fully utilize the cache after loading one index block, we parallel the inner loop (line 3). When multiple threads execute BLAST search for different queries in parallel, the index and subject sequences can be reused and shared among threads sharing the cache.

Algorithm 3: Optimized Multi-threaded Implementation

input : DI: database index, Q: query sequences
output: G: top-scoring gapped alignments with traceback

1 **foreach** *database index block* $dIdxBlk_i$ *in* DI **do**
2 *#pragma omp parallel for schedule(dynamic)*;
3 **foreach** *sequence* q_i *in* Q **do**
4 $hits \leftarrow hitDetect(dIdxBlk_i, q_i)$;
5 $sortedHitPairs \leftarrow hitFilterAndSort(hits)$;
6 $ungapExts \leftarrow ungapExt(sortedHitPairs)$;
 $gapExts[i] \leftarrow gappedExt(ungappedExts)$;
7 *#pragma omp parallel for schedule(dynamic)*;
8 **foreach** q_i *in* Q **do**
9 $sortedGapExts[i] \leftarrow sortGapExt(gapExts[i])$;
 $G \leftarrow traceback(sortedGapExts[i])$;

2) Inter-node Parallelism: Because the BLAST algorithm is input-sensitive and the execution time is unpredictable, the load balance of BLAST on multiple nodes could be a critical problem. The existing methods partition input queries, database, or both. For example, mpiBLAST [9] partitions the compute nodes evenly to multiple groups, as

well as the database to multiple blocks. Each node group has all database blocks, while each node of the group only has one or several database blocks. At runtime, a dedicated super node is responsible for scheduling input queries to a node group. After searching all database blocks on multiple nodes of the group, the results are merged, sorted, and returned back (only the top picks). This method is also used on cloud with MapReduce model [11], [12], [10]. In this paper, we follow this idea on large scale clusters, but only consider one group of compute nodes.

3) Data Partitioning: We also improve the data partitioning for the load balance. First, we sort the database by sequence length, and distribute sequences into database blocks/partitions in a round robin manner, which can make every database block to have nearly same number of subject sequences following a similar distribution of sequence length. We dispatch the database blocks to compute nodes evenly, and then build the database index on each node in parallel. At runtime, we duplicate the input queries to all compute nodes, and we can get roughly same execution time for a given query on each blocks. Second, in order to further reduce the possible skew between MPI processes on multiple nodes, instead of merging results for individual queries, we merge results after the local alignment for all queries in a batch. We also build a data partitioning framework that can automatically generate a given partitioning algorithm on top of MPI and MapReduce to implement and tune performance of different partitioning algorithms, which is presented in [33].

V. Performance Evaluation

A. Experimental Setup

Platforms: We evaluate our optimized BLASTP algorithms with the database index on modern multi-core CPUs. For the single-node evaluations, the compute node consists of two Intel® Xeon® Haswell CPUs (E5-2680v3), each of which has 12 cores, 30MB shared L3 cache, and 32KB L1 cache and 256KB L2 cache on each core. For the multi-node evaluations, we use 128 nodes of the Stampede supercomputer, that was 10th on the Top 500 list of November 2015. Each node of our multi-node evaluations has two Intel® Xeon® Sandy Bridge CPUs (E5-2680), where each CPU has 8 cores, 20MB shared L3 cache, and 32KB L1 cache and 256KB L2 cache on each core. All programs are compiled by the Intel® C/C++ compiler 15.3 with the compiler flags `-O3 -fopenmp`. All MPI programs are compiled using Intel® C/C++ compiler 15.3 and MVAPICH 2.2.

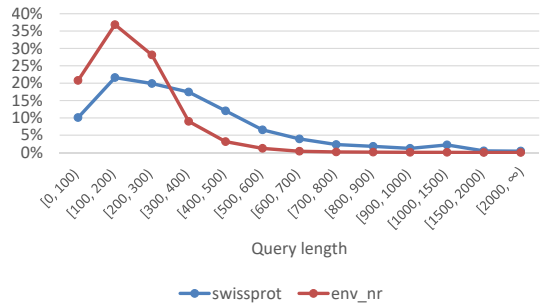


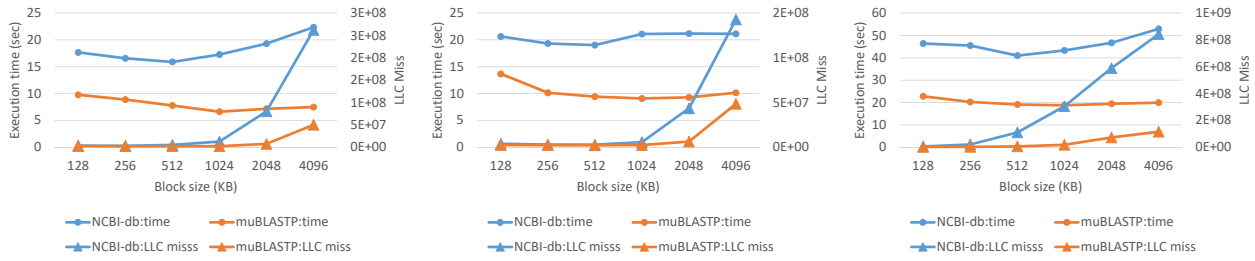
Figure 7: Sequence length distributions of *uniprot_sprot* database and *env_nr* database.

Databases: We choose two typical protein NCBI databases from GenBank [34]. The first is *uniprot_sprot* database, including approximately 300,000 sequences with a total size of 250 MB. The median length and average length of sequences are 292 and 355 bases (or letters), respectively. The second is *env_nr* database, including approximately 6,000,000 sequences with the total size at 1.7 GB. The median length and average length are 177 and 197 bases (or letters), respectively.

Figure 7 shows the distribution of sequence lengths for the *uniprot_sprot* and *env_nr* databases. We observe that the sizes of most sequences from the two databases are in the range from 60 bases to 1000 bases and there are only few sequences longer than 1000 bases. Similar observations were also reported in previous studies for the protein sequence [23].

Queries: According to the length distribution shown in Figure 7, we randomly pick three sets of queries from target databases with different lengths: 128, 256 and 512. To mimic the real world workloads, we prepare the fourth set of queries with the mixed length. This set follows the distribution of sequence length of the target databases. Each set has a batch of 128 queries.

Methods: We evaluate three methods on the single node: the latest NCBI-BLAST 2.30 that uses the query index, labeled as **NCBI**; the NCBI-BLAST 2.30 with the database index as shown in Section III, labeled as **NCBI-db**; and our optimized BLAST, labeled as **muBLASTP**. Note that because there isn't an open sourced BLAST tool using the database index that can get exactly same results of NCBI-BLAST, we implement the second method with our own database index structure but follow the NCBI-BLAST algorithms. On multiple nodes, we compare the MPI version of **muBLASTP** with **mpiBLAST** (version 1.6.0). All performance results in experiments refer to the end to end run times from submitting queries to getting the final results. The database sorting time and index build time is not included, since the index only need to be built once for a given database.



(a) 128 length (b) 256 length (c) 512 length

Figure 8: Performance numbers of multi-threaded NCBI-db and muBLASTP on *uniprot_sprot* database. The batch has 128 queries. The lengths of queries are 128, 256 and 512.

B. Performance with Different Block Sizes

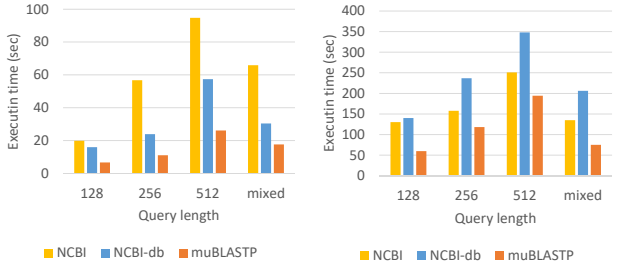
To find the best index block size, we evaluate the performance of database indexed methods, i.e., NCBI-db and muBLASTP, with different block sizes for the *uniprot_sprot* database. Figure 8(a) shows the variable performance. We fix the batch size to 128, having 128 input queries, change the length of query: 128, 256 and 512, and also change the index block size from 128 KB to 4 MB, corresponding to 32K to 1M positions in each index block. The figures show obviously improvements of execution time from muBLASTP in all cases, and the numbers of LLC miss rate give the hint of the reason where the performance comes from: much better cache utilization.

We can also see both the execution time and the LLC miss rate first decrease when increasing the index block size, but later increase rapidly after the 512 KB index block size. The reason for the larger numbers of LLC miss rate for smaller block sizes is the less efficient use of cache line. If the index block size is 512 KB, there are nearly 128K positions (each position is stored in 32-bit Integer). Because one word has 3 letters, there are 24^3 = 13824 possible words. On the average, there are 9 to 10 positions per word (128 * 1024 / 13824), consuming 36 to 40 bytes. Because the index block stores other information than positions, e.g., the lookup table for neighboring words, the block size at 512 KB can fully utilize the cache line. If the index block size is smaller than 512 KB, when positions corresponding to one word are accessed, the algorithms load a 64 byte cache line in LLC cache. However, only a part of the cache line has the positions for the current word, leading to the underutilization of the cache line. When the index block size is larger than 512 KB, the positions for one word may occupy a full cache line, or contiguously several cache lines. The hardware prefetcher may efficiently load the data. However, as shown in the figure, after 1 MB index block size, the LLC miss rate increases rapidly. The major reason is the overhead to access the last hit array. When the block size is 1 MB, there are nearly 256K positions. Because the length of last hit array is twice of number of positions, the last hit array roughly occupies 2 MB memory for each threads, totally 24 MB for 12 threads in our test platform having 30 MB LLC. If the block size is larger than 1 MB, it is highly possible the memory access on the last hit array will lead to severe LLC miss because the last hit array access is random. Without the optimizations of eliminating irregularities, the performances of NCBI-db are reduced much more than those of muBLASTP for larger index block sizes.

Based on the discussion above, to fully utilizing hardware prefetcher, we need to estimate a proper block size to make both index block and the lasthit array fit into L3 cache. Since the lasthit array size for t threads is $t*b*2$, where b is block size, for the given L3 cache size $L3$, we can estimate the optimal block size b by $b = L3/(t*2+1)$.

C. Comparison with Multi-threaded NCBI-BLAST

Figure 9 illustrates the performance comparisons of muBLASTP with NCBI and NCBI-db on two types of protein databases. Figure 9(a) and Figure 9(b) show that muBLASTP can achieve up to 5.1-fold and 3.3-fold speedups over NCBI on the *uniprot_sprot* and *env_nr* database, respectively. Compared to NCBI-db, muBLASTP can deliver up to 3.3-fold and 3.9-fold speedups on the *uniprot_sprot* and *env_nr* database.



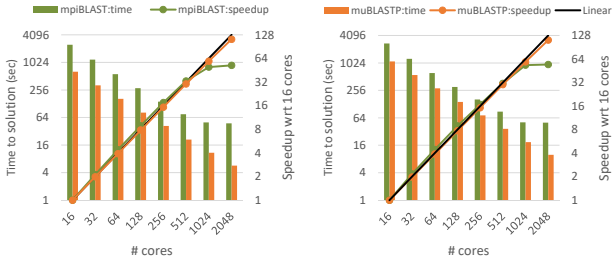
(a) *uniprot_sprot* (b) *env_nr*

Figure 9: Performance comparisons of NCBI, NCBI-db and muBLASTP on *uniprot_sprot* and *env_nr* database.

The figure also illustrates on the larger database, i.e., *env_nr*, with the database indexed NCBI-BLAST (NCBI-db) cannot gain the performance benefit over the query indexed NCBI-BLAST (NCBI). The major reason is the irregularities in the BLAST algorithms lead to further more performance degradation on larger database index. Our optimizations in muBLASTP are designed to resolve these issues and can deliver better performance than NCBI-BLAST no matter which indexing methods are used.

D. Comparison with mpiBLAST on Multiple Nodes

Figure 10 compares the execution time and multinode scaling of muBLASTP and mpiBLAST on up to 128 nodes. As there is no multithreading in mpiBLAST, we launch 16 MPI processes per node for mpiBLAST, while in muBLASTP, we launch one MPI process per node with 16 threads. As expected, our method performs significantly faster on a single node. On multiple nodes, our method and mpiBLAST achieve strong scaling efficiencies of 88-92% and 31-57%, respectively, while scaling from one node (16 cores) to 128 nodes (2048 cores). As a result, on 128 nodes, we can achieve a speedup between 2.2-fold and 8.9-fold speedups over mpiBLAST.



(a) 128 length (b) 256 length

(c) 512 length (d) mixed length

Figure 10: Comparisons of execution time and speedup of multi-node implementation of muBLASTP over mpiBLAST on *env_nr* database.

E. Verification of Alignment Results

To verify that every change or optimization on the BLAST algorithm does not affect outputs, we verify the outputs of every stage in our implementations, including NCBI-db and muBLASTP, are exactly the same as NCBI-BLAST for all test datasets.

VI. Related Work

Many studies have conducted to improve the performance of BLAST tools. NCBI BLAST+ [2], [1] uses pthreads to speedup BLAST on a multicore CPU. On CPU clusters and clouds, TurboBLAST [8], ScalaBLAST [13], mpiBLAST [9], Orion [14], Cloudblast [11], and Azureblast [12] have been proposed. Among them, mpiBLAST is widely used and can strictly follow the algorithms of NCBI-BLAST, which is the gold standard. With the efficient task scheduling and scalable I/O subsystem, mpiBLAST can leverage tens of thousands of processors to speedup BLAST.

To achieve higher throughput on a single node, BLAST has also been mapped and optimized onto various accelerators, such as GPUs [3], [4], [35], [6], [36], [7]. While recently many optimizations have been proposed for BLAST on accelerators, research on multi-core processors is relatively thin, despite the fact that multi-core processors are the most ubiquitously available hardware to researchers, and modern multi-core processors are not far behind many-core processors in peak performance.

Because the hit detection is one of the most time-consuming parts in BLAST, to achieve higher throughput, several index structures are developed to boost the hit detection. For example, Deterministic Finite Automaton(DFA), which is introduced by FSA-BLAST [37], is multiple times smaller than traditional lookup table and more cache-conscious, and widely used for fast pattern-matching [38]. To improve cache performance, NCBI-BLAST also introduces a couple of optimizations into lookup table [16]. First is *pv* array (presence vector), which use a bit array to present if a cell in the lookup table contains query positions. The second is thick backbone, where couples of query positions are embedded into lookup table as there are few query positions for a cell. However, all these techniques are designed for the query index, which has many empty entries and thin entries (few positions in an entry). For the database index, as there are millions of positions of words from subject sequences of the database, every entry of word may contain tons of positions.

Instead of using the query index, a serial of alternative approaches perform hit detection based on the database index, such as [19], [21], [18], [20], [22]. Most of these studies report better performance can be achieved by using the database index, while in this work, we demonstrate that once we want to provide the same accuracy with the database index based methods as NCBI-BLAST, we have to add the overlapping and neighboring words into the database index; and then increased index with the irregularities of traditional BLAST algorithms will kill the performance on multi-core processors. The preliminary research [22] is the only database indexed search tool for protein sequence, that can provide exactly the same accuracy as NCBI-BLAST. To reorder hits for the ungapped extension, the preliminary research uses two-level binning method, that groups hits by diagonal ID first, and then by sequence ID. Compared with our radix sort with pre-filtering method, the two-level binning method has a couple of problems: First, it needs a large amount of preallocated memory for a large number of bins; Second, without pre-filtering, it suffers from heavy data movement to bin a huge number of hits.

VII. Conclusion and Future Work

In this paper, through in-depth performance analysis on a database indexed BLASTP algorithm based on NCBI-BLAST, which can provide the exactly same accuracy

as NCBI-BLAST, we first identify that applying original heuristic algorithm of BLAST on database indexed search will result in serious performance problem, because of random memory accesses for ungapped extension across different subject sequences. To eliminate irregularities in BLAST algorithm, we propose muBLASTP, an optimized database indexed BLAST algorithm for protein sequence search on multicore CPUs and CPU clusters. In muBLASTP, we decompose hit search and ungapped extension stage, and sort hits via radix sort to remove irregularities. And to minimize the overhead of radix sort, we introduce hit pre-filtering ahead radix sort to filter out hits, which are far away each other. With refactored data partition mechanism, we achieve an ideal load balance, and minimum resource contention and synchronization for both intra-node and inter-node parallelism. Experimental results that muBLASTP can achieve up to 5.1-fold speedup over query indexed NCBI-BLAST, and up to 3.9-fold speedup over database indexed NCBI-BLAST. On 128-node CPU cluster, our multi-node implementation can provide linear scaling and achieve up to 8.9-fold speedup over mpiBLAST. In the future work, we will extend our muBLASTP for very long queries.

ACKNOWLEDGEMENTS

This research was supported in part by the NSF BIGDATA Program via IIS-1247693 and the NSF XPS Program via CCF-1337131. We thank Parallel Computing Lab at Intel Corporation and Advanced Research Computing at Virginia Tech for support.

REFERENCES

[1] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic Local Alignment Search Tool." *J. Molecular Biology*, vol. 215, no. 3, pp. 403–410, 1990.

[2] C. Camacho, G. Coulouris, V. Avagyan, N. Ma, J. S. Papadopoulos, K. Bealer, and T. L. Madden, "BLAST+: Architecture and Applications." *BMC Bioinformatics*, vol. 10, p. 421, 2009.

[3] J. Zhang, H. Wang, H. Lin, and W. Feng, "cuBLASTP: Fine-Grained Parallelization of Protein Sequence Search on a GPU," in *29th IEEE Int'l Parallel & Distrib. Proc. Symp.*, 2014.

[4] J. Zhang, H. Wang, and W. Feng, "cuBLASTP: Fine-Grained Parallelization of Protein Sequence Search on CPU+GPU," *IEEE/ACM Trans. Comput. Biol. Bioinf.*, vol. PP, no. 99, pp. 1–1, 2015.

[5] N. Wan, H. Xie, Q. Zhang, K. Zhao, X. Chu, and J. Yu, "A Preliminary Exploration on Parallelized BLAST Algorithm Using GPU," *Computer Engineering & Science*, vol. 31, no. 11, pp. 98–112, 2009.

[6] W. Liu, B. Schmidt, and W. Muller-Wittig, "CUDA-BLASTP: Accelerating BLASTP on CUDA-enabled Graphics Hardware," *IEEE/ACM Trans. Comput. Biol. Bioinf.*, vol. 8, no. 6, pp. 1678–1684, 2011.

[7] K. Zhao, and X. Chu, "G-BLASTN: Accelerating Nucleotide Alignment by Graphics Processors," *Bioinformatics*, vol. 30, pp. 1384–1391, 2014.

[8] R. D. Bjornson, A. H. Sherman, S. B. Weston, N. Willard, and J. Wing, "TurboBLAST(r): A Parallel Implementation of BLAST Built on the TurboHub," in *16th IEEE Int'l Parallel & Distrib. Proc. Symp.*, 2002.

[9] A. E. Darling, L. Carey, and W.-c. Feng, "The Design, Implementation, and Evaluation of mpiBLAST," in *4th Int'l Conf. on Linux Clusters*, 2003.

[10] H. Lin, X. Ma, J. Archuleta, W.-c. Feng, M. Gardner, and Z. Zhang, "Moon: Mapreduce on Opportunistic Environments," in *19th ACM Int'l Symp. on High Performance Distrib. Comput.* ACM, 2010, pp. 95–106.

[11] A. Matsunaga, M. Tsugawa, and J. Fortes, "CloudBlast: Combining MapReduce and Virtualization on Distributed Resources for Bioinformatics Applications," in *4th IEEE Int'l Conf. on e-Science*. IEEE, 2008, pp. 222–229.

[12] W. Lu, J. Jackson, and R. Barga, "AzureBlast: A Case Study of Developing Science Applications on the Cloud," in *19th ACM Int'l Symp. on High Performance Distrib. Comput.* ACM, 2010, pp. 413–420.

[13] C. Oehmen, and J. Nieplocha, "ScalaBLAST: A Scalable Implementation of BLAST for High-Performance Data-Intensive Bioinformatics Analysis." *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, no. 8, pp. 740–749, 2006.

[14] K. Mahadik, S. Chaterji, B. Zhou, M. Kulkarni, and S. Bagchi, "Orion: Scaling Genomic Sequence Matching with Fine-Grained Parallelization," in *High Performance Comput., Netw., Storage and Analysis, SC14: Int'l Conf. for*, Nov. 2014, pp. 449–460.

[15] S. Wu, W. Li, L. Smarr, K. Nelson, S. Yooseph, and M. Torralba, "Large Memory High Performance Computing Enables Comparison Across Human Gut Microbiome of Patients with Autoimmune Diseases and Healthy Subjects," in *the Conf. on Extreme Science and Engineering Discovery Environment: Gateway to Discovery*, ser. XSEDE '13. New York, NY, USA: ACM, 2013, pp. 25:1–25:6.

[16] J. Papadopoulos. (2008) The Developers Guide to BLAST. http://www.boo.net/ jasonp/DevelopersGuideToBLAST.doc.

[17] M. Cameron, H. E. Williams, and A. Cannane, "Improved Gapped Alignment in BLAST," *IEEE/ACM Trans. Comput. Biol. Bioinf.*, vol. 1, no. 3, pp. 116–129, 2004.

[18] W. J. Kent, "BLAT–the BLAST-like Alignment Tool," *Genome Research*, vol. 12, no. 4, pp. 656–664, apr 2002.

[19] Z. Ning, A. J. Cox, and J. C. Mullikin, "SSAHA: a Fast Search Method for Large DNA Databases," *Genome Res*, vol. 11, no. 10, pp. 1725–9, Oct 2001.

[20] A. Morgulis, G. Coulouris, Y. Raytselis, T. L. Madden, R. Agarwala, and A. A. Schäffer, "Database Indexing for Production MegaBLAST Searches." *Bioinformatics*, vol. 24, no. 24, p. 2942, 2008.

[21] H. E. Williams, "Cafe: An Indexed Approach to Searching Genomic Databases," in *21st Int'l ACM SIGIR Conf. on Research and Development in Information Retrieval*, New York, NY, USA, 1998.

[22] J. Zhang, S. Misra, H. Wang, and W. Feng, "muBLASTP: Database-Indexed Protein Sequence Search on Multicore CPUs," *BMC Bioinformatics*, vol. 17, no. 1, p. 443, 2016.

[23] M. Cameron, "Efficient Homology Search for Genomic Sequence Databases," Ph.D. dissertation, School of Computer Science and Information Technology, RMIT University, Nov 2006.

[24] X. Wu, "Improving the Performance and Precision of Bioinformatics Algorithms," Master's thesis, University of Maryland, Aug 2008.

[25] A. Brahme, *Comprehensive Biomedical Physics*, 1st ed. Elsevier, Oct 2014, vol. 6.

[26] T. Rognes, "Faster Smith-Waterman Database Searches with Inter-Sequence SIMD Parallelisation," *BMC Bioinformatics*, vol. 12, no. 1, 2011.

[27] K. Hou, H. Wang, and W.-c. Feng, "AAlign: A SIMD Framework for Pairwise Sequence Alignment on x86-Based Multi-and Many-Core Processors," in *30th IEEE Int'l Parallel & Distrib. Proc. Symp.*, May 2016, pp. 780–789.

[28] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey, "Efficient Implementation of Sorting on Multi-core SIMD CPU Architecture," *the VLDB Endowment*, vol. 1, no. 2, pp. 1313–1324, 2008.

[29] N. Satish, M. Harris, and M. Garland, "Designing Efficient Sorting Algorithms for Manycore GPUs," in *23rd IEEE Int'l Parallel & Distrib. Proc. Symp.* IEEE, 2009, pp. 1–10.

[30] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey, "Fast Sort on CPUs and GPUs: a Case for Bandwidth Oblivious SIMD Sort," in *2010 ACM SIGMOD Int'l Conf. on Management of Data*. ACM, 2010, pp. 351–362.

[31] H. Inoue and K. Taura, "SIMD-and Cache-friendly Algorithm for Sorting An Array of Structures," *VLDB Endow.*, vol. 8, no. 11, pp. 1274–1285, 2015.

[32] K. Hou, H. Wang, and W.-c. Feng, "ASPaS: A Framework for Automatic SIMDization of Parallel Sorting on x86-based Many-core Processors," in *29th ACM on Int'l Conf. on Supercomputing*. ACM, 2015, pp. 383–392.

[33] H. Wang, J. Zhang, D. Zhang, S. Pumma, and W.-c. Feng, "PaPar: A Parallel Data Partitioning Framework for Big Data Applications," in *31st IEEE Int'l Parallel & Distrib. Proc. Symp.* IEEE, 2017.

[34] D. A. Benson, K. Clark, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, and E. W. Sayers, "GenBank." *Nucleic Acids Research*, vol. 42, pp. D32–7, 2014.

[35] P. D. Vouzis, and N. V. Sahinidis, "GPU-BLAST: Using Graphics Processors to Accelerate Protein Sequence Alignment." *Bioinformatics*, vol. 27, no. 2, pp. 182–188, 2011.

[36] S. Xiao, H. Lin, and W. Feng, "Accelerating Protein Sequence Search in a Heterogeneous Computing System," in *25th IEEE Int'l Parallel & Distrib. Proc. Symp.*, 2011.

[37] M. Cameron, H. E. Williams, and A. Cannane, "A Deterministic Finite Automaton for Faster Protein Hit Detection in BLAST." *J. Computational Biology*, vol. 13, no. 4, pp. 965–978, 2006.

[38] X. Yu, W.-c. Feng, D. D. Yao, and M. Becchi, "O3FA: A Scalable Finite Automata-based Pattern-Matching Engine for Out-of-Order Deep Packet Inspection," in *the 2016 Sympos. on Archit. for Netw. and Commun. Syst.*, ser. ANCS '16. New York, NY, USA: ACM, 2016, pp. 1–11.