PTHP: Index for Optimizing Genome Assembly Overlapping and Read Alignment

Sherif Magdy Mohamed Abdelaziz Barakat [1*], Roselina Sallehuddin [2], Siti Sophiayati Yuhaniz [3], Raja Farhana Raja Khairuddin [4], Yusliza Yusoff [5]

[1] *School of Computing, Faculty of Engineering, Universiti Teknologi Malaysia, Skudai, Malaysia.* *sherifmagdy.barakat@gmail.com*

[2] *School of Computing, Faculty of Engineering, Universiti Teknologi Malaysia, Skudai, Malaysia.* *roselina@utm.my*

[3] *Razak Faculty of Technology and Informatics, Universiti Teknologi Malaysia, Kuala Lumpur, Malaysia.* *sophia@utm.my*

[4] *Department of Biology, Universiti Pendidikan Sultan Idris Tanjung, Malim, Malaysia.* *rfrk@fsmt.upsi.edu.my*

[5] *School of Computing, Faculty of Engineering, Universiti Teknologi Malaysia, Skudai, Malaysia.* *yusliza@utm.my*

Abstract: Unfortunately, sequencing technology can only access the genome sequence as massive numbers of short strings are called reads. The genome assembly process constructs the complete genome from these reads based on the overlapping between the reads, called the de novo approach, or aligns the reads based on their positions in the available reference genome, called the reference-guided approach. Millions of reads search for overlapping or alignment, a well-known data structure problem called all-against-all. Many studies have proposed indexing such as hash index, prefix tree index, and parallelization technique to optimize the overlapping or the read alignment individually. However, due to the massive data amount and the repeats, limitations still affect the index efficiency, requiring more enhancements. This article introduces a new hybrid index named Prefix Tree Hash Partitioned index(PTHP), which combines prefix-tree index, hash index, pigeonhole concept, and parallelization. PTHP index reveals significant results on the simulation and real dataset, reducing the computational time complexity of overlapping and read alignment, thus the assembly time outperforming prefix tree index and hash index. Improving the performance of overlapping and read alignment using the PTHP index reveals great results in optimizing the hybrid genome assembly that combines both.

Keywords: Data structures; Genome assembly; Genome indexing; Genome assembly overlapping; Genome assembly read alignment; Genome assembly performance; Genome prefix tree index; Genome hash index

1. INTRODUCTION

The genome is an organism's completed Deoxyribonucleic acid (DNA) sequences [1]. Biologists are interested in knowing this chain to understand the behaviour of genetic diseases [2]. However, the current sequencing technology cannot report the complete DNA sequence, instead of massive numbers of short strings called reads [3]. The genome is sequenced accurately. Each region is sequenced many times, often called coverage, often expressed as 1x, 2x, 3x, and so on [4]. This high coverage makes genome data huge data. For example, the human genome consists of approximately three billion letters with sequencing coverage 60X sequence will have a total genome size of 180 billion letters [5]. Reads must be concatenated together to represent the whole genome, called genome assembly. In computer science, two main approaches are used to construct the sequenced genome from reads, the reference-guided approach and the de novo approach. Reference-guided is used when a known reference genome instance drives the assembly by mapping the reads to their offsets in the reference genome, called the read alignment process [6]. On the other hand, the de novo approach constructs the genome without the aid of any reference genome concatenating the reads based on the exact match between their suffix and prefix, called overlapping. Read alignment in reference guided approach works in computational time complexity O(NL) while (N) is the number of the reads, which may be millions, to be aligned to each offset of reference genome with length (L), which may be billions of letters, which is a highly time-intensive process. In de novo, the computational time complexity of overlapping works in time complexity $O(N)^2$, where (N) is the number of reads (millions). Millions against millions for overlapping or against billions of offsets for alignment is a data structure problem called all-against-all, which is the bottleneck of genome assembly[7], [8], [9].

The index helps to speed up the search time, reducing the number of iterations (computational time complexity), and making the algorithm only focus on a specific part of the data instead of all. Mer is a Latin word that means part of the string, while k is the length of this part, k-mer index is one of the eldest strings indexing methods [10]. The Hash index is the most famous implementation of the k-mer index. Hash index is a widely used index in the genome area to reduce suffix-prefix computational time complexity in de novo overlapping. However, the repetitive sequences in the genome might make two keys generate an identical hash and then both of them point to the same bucket, known as hash collisions that slow down the search in the hash index [11]. On the other hand, the suffix tree (S.T.) (also called prefix tree) is the most popular and widely used indexing tool in genomics [12]. It is a robust index to solve complex problems. It is the base of popular sequence alignment tools, such as MUMmer and RePuter. Unfortunately, due to the massive amount of genome data, the construction of ST is highly memory and CPU-consuming [13].

Parallelism techniques applied to genome data solved many performance challenges to reduce the total processing time. Parallelization splits the extensive process into smaller ones running them simultaneously in parallel [14][15]. Also, using high-performance computing such as Apache Spark helped to speed up the genome assembly of [16],[17]. However, parallelization is introduced in genome assembly only in some stages individually.

A hybrid genome assembly approach combines de novo overlapping and reference-guided read alignment to get higher accuracy in assembling the genomes. Despite the success of indexing and parallelization to optimize the overlapping and read alignment individually, the performance optimization of the hybrid assembly approach remains a challenge. Also, the usage of parallelization in the hybrid assembly approach is yet to be fully implemented [17].

2. RELATED WORK

A study that must be considered is that the keys of k-mer index can be extracted as a shape instead of extracting the full length of the k-mers. This way helped to reduce the total index size and increase the index hitting specificity. The idea implemented to develop Homopolymer Compressed k-mers method proposed by [18], significantly enhances read alignment against the reference genome. Also, the same idea was used by [19]And enhanced to develop a mapping-friendly sequence reductions method by [20].

A novel hash index introduced by [21] applied to the human genome using the combination of seed selection(pigeonhole concept) and efficient parallel construction with hash index, which was implemented to design the Fast and Efficient read Mapper mapper (FEM). The results of FEM outperform other state-of-the-art mappers in terms of both speed and memory footprint.

On the other hand, many studies work on reducing the construction time of ST. The study proposed by [22] took advantage of the pigeonhole principle (P.P.), called the seed and extends principle. It separates the read into non-overlapping partitions and then checks the matching between the partitions instead of matching letter by letter. Then, the verification step must be done if there is a match between the partitions, escaping many unnecessary iterations. The combination of prefix tree index and PP on the *Escherichia coli* genome shows significant results performing the overlapping in 49 seconds, while the prefix tree alone is 757 seconds. However, it is applied only for overlapping in de novo only.

The most widely used software for read alignment is BWA-MEM and the recently published faster version BWA-MEM2. Enumerated Radix Tree (ERT) is a study proposed by [23] combined prefix tree with the PP solution resulting in a 2.1X improvement in overall read alignment throughput over BWA-MEM2. However, it is applied only for read alignment in the reference-guided approach.

Sequencing technologies produce a massive number of short reads. The first step of genome assembly is to extract the reads from FASTA or FASTQ files into structured database tables, index the reads, and finally, overlap or read alignment. Parallel computing solved many performance challenges by splitting the extensive process into smaller sub-processes and running them in parallel [24][25]. One of the most incredible ideas to reduce index construction time is to split big genome files into smaller files processed in parallel, introduced in the high-performance k-mer indexing method (Kmerind) by [26]. Kmerind supports the parallel reading of genome reads' files through file partitioning as a part of this algorithm. More enhancement was introduced by the combination of splitting genome files and the usage of high-scaling platforms, which is the GraphSeq study proposed by [27]. GraphSeq splits a big genome file into several small files loading all of the reads in parallel. It runs on a spark platform achieving 13X, speeding up for de novo genome assembly faster than the traditional platform. A recent similar study was introduced by [28]. The IBD algorithm addressed pairwise comparison problems that can be used to overlap two reads or align reads against k-mer of the reference genome [29]. It breaks a significant problem (N^2) into smaller ones using massive parallelization with each (N) comparison. It is applied to the UK Biobank dataset, with a 250X faster time over the standard approach of pairwise alignment.

Hybridization appears in genome assembly in different ways. It can combine data from different platforms to enhance the accuracy of the genome assembly [30]. Also can be done through a combination of de novo and reference-guided approaches introducing the hybrid assembly approach and bypassing the limitations of both approaches. An example of this hybridization is DACCOR, which is a hybrid genome assembly proposed by [31]. DACCOR applied to assemble the bacterial genome Treponema pallidum reveals higher assembly accuracy. However, the hybrid assembly approach comes with significant performance challenges. The total assembly time in the hybrid approach is challenging for any traditional algorithm or hardware architecture. Although the performance challenge is optimized in each approach, no indexing yet been introduced to fit overlapping and reads alignment in the hybrid assembly approach.

Regardless of the success of data structure indexing and parallelization in optimizing overlapping or reads alignment, there are few methods to optimize both overlapping and read alignment. This limitation might challenge the performance of hybrid assemblies that combine overlapping and reads alignment.

3. METHOD

A. Index Construction

Prefix Tree Hash Partitioned index (PTHP) combines the benefits of a few data structure concepts, introducing an enterprise index to optimize all assembly stages. It combines prefix tree and hash indexing to optimize the overlapping process. On the other hand, reads are stored at the index as non-overlapped partitions (pigeonhole concept) to optimize the reads alignment process. The construction combines two parallelization concepts, splitting the big genome file into smaller files loading them in parallel, and constructing the index across multiple nodes in parallel. In this study, the prefix tree index is constructed across four nodes named according to genome letters A, C, G, and T. Six-teen leaves tables are constructed in each Node and named as two genome letters. Genome letters are only four letters. Thus the probabilities of combining two letters generate an array with a length of sixteen. This construction with a fixed number of leaves benefits the low consumption of CPU. However, there are massive iterations in overlapping reads or aligning them against the reference genome. Therefore, a hash index is constructed in each leaf table with a fixed 64 hash groups, each named as three genome letters. The probabilities of combining three letters generate an array of length sixty-four. The PTHP index is constructed in four stages.

In the first stage of the PTHP index construction, the function FilesInitializer is called to create an array of partitioned FASTA files. Let N[] is the array of four physical nodes (N=[A, C, G, T]) and T[] be the array of sixteen leaves tables in each Node (T=[A.A., A.C., CA, CC, C.G.,.., ti]) thus the partition files array F[] can be constructed by the multiplication of N[] and T[] with sixty-four elements (F=[AAA, AAC, AAG,….CCA,…, fi]). The output of this stage is sixty-four Empty FASTA files named as the elements of array F[].

The second stage of the PTHP construction is to call the FileSpliter function to stream reads from the main genome FASTA file or the array of genome files into the sixty-four partitioned files, the previous stage's output. Each read is streamed based on its first three letters into the corresponding partitioned file named these letters. All the reads containing (N) letters are filtered out during this stage to ensure data quality. For instance, reads that start with "ACG" will be written into the partition file "ACG". During this stage, the maximum read length maxL is detected and outputted. The below pseudocode is for FileSpliter function.

```
Let R[] =[r1,r2,r3,.,i] denote an array of reads in a given genome FASTA file
Let F[]=[[AAA,AAC,AAG,..,j] denote the array of partitioned files
Let MaxL denote the maximum read length with default value MaxL=1
Foreach read r in R[] Do
            Let Sub.r=Left (ri,3) denote the first 3 letters of the read
             So: F[j]=Sub.r
             OpenFile (F[j])
                    IF (ri.contains("N") ≠True)
                              Add ri to new line in F[j]
                                  IF(ri.Length> MaxL)
                                       Set   MaxL= ri.length
                                  End iF
                           Else
                           Skip ri with continue
                     End IF
        CloseFile (F[i])
End Foreach
Return MaxL and F[]
```

In the third stage of PTHP construction, The Mapper function maps each partition file to load its data into the corresponding Table in the corresponding Node and also maps the load to work in the dedicated CPU core. For instance, file "ATT" loads its data into node "A" leaf table "T.T." running on CPU core number 1. Then each read is split into no-overlapped partitions through the Partitioner function, which is nested called through the execution of the Mapper function. The number of partitions (P.N.) is calculated based on the partition length (P.L.) as input petameter and maxL. The pseudocode of the Partitioner function as below.

```
Let r is the read to be split into partitions
Let P[]=[P1 ,P2, ….Pn] array of partitions
PN= maxL/ PL
IF(PN is Integer number)  Then return PN
Else   Return PN=  PN to the higher integer(example 4.3 or 4.6       will be 5)
End IF
For n ←1 n=PN
   IF(r.length=0)
     Break For loop
   Else IF(r.length<PL)
     P[n]=r
     Break loop
   Else
     P[n]=substring(r,1, PL)
     ri=substring(r,PL+1,ri.Lengh) // update r
    End IF
End For
Return P[]
```

Finally, in the fourth stage, the Loader function is called to physically insert indexed partitioned reads into their corresponding Table. The loader function extracts the node key (NK), table key (Leaf key) (TK), and hash key (HK). The loader function also adds a unique identity number for each read RID, which is unique across all tables and nodes. The loader function runs in each Node for each Table in parallel, adding significant speed to PTHP index construction. The final output of the Loader function is sixteen tables in each node store indexed partitioned reads as the below pseudocode. Alao, The below example in Figure (1) shows how to index two reads (r1) and (r2). Both reads start with the letter (A), which means they are indexed under Node (A). The second and third letters are (TT), which means that both reads are under the leaf table (T.T.) in Node (A). Each read in this leaf is indexed under its corresponding hash group based on the reader's fourth, fifth, and sixth letters, which match the exact hash group name. In the below example, the fourth, fifth, and sixth letters of r1 are (AAA), which means that r1 is indexed under the hash group (AAA) while r2 is under the hash group (AGC).

```
Let N[] = [A, C, G, T] denote an ordered of nodes
Let F[]=[AAA,AAC,AAG,j] denote the array of partitioned files of genome reads
Let T[] = [AA, AC, AG, AT, CA, CC, CG,…….., k] denote an ordered of tables in each N[i]
Let R =[r1,r2,r3,…..,ri] denote an array of raw data in each file in each F[]
//In Parallel
OpenNode (N[i])
//Select only the file from F[] that its name start with the same node name
Foreach file in F[j] where substring(F[j],1)= N[i]
          Let T[k]= Substring(F[j],2,2) donate the second and third letters of file name which corresponds to
          the table name and all reads in this file their second and third letter are same.
                Foreach read r in F[j]
                    //Index the read
                    Let HK=substring(ri,3,3) is the hash which is letters 4,5 and 6 from the read
                    //Split the read into non-overlapped partitions
                    Let P[] the array of partitions of read
                    //Create unique identity number of read
                    Let RID is the unique identity of r
                    //Insert indexed read to corresponding table
                    Insert into table T[k] (RID, HK, P[1], P[2]), P[3],..P[PN])
                End Foreach
End Foreach
CloseNode (N[i])
```

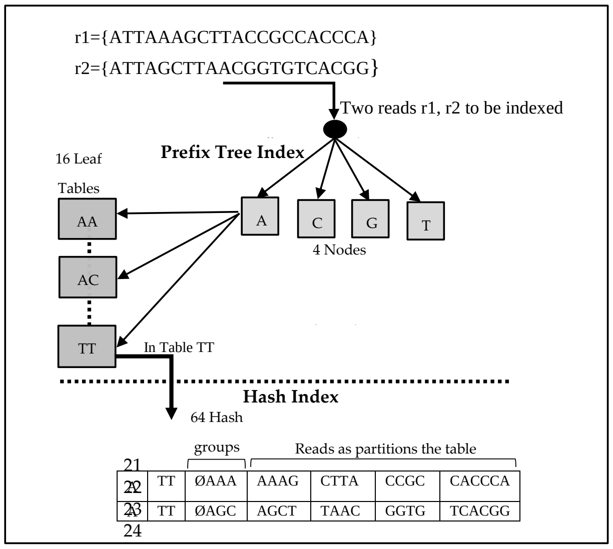

Figure 1. Example of indexing two reads using PTHP index

The final output PTHP construction is sixteen tables in each Node (four nodes) storing indexed partitioned reads at their hash groups, as shown in Figure (2).

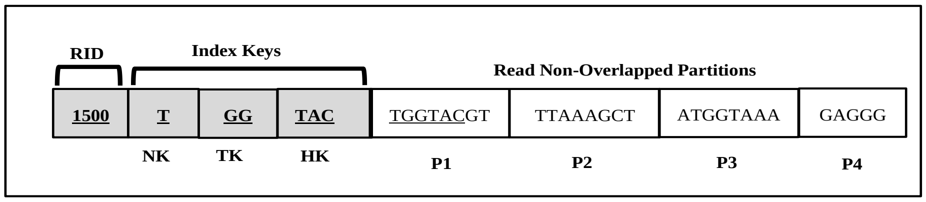

Figure 2. Example of indeed read

The reference genome file is indexed similarly to reads indexing after overlapping processes with only two differences. First, the indexed k-mers of a reference genome for each Node are stored in one Table in that Node instead of the sixteen tables. For example, all k-mers of the reference genome that started with the letter (A) are stored in one Table in Node (A). The second difference is that the Loader function does not add RID for the k-mers. Instead, they are uniquely identified by their original offsets (their positions in the reference genome), which are detected during the file splitter stage of the reference genome, as shown in figure(3). The length of k-mer is calculated based on the average of maxL before overlapping and after overlapping to ensure the minimal difference between k-mer length and contig length after overlapping.

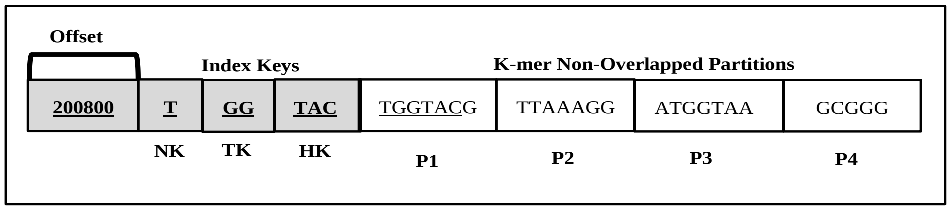

Figure 3. Example of indexed k-mer of the reference genome

B. QUERYING PTHP INDEX

The overlapping process is implemented in three main steps. The first step is to load all indexed suffixes into the suffix table in the master node, and then all indexed prefix reads are loaded into the prefix table. The second step is to concatenate matched suf-fix-prefix reads, loading concatenated reads (contigs) into an overlapping table in the master node. Finally, it overwrites suffix reads with corresponding overlapped ones (contigs) and then deletes prefix reads from the dataset. Searching in the hash groups follows binary search, significantly reducing the computational time complexity of over-lapping. The below example in Figure (4) shows a suffix read (r1) searching for overlap-ping in the prefix table.

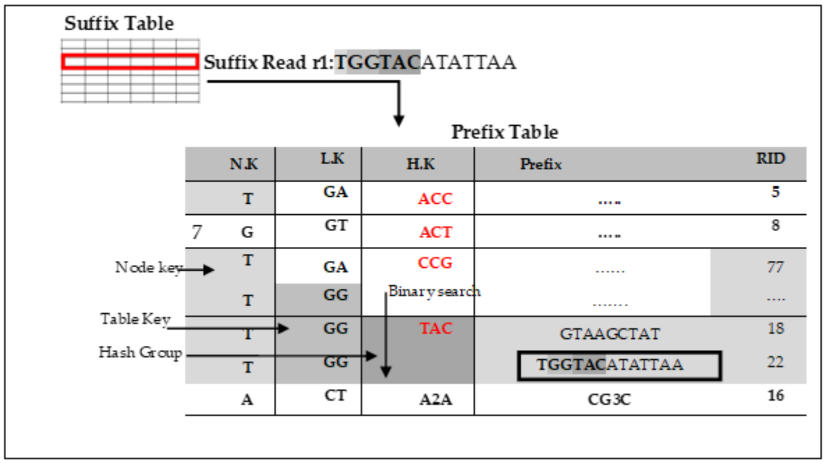

Figure 4. Queqying PTHP index for overlapping

PTHP index stores indexed reads as non-overlapped partitions, and this implementation is to optimize the read alignment process. As shown in Figure (5), If the read (r) hits the k-mer index as a first check, then the partition match is performed. The partition matching is performed as a pattern match instead of the full partition match according to the idea of [19][20]. If the index key is matched and all partitions have a pattern match with the corresponding k-mer partitions, then a full checking match is performed by calling the similarity function. Otherwise, this contig is a mismatch that k-mer and checking the next contig skipping similarity check for this contig. PTHP index is implemented to store all indexed reference genome's k-mers in each Node in one Table, which means that all reads in each Table in each node search for alignment in the k-mers Table stored in the same Node. Reads alignment runs in all nodes in parallel at the same time, reducing the total alignment time.

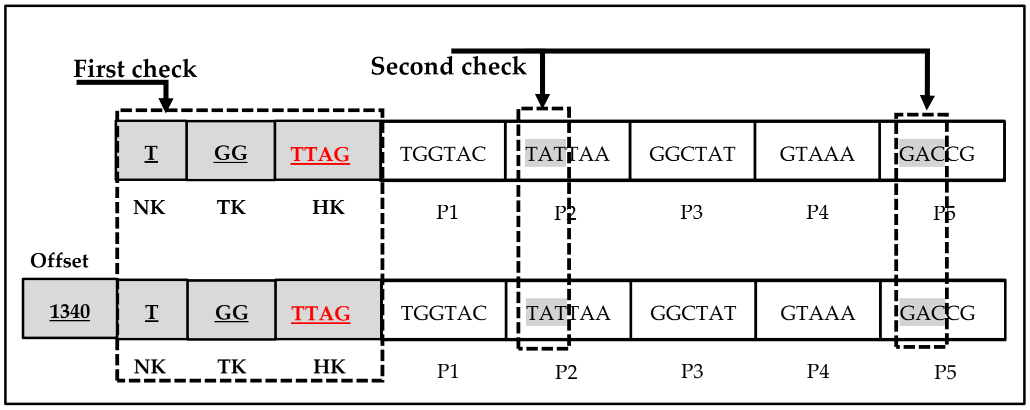

Figure 5. Partition pattern match

Read alignment is implemented in two main steps. The first step is to call the Aligner function to perform the previously mentioned checking index hit and partition pattern match between the read and the expected matched k-mers of the reference genome. If the checking is true, the similarity is called, as shown in the pseudocode below, for the aligner function. We assume in the below example that the similarity threshold of allowed edits (mismatch) is 10%.

```
Let C=[c1, c2, c3,….,cn] is the array of contigs to be mapped
Let P[]={p1, p2, p3, p4, pi} array of partitions
Let P ∈ R[n]
Let K=[k1, k2, k3….,kj] is the k-mer of reference genome in the same node
Let KP=[kp1, kp2, kp3, kp4, kpi] array of k-mers's partitions
KP ∈ K[j]
Let A is pecantage of allowed mismatch as input parameter, asume it is 10%
Foreach contig in C[]
Select from K[] where R[].NK= K[].NK and R[].TK= K[].TK and R[].HK= K[].HK
        IF ( left(R[].P[3],3)= left(K[].KP[3],3) &( left(R[].P[5],3)= left(K[].KP[5],3)
        OR
        //Other pattern match criteria
        Then Run similarity check for (R[], K[])
                IF(SIMILARITY(R[], K[])<=A)
                Then r mapped to k
                End IF
        End IF
End Foreach
```

Mapping reads to the reference genome is not necessarily have an exact match due to individual variations. The approximate match parameter, which sets the number of allowed edits in this research, is a dynamic parameter that depends on the used reference genome. For example, the number of edits can be 10% of the contig length when using the reference genome of the same species. In this case, a close species reference genome is used a higher number of allowed edits is used. The pseudocode below explains the verification step through the similarity function, which accepts the contig and the k-mer re-turning the similarity as a percentage and stopping when the threshold of allowed edits is reached.

```
Let c is the contig
Let k be the reference k-mer
Let Len(c)=Len(k)=L
Let i be the counter i=1 to L
Let m be the number of mismatched characters
Let P= (m/L) * 100  the similarity as a percentage
Let A is the percentage of allowed mismatch as the input parameter, assume it is 10%
do
  IF(SUBSTRING(c, i,1)≠ SUBSTRING(k, i, 1))
    m=m+1
  End IF
  IF(P> A)
    Exit Loop
  End IF
i=i+1
while i<=L
RETURN P
```

4. RESULTS AND DISCUSSION

Fundamentally, the efficiency of the index is evaluated by its impaction in reducing the number of iterations (time complexity). The time complexity directly reflects the total process time, such as overlapping or reads alignment. However, (GC) content is the percentage of two letters, G and C, in the genome. It is a high portion in some genomes, comprising more than 50% (CG). This portion of (CG) [5],[9]repeat negatively affects the efficiency of some indexing methods, especially prefix-tree index, because many reads might start with (CG) or (GC), and they are under the same root, which affects the data distribution across the Node of the index reducing the search in this index [5]. The same problem might happen in the hash index, hash collisions which slow down the search in the hash index [11]. The below pseudocode revealed a significant impaction of the PTHP index, reducing overlapping computational time complexity to be O([(N/2)]* Log2(N/2)) as a linear time complexity instead of O(N)2 under worst-case when half of the dataset reads loaded into the hash group that starts with of (CG).

```
Let N the total number of reads in the dataset
R[]={r1, r2, r3,……, j=N/2} loaded into node (C) table (CG) hash group (CGC)
R[]={r1, r2, r3,……, k=N/2} loaded into node (G)  table (GC) hash group (GCG)
Let j is the number of reads in node C table CG hash group CGC
Let k is the number of reads in node G leaf GC hash group GCG

For j=1← j= N/2    //In Node C
  For k=1  ←  k=N/2    //In node G
    If (suffix-prefix matched)
      Overlap
  End for
End For

Thus: Total iterations in node (C)=(N/2)* (N/2)
Hence: Search in the hash group follow a binary search
Thus: Total iterations in the node (C) works in time complexity:
 $O([(N/2)]* Log_2(N/2))$

According to the above pseudo-code, Let assume that N=1000,000 reads
So: the overlapping time complexity in the node (C) searching in node (G) is
≈O(9.5N) while the original algorithm works in $O(1000000)^2$
```

In order to evaluate the PTHP index, simulation datasets are prepared by iteratively extracting part of the Escherichia Coli genome reference genome, which was downloaded from NCBI under accession number (CU928164.2). We extracted four random regions with lengths of 10080 bases, 20160 bases, 40320 bases, and 161280basese respectively, to be used as reference genomes. Three read datasets were extracted from each previously simulated reference genome with read lengths of 201 bases, 251 bases, and 301 bases. The main reason to generate datasets with different read lengths and read counts is to evaluate the impact of the read length and read count on PTHP index time complexity. Reads are extracted from the reference genomes for each length three times by skipping 25 letters, 50 letters, and 100 letters, as shown in the below pseudo code, to achieve enough coverage.

```
Let refrencefiles=[file1,file2,file3,file4] to be the array of the reference genome files
Let lengths = [201, 251, 301] to be the array of the read length
Let rd is the array of generated FASTA Files with count n=0

Foreatch f in refrencefiles
  Foreach l in  lengths
   For i = 0          i < refrencefiles.Length - l + 1 (i = i + 25)
     rd[n].WriteLine f.Substring(i, l) //To write the read
   For i = 0          i < refrencefiles.Length - l + 1 (i = i + 50)
     rd[n].WriteLine f.Substring(i, l) //To write the read
   For i = 0          i < refrencefiles.Length - l + 1 (i = i + 100)
     rd[n].WriteLine f.Substring(i, l) //To write the read
      n=n+1
  End Foreach
End Foreach
```

PTHP index tested on the simulation of four datasets with the same read length of 201 bases and different read counts, 1212, 2205, 4028, and 15922, respectively, three times using the native prefix tree index [32], native hash index [33] and PTHP index. The over-lapping run iteratively started with an overlapping length of 200 and decreased by one every iteration until 140 bases which consider 40% of the read length according to the recommended minimum imperial overlapping length [34]. Figure (6) below shows significant computational time complexity reduction when using the PTHP index. For example, at read count 2205, the maximum number of iterations to find a suffix-prefix match for overlapping using a hash index is 6130 O(2.8N), and PT index is 3400 O(1.5N), while PTHP performs the overlapping suffix-prefix search in only 433 iterations, which is O (0.2N). The PTHP index reveals a significant reduction in iterations number, approximately lower computational time complexity fourteen times than the hash index, and seven times more than the prefix tree index.

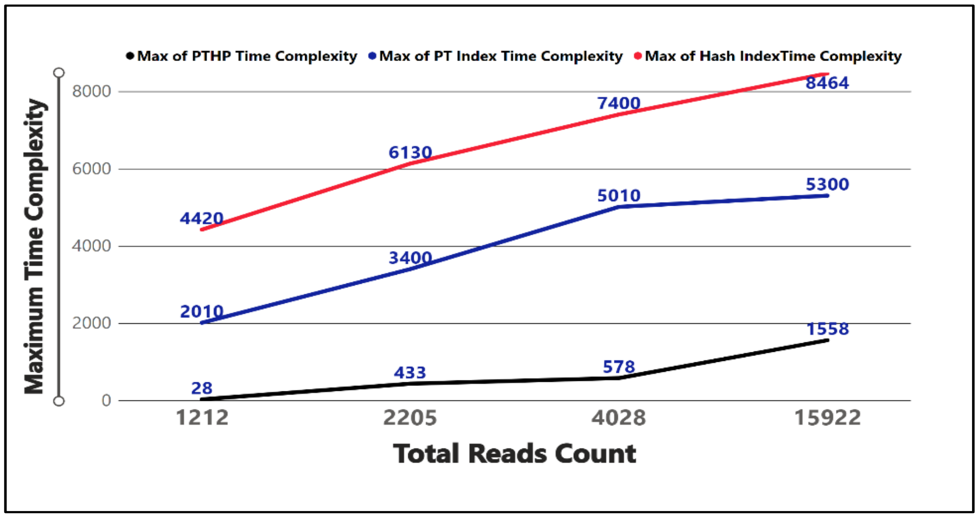

Figure 6. Example of indexing two reads using PTHP index

At the same read length of 201 bases, As a result of reducing the number of iterations to find the overlapping (computational time complexity), the overlapping time is significantly reduced using the PTHP index compared to the prefix tree index and hash index. For example, in figure (7), at read count 4028, the average overlapping time of the hash index is approximately 1.12 seconds, the prefix tree index is 0.60 seconds, while the PTHP index average overlapping time is only 0.05 seconds, approximately 22 times faster than the hash index and 12 times faster than the PT index.

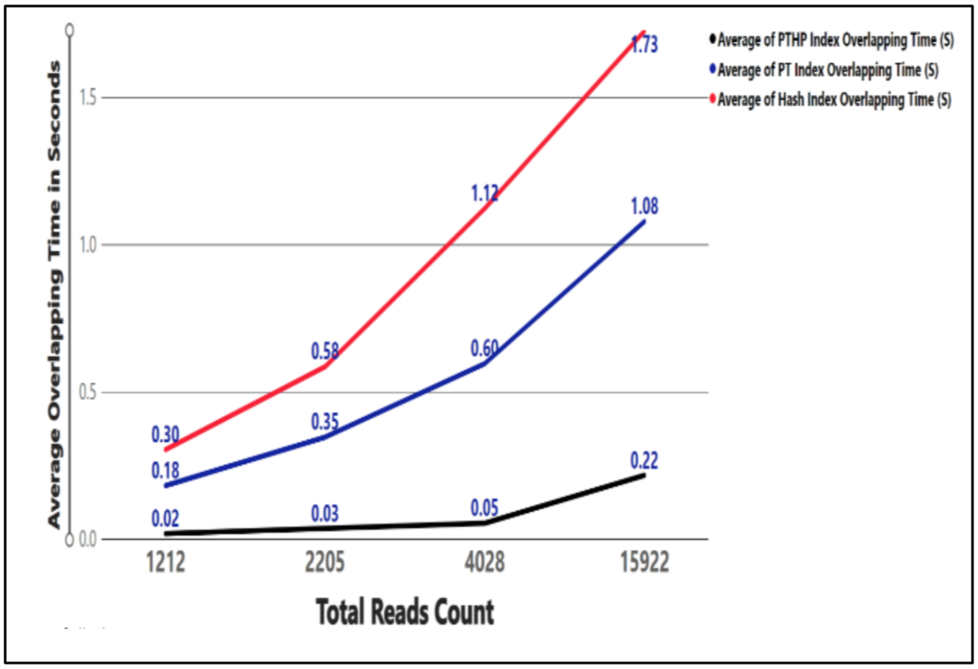

Figure 7. Example of indexing two reads using PTHP index

The PTHP index was also tested on the real dataset for the Escherichia coli genome used by [21] in a superior Prefix tree and pigeonhole solution to solve the all-suffix-prefix problem in overlapping. The PTHP index runs on the same server specifications at the same overlapping length of 30 bases. The results reveal that PTHP, with the combination of prefix tree index, hash index, and parallelization, outperforms both PT and PH solution proposed by [21] in terms of time and space consumption. Table (1) shows that PTHP performs the overlapping of the dataset in just 5 seconds, which is approximately nine times faster than the PH solution. PTHP stores the read and its index keys in the same row. This implementation results in the low space needed to store and run the PTHP in-dex. As shown in Table (1), the PTHP index is stored in 144.4 MB, PT at 230 MB, and PH at 298 MB. Table (1) also shows the significant impact of parallelization on speeding up the overlapping process. For example, the total overlapping time when running the PTHP index in one Node is 23 seconds, while it is approximately faster five times when running the PTHP index on four nodes.

Table 1. **Comparison between PT index, PH solution and PTHP index in space and time consumption**

	PT	PH	PTHP on one Node	PTHP on 4 Nodes
Space consumption	230 MB	298 MB	200.5 MB	144.2 MB
Time consumption	757 second	49 second	23 Seconds	5 Seconds

C. PTHP Index for read alignment

PTHP index store the indexed reads as non-overlapped partitions reducing the number of reiterations aligning the reads to the reference genome by the index hit and pattern partition match. The PTHP index is tested for read alignment on the simulation datasets for two different read counts, 388 and 804, respectively, with the same read length of 201 bases. The results in Figure (8) show the maximum read alignment time complexity with the hash index is 390820, which is approximately O(486N), the PT index is 190043which is O(236N), while the PTHP index is only 16311, which is O(20N), 24 times lower complexity than the hash index and 12 times lower complexity than PT index.

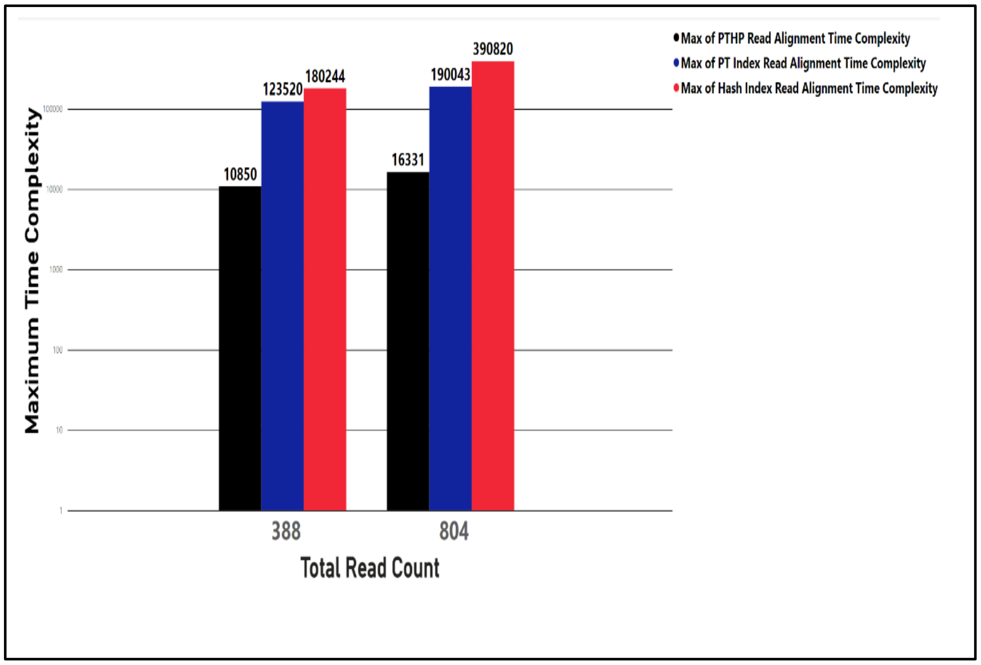

Figure 8. Example of indexing two reads using PTHP index

The results of the PTHP index in Figure (9) at the same read length of 201 bases reveal a significant reduction in alignment time. For example, at reads count 804, the average read alignment time in the hash index is 3104 seconds, and the PT index is 858 seconds, while the PTHP index performs alignment in only 141 seconds, which is approximately faster 22 times than the hash index and six times than PT index.

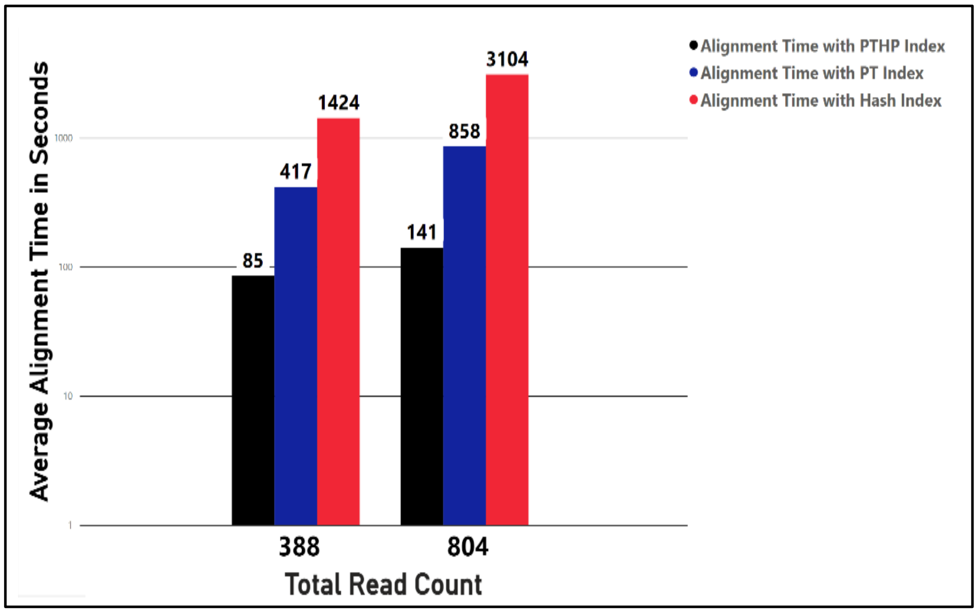

Figure 9. Example of indexing two reads using PTHP index

Figure (10) also shows another factor that impacts the alignment time complexity: partition size. In Figure (10), at read count 804, the time complexity reduced from 16331 to 12221, then 10242 when the window size increased from 51 to 67, then 101, respectively. According to the pattern match between partitions, when the window (partition) size is increased from 51 to 67, the number of petitions is less; thus, the number of patterns is reduced, reducing alignment time complexity because the number of patterns is lower with lower partition numbers.

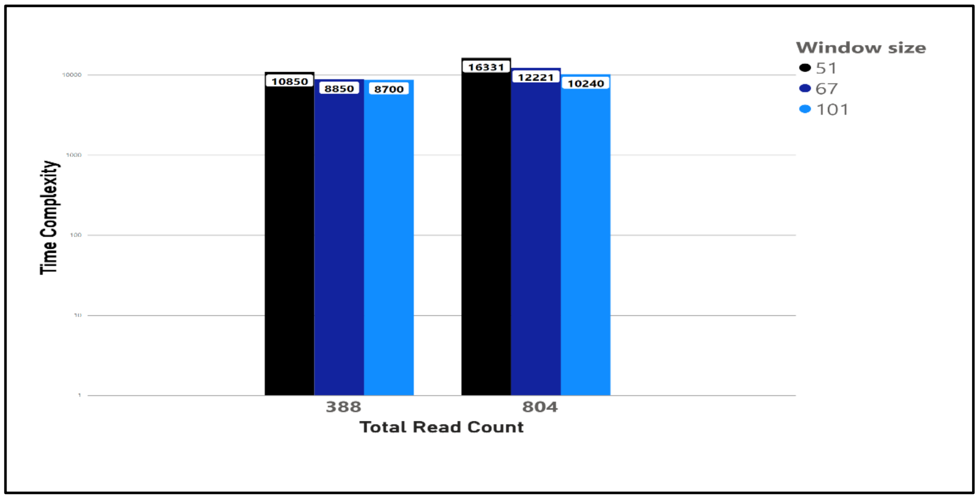

Figure 10. Example of indexing two reads using PTHP index

Also, we tested the PTHP index for read alignment in the real dataset of the human genome with 5 million reads from NCB accession numbers SRR826460 and SRR826471 against the reference genome hg19 downloaded from NCBI. The computational time complexity of the FEM [21] method using the only hash index is 569985422, which is O(113N), and the total read alignment time is 48 seconds. Using the PTHP index with the combination of prefix tree index, hash index, and pigeonhole concept to perform the read alignment on the same dataset reveals significant results in reducing the computational time complexity to only 152120444, which is O(30N) and performing the read alignment in 15 seconds, which is faster 3.7 times than FEM method as shown in Table (2).

Table 2 Comparison between FEM index and PTHP index on the human genome read alignment

Genome	Reference	Index type	Alignment computational time complexity	Total Alignment time
Human Genome	FEM Method	Hash index	569985422 iterations	48 seconds
	PTHP index	PTHP index	152120444 iterations	15 seconds

Also, we tested the PTHP index for read alignment in the real dataset of the human genome with 3 million reads downloaded from platinum genome ERR194161. The computational time complexity of the ERT method [23] using an enhanced prefix tree index is 1463585220, which is O(487N), and the total read alignment time is 6912 seconds. As shown in Table (3), using the PTHP index reduces the computational time complexity to only 364952633, which is O(121N), and performs the read alignment in 4210 seconds, which is faster four times than the ERT index.

Table 3 Table 4 Comparison between ERT index and PTHP index on the human genome read alignment

Genome	Reference	Index type	Alignment computational time complexity	Total Alignment time
Human Genome hg19	ERT Method	Prefix tree index	1463585220 iterations	6912 seconds
	PTHP index	PTHP index	364952633 iterations	2210 seconds

D. PTHP Index for Optimizing Assembly Hybrid Approach

DACCOR is a hybrid assembly [31] with no performance optimization, and it remains its performance optimization is declared as the future work of that study. For this reason, we evaluated PTHP to optimize DACCOR as an example of the hybrid assembly approach. , First, we downloaded DACCOR tools from (https://github.com/Integrative-Transcriptomics/Daccor) then ran DACCOR on Treponema pallidum genome using the exact server specification used by DACCOR (4 CPU cores and 500 GB of RAM) and recorded its overlapping computational time complexity, read alignment computational time complexity. Also, we recorded the total overlapping time and read alignment time. Second, we run the same experiment on the same server using the same dataset using the PTHP index. Table (4) shows that the overlapping time complexity of DACCOR[31] is 4253350, which is O(81N), while the number of reads is 52032. PTHP index run for overlapping on the same dataset reveals a significant reduction of the overlapping computational time complexity to be only O(23N) which speeds up the overlapping in DACCOR 3.5 times faster. Read alignment was also tested using the PTHP index, and Table (4.6) shows that the read alignment time complexity of DACCOR is 1235856 O(23N). When using the PTHP index, the read alignment computational time complexity is only O(9.5N) which speeds up the read alignment in DACCOR 2.4 times faster than DACCOR. The total assembly time in DACCOR without indexing is 321 seconds, while using DACCOR with the PTHP index is only 134, which is 2.3X faster.

Table 5 PTHP index to optimize the overlapping and read alignment in the DACCOR hybrid assembly approach

Method	Overlapping computational time complexity	Overlapping Time	Read alignment computational time complexity	Alignment time	Total Assembly time
DACCOR without index	4253350	76 S	1235856	245 S	321 S
DACCOR with PTHP index	1221354	22 S	494831	112 S	134 S

5. CONCLUSION

Genome data is very massive data. Millions of reads need to be assembled. Overlapping millions of reads in de novo or aligning them against billions of k-mers of the reference genome is a very high computational challenge. Tremendous efforts introduce solutions to optimize overlapping and read alignment in each approach, individually using prefix tree index, hash index, and pigeonhole concept. Parallelization is introduced on genome data in some stages of genome assembly, such as streaming reads from big genome FASTA files and the construction of some indexing methods. However, the construction time is the bottleneck of the prefix tree index, while the collision is the bottleneck of the hash index. Also, no parallelization has been introduced for all stages of genome assembly. Regardless of individual efforts to enhance overlapping or reads alignment individually, the optimization of both together in the hybrid assembly approach is still a big challenge.

This article introduces the PTHP index as an enterprise solution to fit the optimization of overlapping and read alignment together. The PTHP index combines the prefix tree index with fixed depth reducing its construction time, with the constructed hash in-dex at each leaf of constructed prefix tree, reducing its collision degree. This combination outperforms both, adding significant optimization of overlapping in linear time complexity. The pigeonhole concept of PTPH with parallelization significantly speeds up the read's alignment process. PTHP index on real datasets reveals significant optimization results for overlapping and read alignment in terms of computational time complexity and total time.

Using the PTHP index to optimize the hybrid assembly, DACCOR reveals a total assembly time of 2.3X faster.

The future work of this study is to increase the level of parallelization by implementing the PTHP index on eight and sixteen nodes to optimize the performance of hybrid assembly for giant genomes such as the Human genome.

ACKNOWLEDGMENT

Special appreciation to reviewers for useful guidance and comments. The authors greatly acknowledge the UTM Fundamental Research Grant (UTM FR), Ministry of Higher Education Malaysia (MOHE) for financial support through grant Vot. No: Q.J130000.2551.20H71. Thank you to the Research Management Center and School of Computing, Universiti Teknologi Malaysia (UTM), for thorough research support..

Conflicts of Interest*: The authors declare no conflict of interest..*

Supplementary Materials*: The following supporting information can be downloaded at: https://github.com/SherifMagdyBarakat/PTHP.git.*

6. REFERENCES

[1]. Baxevanis, A. D. (2020). Biological sequence databases. Bioinformatics, 1-18.
[2]. Medvedev, P. (2019). Modeling biological problems in computer science: a case study in genome assembly. Briefings in bioinformatics, 20(4), 1376-1383.
[3]. Warnke-Sommer, J. D., & Ali, H. H. (2017, May). Parallel NGS Assembly Using Distributed Assembly Graphs Enriched with Biological Knowledge. In 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) (pp. 273-282). IEEE.
[4]. Byrska-Bishop, M., Evani, U. S., Zhao, X., Basile, A. O., Abel, H. J., Regier, A. A., ... & Human Genome Structural Variation Consortium. (2021). High coverage whole genome sequencing of the expanded 1000 Genomes Project cohort including 602 trios. bioRxiv.
[5]. Dominguez Del Angel, V., Hjerde, E., Sterck, L., Capella-Gutierrez, S., Notredame, C., Vinnere Pettersson, O., Amselem, J., Bouri, L., Bocs, S., Klopp, C., Gibrat, J. F., Vlasova, A., Leskosek, B. L., Soler, L., Binzer-Panchal, M., & Lantz, H. 2018. Ten steps to get started in Genome Assembly and Annotation. F1000Research, 7, ELIXIR-148. https://doi.org/10.12688/f1000research.13598.1 7,39
[6]. Kim, J., Ji, M., & Yi, G. (2020). A Review on Sequence Alignment Algorithms for Short Reads Based on Next-Generation Sequencing. IEEE Access, 8, 189811-189822.
[7]. Baichoo, S., & Ouzounis, C. A. (2017). Computational complexity of algorithms for sequence comparison, short-read assembly and genome alignment. Biosystems, 156, 72-85.
[8]. Allmer, J. (2016). Exact pattern matching: Adapting the Boyer-Moore algorithm for DNA searches (No. e1758v1). PeerJ PrePrints.35
[9]. Wu, B., Li, M., Liao, X., Luo, J., Wu, F. X., Pan, Y., & Wang, J. (2018). MEC: Misassembly error correction in contigs based on distribution of paired-end reads and statistics of GC-contents. IEEE/ACM transactions on computational biology and bioinformatics, 17(3), 847-857. 30
[10]. Islam, A. T., Pramanik, S., Ji, X., Cole, J. R., & Zhu, Q. (2015, November). Back translated peptide K-mer search and local alignment in large DNA sequence databases using BoND-SD-tree indexing. In 2015 IEEE 15th International Conference on Bioinformatics and Bioengineering (BIBE) (pp. 1-6). IEEE.14
[11]. Xiaolei, W., Wubin, Q., Chenggang, Z. and Dongsheng, Z., 2015. Kmer-indexer: A Fast K-mer Indexing Program. Studies in health technology and informatics, 216, pp.1083-1083 50
[12]. Labeit, J., Shun, J., & Blelloch, G. E. (2017). Parallel lightweight wavelet tree, suffix array and FM-index construction. Journal of Discrete Algorithms, 43, 2-17. 16
[13]. Wan, S. and Zou, Q., 2017. HAlign-II: efficient ultra-large multiple sequence alignment and phylogenetic tree reconstruction with distributed and parallel computing. Algorithms for Molecular Biology, 12(1), p.25.
[14]. Ocaña, K., & de Oliveira, D. (2015). Parallel computing in genomic research: advances and applications. Advances and applications in bioinformatics and chemistry: AABC, 8, 23.
[15]. Pingali, K.D., Tanay, K.C.P. and Baruah, P.K., 2017, March. GPU accelerated suffix array construction for large genome sequences. In Innovations in Information, Embedded and Communication Systems (ICIIECS), 2017 International Conference on (pp. 1-6). IEEE. 52
[16]. Paul, A.J., Lawrence, D., Song, M., Lim, S.H., Pan, C. and Ahn, T.H., 2019. Using Apache Spark on genome assembly for scalable overlap-graph reduction. Human genomics, 13(1), p.48.33
[17]. Pan, T. C., Misra, S., & Aluru, S. (2018, November). Optimizing high performance distributed memory parallel hash tables for DNA k-mer counting. In SC18: International Conference for High Performance Computing,

Networking, Storage and Analysis (pp. 135-147). IEEE. 20

[18]. Li, H. (2018). Minimap2: pairwise alignment for nucleotide sequences. Bioinformatics, 34(18), 3094-3100.

[19]. Liu, Y., Yu, Z., Dinger, M. E., & Li, J. (2019). Index suffix–prefix overlaps by (w, k)-minimizer to generate long contigs for reads compression. Bioinformatics, 35(12), 2066-2074.

[20]. Blassel, L., Medvedev, P., & Chikhi, R. (2022). Mapping-friendly sequence reductions: Going beyond homopolymer compression. Iscience, 25(11), 105305.

[21]. Zhang, H., Chan, Y., Fan, K., Schmidt, B. and Liu, W., 2018. Fast and efficient short read mapping based on a succinct hash index. BMC bioinformatics, 19, pp.1-14.32, 40

[22]. Haj Rachid, M. (2017) "Two efficient techniques to find approximate overlaps between sequences," BioMed Research International, 2017, pp. 1–8. Available at: https://doi.org/10.1155/2017/2731385. 13,39

[23]. Subramaniyan, A., Wadden, J., Goliya, K., Ozog, N., Wu, X., Narayanasamy, S., Blaauw, D. and Das, R., 2021, June. Accelerated seeding for genome sequence alignment with enumerated radix trees. In 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA) (pp. 388-401). IEEE.

[24]. Shi, L., & Wang, Z. (2019). Computational strategies for scalable genomics analysis. Genes, 10(12), 1017.

[25]. Ellis, M., Georganas, E., Egan, R., Hofmeyr, S., Buluç, A., Cook, B., ... & Yelick, K. (2017, August). Performance characterization of de novo genome assembly on leading parallel systems. In European Conference on Parallel Processing (pp. 79-91). Springer, Cham.

[26]. Pan, T., Flick, P., Jain, C., Liu, Y. and Aluru, S., 2017. Kmerind: A flexible parallel library for k-mer indexing of biological sequences on distributed memory systems. IEEE/ACM transactions on computational biology and bioinformatics.56

[27]. Su, C. T., Chang, M. T., Cheng, Y. C., Li, Y. L., & Wang, Y. T. (2018). GraphSeq: Accelerating String Graph Construction for De Novo Assembly on Spark. bioRxiv, 321729.25

[28]. Paul, A. J., Lawrence, D., Song, M., Lim, S. H., Pan, C., & Ahn, T. H. (2018, December). Sora: Scalable overlap-graph reduction algorithms for genome assembly using apache spark in the cloud. In 2018 IEEE International Conference on Bioinformatics and Biomedicine (BIBM) (pp. 718-723). IEEE.

[29]. Sapin, E., & Keller, M. C. (2021). Novel approach for parallelizing pairwise comparison problems as applied to detecting segments identical by decent in whole-genome data. Bioinformatics, 37(15), 2121-2125.

[30]. Chen, Z., Erickson, D. L., & Meng, J. (2020). Benchmarking hybrid assembly approaches for genomic analyses of bacterial pathogens using Illumina and Oxford Nanopore sequencing. BMC genomics, 21(1), 1

[31]. Seitz, A., Hanssen, F., & Nieselt, K. (2018). DACCOR–Detection, characterization, and reconstruction of repetitive regions in bacterial genomes. PeerJ, 6, e4742.

[32]. Rheinländer, A., Knobloch, M., Hochmuth, N., & Leser, U. (2010, June). Prefix Tree Indexing for Similarity Search and Similarity Joins on Genomic Data. In SSDBM (pp. 519-536).

[33]. Wu, T. D. (2016). Bitpacking techniques for indexing genomes: II. Enhanced suffix arrays. Algorithms for Molecular Biology, 11, 1-16.

[34]. Hui, J., Shomorony, I., Ramchandran, K. and Courtade, T.A., 2016, July. Overlap-based genome assembly from variable-length reads. In ISIT (pp. 1018-1022).

DOI: https://doi.org/10.15379/ijmst.v10i1.2690