2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum

Investigating Memory Optimization of Hash-index for Next Generation Sequencing on Multi-core Architecture

Wendi Wang†‡, Wen Tang†‡, Linchuan Li†‡, Guangming Tan†§, Peiheng Zhang†, Ninghui Sun†§
†*High Performance Computer Research Center, Institute of Computing Technology, Chinese Academy of Sciences*
§*State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences*
‡*Graduate University of Chinese Academy of Sciences*
Email: {wangwendi, tangwen, lilinchuan, zph}@ncic.ac.cn, {tgm, snh}@ict.ac.cn

***Abstract*—Next Generation Sequencing (NGS) is gaining interests due to the increased requirements and the decreased sequencing cost. The important and prerequisite step of most NGS applications is the mapping of short sequences, called reads, to the template reference sequences. Both the explosion of NGS data with over billions of reads generated each day and the data-intensive computations pose great challenges to the capability of existing computing systems. In this paper, we take a hash-index based algorithm (*PerM*) as an example to investigate the optimization approaches for accelerating NGS reads mapping on multi-core architectures. First, we propose a new parallel algorithm that reorders bucket access in hash index among multiple threads so that data locality in shared cache is improved. Second, in order to reduce the number of empty hash bucket, we propose a serialized hash index compression algorithm, which coincides with the sequential access nature of our new parallel algorithm. With reduced hash index size, it also becomes possible for us to use longer hash keys, which alleviates the hash conflicts and improves the query performance. Our experiment on an 8-socket 8-cores Intel Xeon X7550 SMP with 128 GB memory shows that the new parallel algorithm reduces LLC miss ratio to be $8\% \sim 15\%$ of the original algorithm and the overall performance is improved by $4 \sim 11$ times (6 times avg.).**

I. Introduction

The rapid development of high-throughput next generation sequencing (NGS) technologies influences the scope and scale of biological and medical research substantially. It is promising that NGS can be utilized to address a broad range of problems, including genome-wide polymorphisms detection [1], small RNAs analysis [2], de-novo sequencing [3], re-sequencing [4], etc. In these applications, the most important and prerequisite step is the short sequencing reads mapping, where huge amount of short sequences are mapped against given long reference sequences (3Gbps for human genome). The new generation of sequencers (from Illumina, SOLiD and Helics) can generate data in short-read format at the speed of the order of giga base-pair (Gbp) per day [5]. The produced data is organized in small fragments, called reads, with typical length lies in the range of 30bp∼200bp. The processing speed of traditional methods, such as BLAT [6], cannot keep pace with the high-throughput rate of the new sequencers [10].

Therefore, it is not surprising that many new mapping tools have been developed in recent years. Table I summarizes several representative tools for short sequencing reads mapping. There are two main core algorithms for short sequencing reads mapping: FM-index [13] and hash index [4]. The performance of FM-index algorithms slows down when the mismatch threshold get increased [4]. For example, *PerM* can be 2∼8 times faster than the BWT-based algorithms, noticing the high mismatch threshold considered in this paper (5 mismatches in 100bp reads), we omit the discussion of the FM-index algorithms in this paper. There are some alternative algorithms, such as merge sorting, due to the low efficiency, we exclude them in following discussion.

TABLE I
Representative short sequencing reads mapping software.

Software	Algorithm	Space Complexity[1]	Speed[2]
Bowtie [13]	FM-index	$\Theta(n)$	≈3
SOAPv2 [14]	FM-index	$\Theta(n)$	≈4.1
MAQ [11]	Hashing reads	$\Theta(kn)$	≈0.2
RMAP [15]	Hashing reads	$\Theta(kn)$	≈3.8
PerM [4]	Hashing ref.	$\Theta(n)$	≈**5**
SHRiMP [16]	Hashing ref.	$\Theta(kn)$	≈0.15

[1]The n and k denotes the size of the reference sequence and the number of seed patterns respectively.
[2]Measured in terms of aligned Gbp per CPU day.

The processing speed of existing mapping solutions lag behind the data generation speed of the next-generation sequencers. Listed in Table I, setting the mismatch threshold to 2 (which indicates supporting of 2 mismatches in mapping), the speed of mapping 50bp reads against human genome is about 5 Gb/day with *PerM*, which is about 11 times slower than the data generation speed (55 Gb/day) of the Illumina HiSeq 2000 sequencer [8].

The algorithmic innovation of *PerM* results in high speed of short sequencing reads mapping, however, it is far from the requirement of the solution in the future. For example, when we increase the mismatch threshold to 3, the query speed of *PerM* is reduced over 80%. The problem is compounded by the fact that the ever increasing reads length of next-generation sequencers (150bp for now), which requires increasing the mismatch threshold further, can easily exceed current reads length limitation (128bp) of *PerM* and incurs severe performance degradation. Evaluated on 100bp reads, with mismatch threshold of 5, the gap between the data generation speed and the query speed of *PerM* is around 20∼50 times. Therefore, it is necessary to improve the mapping speed further.

In this paper, we investigate how to leverage multi-core architectures to accelerate the mapping based on hash index

978-0-7695-4676-6/12 $26.00 © 2012 IEEE
DOI 10.1109/IPDPSW.2012.83

IEEE computer society

algorithms. As a well-accepted approach that trading space for speed, building hash index can be utilized to accelerate various applications, such as approximate nearest neighbor querying [18] and database indexing [19]. In this regards, our study of optimizing *PerM* can be extended to other applications based on hash index algorithms.

A key finding of our study, of potential use to many algorithms based on hash index beyond short sequencing reads mapping is that rearranging access order of hash index to avoid random memory access, combined with hash index compression, can introduce considerable performance boost. Although there are plenty of existing work dedicated to algorithm and data structure design, orchestrating data layout through memory hierarchy for optimal performance has not been considered yet. Specifically, we make three main contributions in this paper:

- Based on detailed evaluation of an up-to-date NGS mapping algorithm, called *PerM*, we observe that irregular memory accesses and empty hash buckets are two main performance issues for parallelizing this application.
- We propose two optimization strategies to address the aforementioned two problems accordingly. First, a new parallel algorithm being aware of multi-core cache sharing is developed by reordering hash bucket access in hash index. Second, we propose a hash index compression algorithm to reduce the number of empty hash buckets, which enables the using of longer hash key to achieve higher accuracy and lower hash conflicts.
- We evaluate the proposed optimizations by improving *PerM* running on a 64-core SMP system, on which a speedup of 4∼11 times (6 times avg.) is achieved. In particular, the LLC miss ratio and the index size are reduced to be 8%∼15% and 30% of the original *PerM*.

The rest of this paper is organized as follows. A brief introduction to the computing flow of *PerM* is given in Section 2. Followed by detailed performance analysis in Section 3. Optimization approaches are discussed in Section 4, experimental results are discussed in Section 5 followed by related work in Section 6. Finally, we conclude our work in Section 7.

II. PERM AND SHORT SEQUENCING READS MAPPING

In this section, we give a brief introduction to the computing flow of *PerM*, which is an up-to-date and typical implementation of short sequencing reads mapping based on hash index.

The basic framework of *PerM* includes 5 main steps outlined in Algorithm 2. The similarity of genomes can be exploited to do template-based sequence mapping, whose framework underlies all short reads re-sequencing algorithms. *PerM* uses a seeded alignment heuristic to limit end-to-end alignment to pairs of sequences that are priori likely to be highly similar. However, the existence of differences in gene expression indicates that, along with finding accurate sub-strings, it is also crucial to find mapping sites that contain mismatches. There is a threshold of mismatch that must be applied in short sequencing reads mapping in order for the mapping results to be useful, i.e. 5 mismatches for mapping 100bp reads.

A. Hash index generation

Listed in Algorithm 1, the hash index generation process of *PerM* is an one-time off-line pass, and the generated index can be stored on disk for later reuse. The algorithm of generating hash index can be considered as a variation of the bucket sort algorithm, and is composed of 3 major steps: allocating memory for two levels of indices, partitioning data in bucket and sorting items in each bucket. Every consecutive sub-string (the length of sub-string depends on selected seed pattern and hash key weight, *kw*, given by users for expected sensitivity, e.g. 13bp in *PerM*) in reference sequences is used to generate first-level hash table (*L1idx*), while the positions of the sub-strings in reference sequences are stored in corresponding hash buckets in second-level offset index (*L2idx*). In Algorithm 1, the correlation between *L1idx* and *L2idx* is established as follows: First, entries in *L1idx* are used as counters for counting bucket size dynamically (Line 3∼6); Next, noticing that the hash buckets are stored consecutively in increasing order of bucket number in *L2idx*, the counter values in *L1idx* can be converted to absolute offsets in *L2idx* (Line 7∼8), which can be used to address hash buckets in *L2idx* directly. After the step of dynamic memory allocation, the next step is to fill the *L2idx* with offsets in reference sequences with respect to hash key (Line 10∼14).

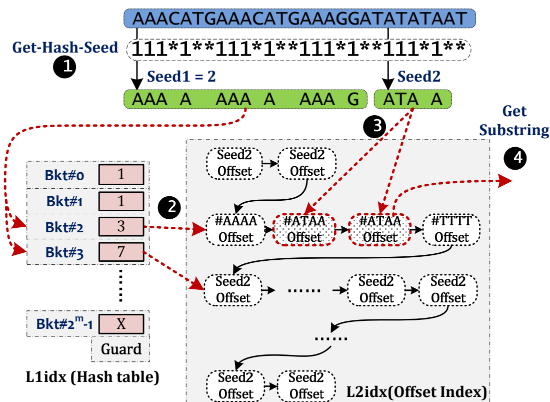


Fig. 1. Query flow of using the two levels of indices.

Due to limited memory budget in *PerM*, only a fraction of hash key (*seed1* in Figure 1) can be used to build first-level hash table. However, in contradiction, in order to improve sensitivity of hash key, the applied hash key weight (*kw*) should be as large as possible. The solution proposed in *PerM* is an additional sorting step in each bucket. At Line 15∼16, executed in a bucket-by-bucket mode, items in each bucket of *L2idx* are sorted in ascending order of the *seed2* field. In this way, the sensitivity of using a shorter hash key (*seed1*) can be improved by invoking binary search of the *seed2* field in bucket, which excludes items that have a different *seed2* value. The two levels of indices in combination provides a fast way for searching approximate mapping positions in reference sequences.

Algorithm 1: Build-Hash-Index Algorithm

```
   input  : rs – reference sequences, kw – hash key weight
   output : idxlist – (L1idx, L2idx, ref) 3-tuple index list
 1 foreach ref in rs do
 2     L1idx = Allocate-Memory(2^kw + 1);
 3     foreach offset in ref do //Count bucket size
 4         seq = Get-Substring(ref, offset);
 5         (seed1, seed2) = Get-Hash-Seed(seq);
 6         L1idx[seed1]++;
 7     foreach i in (0, 2^kw − 1) do //Compute bucket boundary
 8         L1idx[i+1] += L1idx[i];
 9     L2idx = Allocate-Memory(L1idx[2^kw]);
10     foreach offset in ref do //Hash data into buckets
11         seq = Get-Substring(ref, offset);
12         (seed1, seed2) = Get-Hash-Seed(seq);
13         pos = next empty entry in bucket seed1 of L2idx;
14         L2idx[pos] = (seed2, offset);
15     foreach i in (0, 2^kw − 1) do //Sort buckets
16         Sorting L2idx.seed2 in (L1idx[i]:L1idx[i+1]);
17     Store (L1idx, L2idx, ref) in idxlist;
```

Algorithm 2: PerM Algorithm

```
   input  : rl – a DNA readlist, rs – reference sequences
   output : Mapping result of all reads in readlist
 1 idxlist=Build-Hash-Index(rs) //off-line;
 2 foreach (L1idx, L2idx, ref) in idxlist do
 3     foreach read in rl do
 4         //step ❶;
 5         (seed1, seed2) = Get-Hash-Seed(read);
 6         //step ❷;
 7         (start, end) = Query-L1-Index(L1idx, seed1);
 8         //step ❸;
 9         (L, U) = Query-L2-Index(L2idx, start, end,
           seed2) ;
10         foreach i in (L, U) do
11             //step ❹;
12             seq = Get-Substring(ref,L2idx[i].offset);
13             //step ❺;
14             Pair-Wise-Comparison(seq, read);
```

B. Query with Hash Index

Extracting offsets and invoking pair-wise alignment of reads and sub-strings of reference sequences are the key steps involved in the query flow. The first 4 of 5 steps of Algorithm 2 are illustrated in Figure 1. First, at step ❶, hash keys are generated by calling ***Get-Hash-Seed***, which are further divided into two parts. At step ❷, the leading and longer part (*seed1*) is used to access two consecutive entries in *L1idx*, which define an initial access range in *L2idx*. The data structure of *L2idx* is a sequential array, the start position of each bucket is calculated statically and stored in corresponding entry in *L1idx* during hash index generation. Therefore, buckets boundaries in *L2idx* can be trivially decided by accessing two consecutive entries in *L1idx*. Taking the *i*th bucket as an example, the start position and the position next to the end position is stored in the *i*th and the *i*+1th entry in *L1idx*, respectively. Given the bucket number (*seed1*), the corresponding start and end position can be decided by ***Query-L1-Index*** in constant time. Only the items that have the same *seed2* value are of interest in alignment. In order to improve sensitivity of hash key, using the remaining part of hash key (*seed2*), the initial access range, which spans an entire bucket, is refined by ***Query-L2-Index*** at step ❸. The buckets of *L2idx* are pre-sorted, as explained previously, the refinement can be simplified as finding lower and upper bound of given *seed2* value. At step ❹, offsets within the refined access range of *L2idx* are used to retrieve sub-strings of reference sequences. To get the final mapping result, ***Pair-Wise-Comparison*** is invoked at step ❺, which executes string matching between reads and sub-strings of reference sequence. We only present a short description of *PerM* algorithm here and refer readers to their paper [4] for more details.

III. Performance Analysis and Motivation

A. Design Considerations

Short sequencing reads mapping is a data intensive workload. On one hand, *the exploding increasing of gene data leads to high requirement on computer memory size*. Among the softwares listed in Table I, *PerM* consumes more memory space. We argue that the relatively high memory consumption will not be a serious problem because memory systems are getting cheaper and larger. The cost of high performance computers appears to be increasingly dominated by the cost of memory to the point where the amount of memory per core is decreasing. However, for NGS applications, building a system with huge amount of processors is not the only option. For example, in turn, we can build alternative systems with large memory, whereby larger and more efficient hash index can be built. There is a trade-off, though, in that the performance of large amount of processors with moderate memories can be surpassed by fewer processors with larger memories. Therefore, it opens another way to extend computing systems by means of increasing the memory capacity, rather than merely increasing the number of processors.

On the other hand, *there is few floating-point operations and low ratio of arithmetic operations to memory operations so that the performance is bounded by memory accesses in the NGS applications*. Unfortunately, amount of irregular memory accesses are observed in hash index algorithms including *PerM*. The irregularity is mainly caused by random access in hash table and reference sequences, which occupies most of the execution overhead of the mapping process. A dedicated discussion of the irregular memory access is given in Section III-B1. In addition to the problem of irregular memory access, a large portion of empty hash bucket is another performance issue. Leaving most buckets in hash index empty does not take advantage of the cache line mechanism of modern CPUs, with which spatial data locality can be improved by loading adjacent data in advance. As only the non-empty buckets are of interest, considerable memory bandwidth can be wasted on loading hash table entries that point to empty

buckets. By removing empty hash buckets, the hash table size can be greatly reduced, which makes it possible to build hash table with longer hash key (52 bit in this paper) that otherwise would be too big to fit in main memory.

B. Performance Analysis

Due to little arithmetic operations involved in the mapping process of *PerM*, we focus on investigating its memory performance of searching through the hash index. We profile the behavior of cache and memory utilization, which are critical performance features that drive us to develop proper optimization strategies.

1) Irregular memory access: We list the profiled cache miss rate of kernel functions of *PerM* in Table II. The excessive random memory access in hash index accounts for the reported high cache miss rate. In particular, we highlight the sources of random memory access in the query flow with dotted lines in Figure 1. There are plenty of irregular memory accesses in the 4 most time-consuming steps of *PerM*: Attributed to the random nature of hash key generation at step ❶, two consecutive items in first-level hash table (*L1idx*), which are used to define an initial access range in second-level offset index (*L2idx*), are accessed randomly at step ❷; At step ❸, the initial access range is further refined by applying binary search over the *seed2* field; At step ❹, the offsets in the refined access range are used to retrieve sub-strings in reference sequences randomly.

TABLE II
THE LAST LEVEL CACHE MISS OF KERNEL FUNCTIONS

Function	Time Percentage	LLC Miss
Get-Hash-Seed	2.18%	0.04
Query-L1-Index	8.15%	0.44
Query-L2-Index	40.35%	0.62
Get-Substring	24.45%	0.84
Pair-Wise-Comparison	13.12%	0

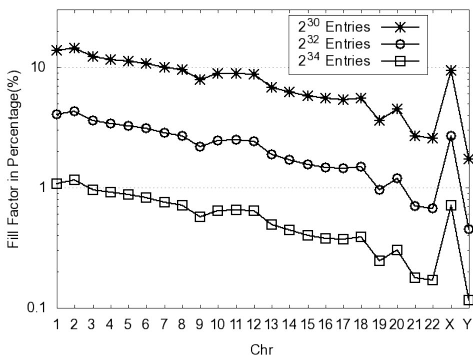


Fig. 2. The fill factors of hash index decrease with the increasing number of entries that is determined by given hash key weight.

2) Empty hash buckets: The nucleotide composition of genome, which exhibits biased presentation of specific nucleotide patterns and structured repeats, is far from random. When building hash index for genome, the generated hash keys, which are summaries of original sequences, are more likely to cluster in few buckets. The severe contentions in few hash buckets indicates that most remaining buckets are empty and wasted, which incurs the problem of low fill factor rate of the hash table. Evaluated on chromosomes of human genome, in Figure 2, the percentage of non-empty buckets in hash tables (*L1idx*) is well below 1% when hash key weight (*kw*) exceeds 16bp. The y-axis illustrates the percentage of presented hash keys in entire hash key space. One observation, which can be explained as that the increasing rate of unique hash key is lower than the increasing rate of hash entries, is that *the fill factor decreases steadily when we increase* kw. Another interesting observation of practical usage is that, although increasing *kw* exerts severe impacts on *L1idx*, the influence on *L2idx* is rather limited. *Therefore, when* kw *get increased, the additional index size is largely attributed to empty entries in* L1idx. In following section, based on this observation, we propose an index compression method to remove empty entries in *L1idx*. Removing empty buckets in hash index also makes it possible to build hash index with larger *kw* that otherwise would be too big to fit in main memory. For example, using our approach, with a *kw* of 26bp, the size of hash index is only 40.5 GB. In contrast, the hash index will occupy a prohibitive 16 PB memory in original *PerM*. Using of longer hash key facilitates short sequencing reads mapping in two aspects: First, the sensitivity of hash key gets increased, whereby false positive within buckets can be reduced. Second, the step of ***Query-L2-Index*** in Algorithm 2, which refines access range by invoking binary search in bucket, can be eliminated. On the other hand, leaving most buckets in hash table empty does not take advantage of the pre-fetch functionality of cache line of modern CPUs. As only part of pre-fetched data is used, most memory bandwidth is actually wasted, which prevents designs from reaching optimal throughput with respect to available parallelism.

IV. PARALLELIZATION AND OPTIMIZATION

To alleviate the gap between the speed of CPU and memory, in modern computers, the memory hierarchy is usually organized into layers. In a typical memory hierarchy, the off-chip main memory is at the bottom layer. Above it, one or more levels of faster but smaller cache memory reside. The order in which applications access data exerts heavy impacts on spatial data locality. To improve spatial data locality, cache system relies on block data transfer to exchange data with main memory. The size of data blocks (a.k.a the cache line) is usually larger than the requested data size of individual read/write instructions of CPUs. In this way, memory operations on adjacent data can be fulfilled by few cache line load/store operations, which dispenses with multiple time-consuming main memory accesses. However, the random access nature of hash index query does not take the advantage of the cache system in modern CPUs. Therefore, how to improve data locality of hash index access determines the extent to which the memory system can be efficiently utilized

and the extent to which the application can be accelerated through parallelization.

A. Parallelism Analysis

Potentially, there are two levels of parallelism to be exploited for the 3 *for-loop* (Line 2, 3 and 10 of Algorithm 2) in *PerM* algorithm. The third level of parallelism in Line 10 of Algorithm 2 is confined by the size of hash bucket. The degree of parallelism at this level is very limited due to non-uniform distribution of hash bucket size shown in Figure 3. In fact, the statistics results of human genome show that the number of items in most of buckets is less than 100. Therefore, it prefers to only focusing on the first two levels of parallelism.

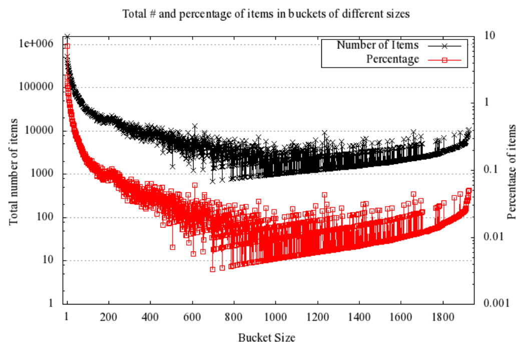


Fig. 3. Distribution of the bucket size of hash index for human genome chr1. The buckets are clustered according to their size.

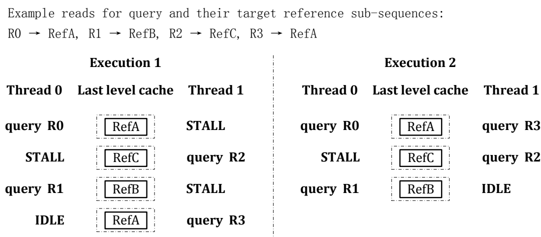


Fig. 4. Performance effect of execution order

At first glance, it is easy to develop the embarrassing parallelism by partition the set of DNA reads among multiple processors (cores). We note that the straightforward implementation does not take into account cache sharing in current multi-core architecture. The number of cores on commercial multi-core architecture increases to 6 or 12, i.e. Intel Westmere and AMD Magny-core. A common feature of these multi-core processors is that the last level cache is shared by all cores on the same chip. A highly efficient utilization of the shared cache is important for multi-threaded program to achieve better performance on multi-core architectures [9]. For an instance of our targeted mapping algorithm, Figure 4 illustrates a possible transformation to increase the utilization of shared cache among multiple threads. In the example there are two threads or cores, each of them maps reads {*R0, R1*} and {*R2, R3*} to reference, respectively. Assume that both *R0* and *R3* match with the same position in reference, which is referred to as *RefA*, *R1* matches with *RefB*, *R2* matches with *RefC*. Without loss of generality, for simplicity we assume that there is only one cache line in the shared cache. Figure 4 shows two execution modes with different query order. *Execution 1* in the left part is an execution order in the naive parallel algorithm, which directly partitions reads among multiple threads. Due to cache line conflict, the execution of two parallel threads is serialized. If we could change the query order, i.e. *thread 1* reads *R3* before *R2*, two threads would share the same data in cache and run in parallel, and then the overall execution time could be reduced. The right part of Figure 4 shows the execution order (*Execution 2*).

B. Reorder Bucket Access in Hash Index

As discussed in previous section, the two levels of indices in *PerM* are organized as sequential arrays and stored in increasing order of hash key. The higher and lower parts of a hash key (*seed1* and *seed2*) are used to address the two levels of indices, which exhibits severe performance issue of random memory access (see Figure 1). To mitigate the problem, built upon the concept of *Hash-Join* [20], we resort to an optimization step to reorder and join query reads with respect to the accessed hash bucket number in reference index.

The *Hash-Join* algorithm is a well-accepted technique, which trading CPU power and memory space for speed, to join two or more distinct data tables with shared data fields in database. In *Hash-Join*, by hashing the shared data fields, one of the data table (usually the smaller one) is converted to a temporary hash table in advance and resided in main memory. Then, the join operation for the larger data tables can be simplified as in-memory hash table probing with respect to the shared data fields. In this way, the time-consuming join operation can be fulfilled by traversing each data table only once. In short sequencing reads mapping that based on hash index, the reference sequences are already expressed in form of hash table, which coincides with the *Hash-Join* technique. In particular, for hash table probing, the step of hashing the shared data fields can be harmoniously replaced with hash key generation. Therefore, by making hash index for query reads as well, the technique of *Hash-Join* can be applied to optimize short sequencing reads mapping. The performance improvement of applying *Hash-Join* can be explained in two aspects: 1) For query reads that share the same hash key, the same bucket in hash index will be accessed. *By joining query reads according to their hash keys, buckets of hash table need to be accessed only once*; 2) In a typical query, over hundreds of millions of reads are processed in batch. *By calculating the hash bucket numbers that reads going to access in advance, and reordering them in increasing order, it becomes possible to access the hash index in strictly increasing order*, which avoids random memory access. In this paper, the optimization of *Hash-Join* is implemented by indexing reads with the same strategy used for indexing reference sequences.

To utilize the sequential nature of the reordered hash index, as listed in Algorithm 3, a new query algorithm is proposed.

The new query algorithm iterates the hash index with respect to the reordered hash bucket numbers and invokes pair-wise alignment on items in corresponding buckets in both reads and reference index. Figure 5 illustrates an example of our query algorithm. The query seeds from two separated reads (each read contains several query seeds), denoted as *A0, A1, A2* and *B0, B1, B2*, are queried in lexicographic order in original *PerM*. In contrast, the buckets of hash index will be probed sequentially from top to bottom in our reordered query algorithm. Therefore, *B1* will be queried first, as it is the only seed that accesses bucket 0. Both *B2* and *A0* access bucket 1, therefore they can be joined together to reduce a hash index access. As a result, the number of bucket access can be reduced from 6 to 4.

Algorithm 3: Reordered hash index query

```
input  : idxlist – (L1idx, L2idx) 2-tuple index list,
         b – (L1idx, L2idx) 2-tuple reads index
output : Mapping result of all reads in b
1  foreach a in idxlist do
2      j = 0;
3      for i in b.L1idx do
4          repeat //Matching bucket number
5              m = b.L1idx[i].bktnum;
6              n = a.L1idx[j].bktnum;
7              if m > n then j++;
8              else if m < n then i++;
9          until m == n;
10         foreach p in jth bucket of a.L2idx do //Alignment
11             foreach q in ith bucket of b.L2idx do
12                 Pair-Wise-Comparison(p.seq, q.seq);
13         i++;  j++;
```

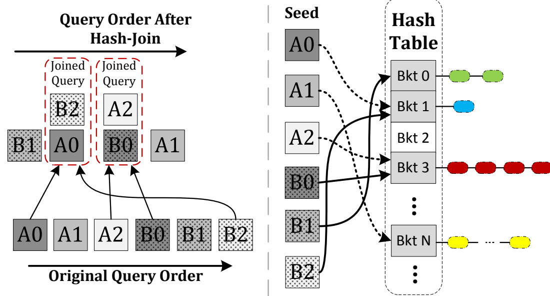


Fig. 5. Reordering queries based on Hash-Join.

The increased data locality comes at the cost of out-of-order result collection. The sequential traversal of hash index gives rise to a different query order, in which even the query order within reads will change. As illustrated in Figure 5, according to given hash key pattern, the reads are divided into several sub-queries, which will be processed sequentially in original *PerM*. However, after applying the *Hash-Join*, the sub-queries will be tangled with each other, which leads to the loss of correspondence between reads and their results. The solution to this problem is an auxiliary post-processing step, which is responsible for collecting scattered results and trimming redundant mapping sites.

In complement, the second optimization takes into account the data access locality in reference sequences. Instead of storing the indirect offsets of reference sequences, the random access in reference sequences can be avoided by storing the sequence data in the first place. Although the memory consumption increases linearly as length of reads increased, we argue that the problem is affordable with large memory at our disposal. Furthermore, leveraging the hash index compression method discussed in following section, we can combine the two levels of indices in *PerM* to form a unified index, whereby the bucket boundary refinement in second-level offset index (Line 9 of Algorithm 2) can be eliminated.

C. Hash Index Compression

Increasing hash key weight (*kw*) is a common practice to reduce hash conflicts and improve query performance. However, the high space overhead stemming from the fact that most hash buckets are empty can easily offset the performance gains. The problem is compounded by the highly biased composition of genome sequences. As explained in Figure 3, we learn that the distribution of generated hash keys is far from random, which can make over 99% of hash buckets empty. The huge amount of empty buckets in hash index slows down our optimized query considerably. Explained in Algorithm 3 (Line 4∼9), after reordering bucket access order, a traversal of the entire hash index is required for finding all matching bucket pairs. Therefore, it is worth while to reduce the hash index traversal time by removing empty buckets. However, operated as mandatory place-holders for constant time data access, it is impossible to remove the empty buckets in traditional hash table.

More significantly, among the steps listed in Algorithm 2, mapping refinement (Line 9∼14) is the most computing-intensive one. As illustrated in last 3 rows in Table II, over 80% of the total execution time is dedicated to mapping refinement in original *PerM*. However, only 7.4% of the invocations of ***Pair-Wise-Comparison*** (Line 14 of Algorithm 2) contributes to positive matching results, which indicates a considerable waste of computing power on false positive alignments. Intuitively, the problem can be alleviated by increasing hash key weight, which in turn improves the resolution ratio of hash key and reduces false positive in hash index. However, when increasing *kw* in traditional approach, the exponential explosion in memory space renders it practical to only limited *kw*. With the hash index compression discussed in this section, building hash index with unprecedented *kw* (26bp) becomes possible.

1) Serialized Index Compression: In the reordered hash index query algorithm, the hash index will be accessed sequentially. In this regard, the empty buckets introduce nothing but pure access overhead. As only non-empty buckets of reference (reads) index are of interest in alignment, space saving can be achieved by probing and storing non-empty buckets in increasing order of bucket number. Intuitively, the highest

compression ratio can be achieved by reserving bucket pairs that are both non-empty in two indices. However, the drawback is that the compressed index is related to given reference and reads index, and the compressed index becomes invalid once either of the index changed. To overcome the drawback, we resort to a static compression algorithm, which can be explained as sequentially removing of empty buckets in both indices. Taking Figure 6 as an example, bucket 3 and 4 of the reference index are empty and will be removed in the compressed index. After squeezing out the empty buckets, bucket 5 will follow bucket 2 directly in the compressed reference index. Algorithm 4 gives the serialized hash index compression algorithm. For clarity, Algorithm 1 and 4 are listed as separated flows. In practice, however, they can be combined to get a unified flow that dispenses with extra data copys. The loss of bucket-to-bucket correspondence is the primary drawback of this approach. Therefore, an additional step of realigning bucket numbers (Line 4∼9 of Algorithm 3) is required in the reordered hash index query algorithm.

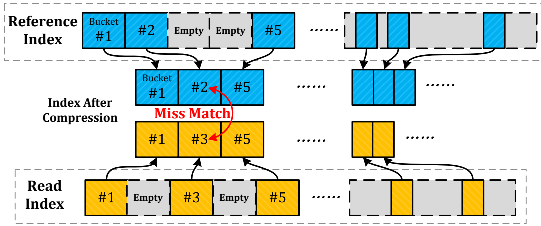


Fig. 6. **Hash index compression by removing empty buckets**. Highlighted as a mismatch between bucket 2 and 3, the mapping of buckets between two indices does not hold after the compression.

Algorithm 4: Serialized hash index compression

```
   input  : idxlist – (L1idx, L2idx, ref) 3-tuple index list
   output : idxlist2 – (L1idx, L2idx) 2-tuple index list
1  foreach a in idxlist do
2      Allocate a 3-tuple index b in idxlist2;
3      idxlist2.b.L2idx = Allocate-Memory(size of a.L2idx);
4      foreach i in a.L2idx do //Mapping & translating L2idx
5          offset = a.L2idx[i].offset;
6          b.L2idx[i].offset = offset;
7          b.L2idx[i].seq = Get-Substring(a.ref, offset);
8      s = Number of non-empty entries in a.L1idx;
9      b.L1idx = Allocate-Memory(s);
10     j = 0;
11     foreach i in a.L1idx do //Compressing L1idx
12         if bucket i of a.L1idx is not empty then
13             b.L1idx[j].start = a.L1idx[i];
14             b.L1idx[j].bktnum = i;
15             j++;
```

2) Comparison of Space and Time Complexity: The comparison of space and time complexity is listed in Table III. For large hash key weight (*kw*), the space overhead of original *PerM*, which grows exponentially with increasing *kw*, is prohibitive. The advantage of serialized hash index lies in the fact that the index size depends on the number of unique

TABLE III
SPACE AND TIME COMPLEXITY. THE n, m AND p REFERS TO THE NUMBER OF THE GENERATED HASH KEYS, THE HASH KEY WEIGHT AND THE NUMBER OF THE UNIQUE HASH KEYS, RESPECTIVELY.

Index Type	Space	Time(best)	Time(worst)
Serialized Hash Index	$\Theta(n)$	$\Theta(1)$	$\Theta(p+n)$
Hash Index in PerM	$\Theta(2^m)$	$\Theta(1)$	$\Theta(n)$

hash key, instead of the *kw* of the applied hash key pattern. Taking the 26bp (52bit) *kw* considered in this paper as an example, although building the prohibitive 16 PB ($2^{52} * 4$ Byte) hash index is impractical, extracting and storing the non-empty buckets for sequential query becomes applicable. As illustrated in Table V, the size of compressed hash index is only about 40.5 GB.

The worst case time complexity is reached when all hash keys are clustered in a single bucket. For serialized hash index, iteration of the entire reference index (Line 4∼9 of Algorithm 3) is needed, which attributes to the p factor in worst case time complexity. Although it is needed to iterate the entire reference index, with huge amount of reads queried in batch, the amortized query time of serialized hash index actually approaches $\Theta(1)$.

V. Experimental Results

A. Experiment Setup

The human genome from GenBank [25] is used as the reference sequences in following experiments. The reads length is set to, but not limited to, 100bp. Up to 3.87 million short reads of human individual, which is provided by Novogene [26], are mapped against the reference sequences. Hash keys are extracted from the first half of reads, in which the mismatch threshold is set to 2. While, in end-to-end alignment, up to 5 mismatches is allowed. All alignment results that satisfy the mismatch threshold are collected without bias and filtering. Considering the diversity among various output formats, time consumed in dumping results to disk is excluded in performance evaluation. In this paper, the experiments are designed to follow the convention of the original *PerM*, where the 24 chromosomes of human genome are handled separately.

We implement and evaluate the optimized parallel *PerM* algorithm on the 8-socket SMP system, which is equipped with Intel Xeon 2 GHz 8 core X7550 CPU and 128 GB memory (easily expanded to 2 TB). All 8 cores in the same socket share an 18 MB L3 cache. The parallel program is implemented with OpenMP and compiled with the latest Intel Compiler. The memory performance is collected using the Intel VTune. For simplicity of presentation, we define several notations used in experimental results comparison:

- *HJ* represents *Hash-Join* optimization that reorders hash buckets (Section IV-B).
- *SH* stands for serialized hash, with which it indicates that the index is compressed by sequentially removing empty hash buckets (Section IV-C).
- Hash key weight is expressed as *Sn*. For example, the 13bp hash key used in original *PerM* can be denoted as

S13. Different hash key weights are combined with the proposed optimizations for comparison.

B. Results Analysis

1) Performance and Scalability: First, we report the speedup of our optimized parallel program over original *PerM*. The overall speedup is shown in Figure 7(a), where the experiments are evaluated by querying reads against the 24 chromosomes individually. Index building time is not calculated, the reads index is built once and shared by all chromosomes. The baseline program is *PerM+S13* which applies all default configurations. When we increase the hash key weight from 13bp to 15bp, only a little improvement is achieved in original *PerM*. In contrast, our optimized algorithm shows significant improved performance for each chromosome's query. Due to varied nucleotide composition of chromosomes that results in different distribution of hash buckets, the reported speedup varies in the range of 4∼11 times while 6 times in average. Note that our new algorithm can run a mapping procedure with a hash key weight of 26bp because of the applied hash index compression optimization. The original *PerM* fails to use such long hash key because its memory consumption exceeds the physical memory capacity.

Without data dependency among query reads and references, the embarrass parallel loops (Line 2 and 3 of Algorithm 2) can be easily partitioned and accelerated with multiple threads. Figure 7(b) compares the execution time of two parallel programs with an increasing number of threads. Due to the limited space we only present the results for one chromosome (*chr1*), on which similar results can be deduced for other chromosomes. With respect to parallel scalability (the ratio of parallel execution time to sequential execution time), attributed to the improved data locality in the shared cache, our optimized version is clearly better than original *PerM*. For example, with 64 threads, the parallel scalability of our optimized version is 42.4 times, while only 32.7 times is achieved in original *PerM*. However, parallel queries push the memory bandwidth to the limit, which prevents our optimized version from reaching higher speedup.

The optimization of merely increasing hash key weight reduces collision in hash index, however, as reported in Figure 8(a), the speedup is restricted to no more than 1.4 times, while the speedup of utilizing *Hash-Join* is at least 4 times. The speedup reported in Figure 8 is not some kind of upper bounds. Noticing the huge number of reads involved in real sequencing application, for experimental purpose, the limited size of reads prevents our optimizations from reaching further speedup.

Increasing hash key weight alleviates collision in hash index, however it comes at the cost of increased hash index size, which can offset benefits of using longer hash key. Illustrated in Figure 8(a), when compared with 13bp hash key, the degraded performance of using 15bp hash key indicates that the increased overhead of traversing hash index actually outweighs performance gains of using longer hash key. Regarding the steep increase in hash index size when the hash key weight get increased, the advantage of using longer hash key is decaying. The problem can be solved by hash index compression. As illustrated in Figure 8(b), the hash index iteration overhead can be reduced by hash index compression, which contributes to another 20% speedup in average.

Using serialized hash index makes it possible to build compact index structure, which not only reduces memory consumption, but also improves query speed. However, when hash key weight exceeds 16bp, instead of 32-bit integer, 64-bit long is required to store the hash key. The advantage of using long hash key can be outweighted by the increased cost of storing and comparing hash keys, which indicates that the optimal hash key weight should be 16bp. Figure 8(b) illustrates that the performance of using 26bp hash key is actually worse than 15bp hash key. However, as explained previously, with 26bp hash key, the two levels of hash indices can be combined, which facilitates the parallelization.

2) Memory Utilization: The increased memory performance, which is largely attributed to the decreased ratio of random memory access, is the key factor that accounts for the reported speedup in previous section. As listed in Table IV, utilizing *Hash-Join* and serialized hash index, the LLC miss ratio can be reduced to 8∼15% of the original *PerM*, CPI and instruction stalls also get improved by 30%.

TABLE IV
MEMORY PERFORMANCE

	CPI[1]	LLC Miss	Retire Stalls[2]
PerM+S13	.757	.542	48.5%
PerM+S15	.748	.452	48.5%
HJ+S13	.544	.046	33.5%
SH+HJ+S26	.568	.045	35.4%

[1]Cycles Per Instruction.
[2]Percentage of cycles in which no instructions are retired.

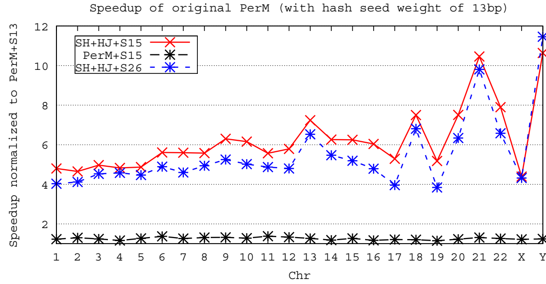
TABLE V
INDEX SIZE(IN GBYTES)

	13bp[1]	14bp	15bp	26bp
SH+HJ	26.6	30.2	34.2	40.5
PerM	18.1	36.1	108.1	-
HJ	28.7	46.7	118.8	-

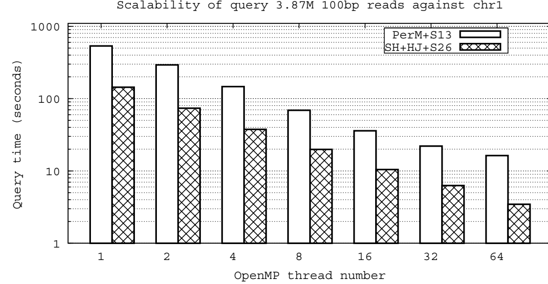
[1]Hash key weight, measured in base pair.

Along with storing the indirect offsets in reference sequences, improving data locality by extracting and storing actual sequence sub-strings in the generated index can introduce considerable index size explosion. However, in combination with hash index compression, with hash key weight of 15bp, Table V illustrates that the index size can be reduced to 30% of the uncompressed one. Further, using hash key as long as 26bp, the sensitivity of hash key can be greatly improved. However, noticing the high similarity of genome sequences, the index size is far smaller than expectation.

3) Index Building Overhead: The hash index generation is an one-time off-line pass and the generated index can be stored on disk for later reuse. Therefore, in previous

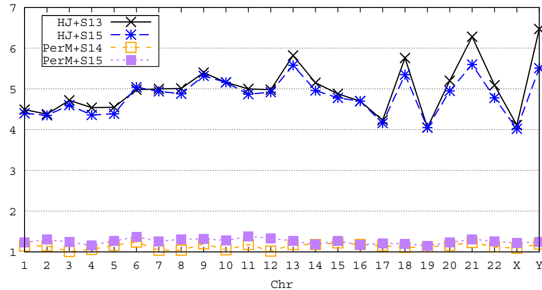


(a) Single-thread performance

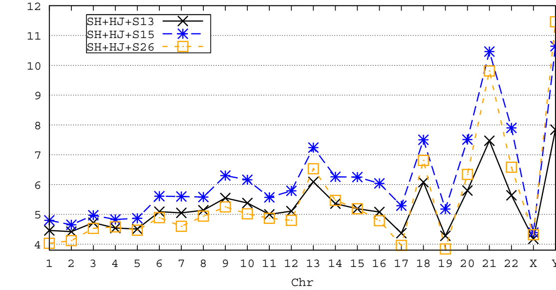


(b) Scalability comparison

Fig. 7. **Speedup and scalability:** speedup is normalized to original *PerM* with hash key weight of 13bp.



(a) Varying hash key weight and Hash-Join



(b) Varying hash key weight, Hash-Join and index compression

Fig. 8. **Speedup of individual optimization:** speedup is normalized to PerM with hash key weight of 13bp.

section, time consumed in index building is not included in query performance evaluation. As illustrated in Algorithm 1, in *PerM*, the time of building hash index is composed of 3 parts: counting items in bucket (Line 3∼8), hashing and partitioning data to buckets (Line 10∼14); and sorting items in bucket (Line 15∼16). Figure 9 explains the composition of each part. Time consumed in counting items remains stable for all optimizations. In order to improve data locality, in our optimized versions, we choose to extract and store the *seed2* data when hashing data into bucket, which accounts for the increased time reported as ***HASH*** bars in Figure 9. However, with explicitly stored *seed2* data, time consumed in bucket sorting can be greatly reduced. The net effect is that, although the composition of index building time changes, the optimizations introduced in this paper exert little impact on aggregated index building time.

The overhead of indexing reads is also depicted as ***READS*** bars in Figure 9, which occupies a little portion of the index building time. When querying reads against all 24 chromosomes, the process of indexing reads is executed only once, and the generated reads index can be shared across chromosomes. Therefore, compared with the query time, the overhead of indexing reads is negligible. Although not covered in this paper, the preprocessing step can be parallelized with trivial effort.

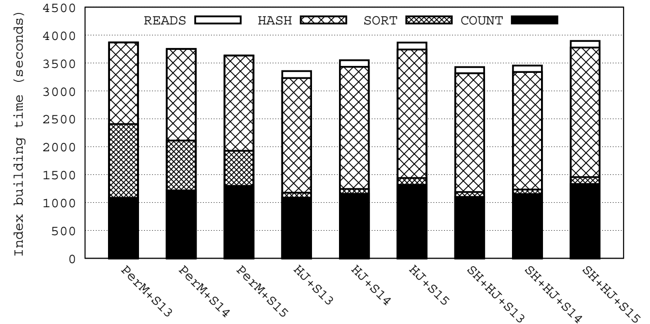


Fig. 9. Index building time. The ***READS*** bars indicate the time consumed in building reads index.

VI. Related Work

Traditional sequence alignment tools, such as BLAT [6], are designed for long sequences, and are not optimized for NGS applications with huge amount of short reads. In recent years, short sequencing reads mapping has emerged as an active research topic. Amongst the numerous work published in literature, two of the most important categories, according to the applied index structure, are hash index and FM-index (Burrows-Wheeler transform, BWT) [10]. For the first category, MAQ [11], PerM [4], GNUMAP [12], RMAP [15] are prevalent solutions, while Bowtie [13] and SOAP2 [14] are representatives in the second category. The query flow of BWT-based algorithms, which involves extensive pointer calculations, is far intricate than hash-based algorithms.

In most of the work discussed so far, improving single-thread performance and reducing memory footprint are the primary focus, which leaves parallel optimization largely ignored. The scalability for running Bowtie [13] using four threads on a 4-core server is 3.12 times. Limited by the experimental scales, it is not clear if Bowtie can be parallelized further beyond 4 threads. Most existing work on

parallel short sequencing reads mapping focus on optimal workload partitioning. In [21], six parallelization methods are proposed, including partitioning the reads, the genome, and both the reads and the genome. The parallelization strategies investigated in [24] try to optimize data distribution for parallel execution of BWT-based algorithms. In contrast, we focus on single-thread optimization in this paper, which is orthogonal to those workload partitioning strategies, in this regard their work can be used to extend our work. To our best knowledge, our work is the first approach that optimizes parallel NGS performance in aware of orchestrating data layout through memory hierarchy.

The massive parallelism involved in NGS, makes it easier to justify using distributed clusters or the GPUs for acceleration. Using MPI with multi-threading to parallelize the GNUMAP is introduced in [22], where the performance can be linearly scaled up to 256 processors across 32 nodes. The CloudBurst [23] presents efforts of parallelizing the RMAP on distributed architectures based on the Google MapReduce framework. On a 96 cores remote computing cloud, up to 100x speedup is achieved in the CloudBurst. A GPU optimized version, which uses GPU-compatible binary search instead of original hashing technique, of the RMAP is introduced in [17]. Measured on a NVIDIA Tesla C1060 GPU, in terms of overall execution time, up to 9.6-fold speedup over a sequential implantation of the RMAP on a traditional PC is reported.

VII. Conclusion

With the increased capacity and decreased cost of memory chip, utilizing large memory system to improve next-generation sequencing has just now becoming a practical consideration. We take the position that exploration of the memory hierarchy should be a foremost design consideration. We optimized and parallelized a short sequencing reads mapping program, called *PerM*, based on hash index algorithm. Our optimizations focus on memory utilization on current multi-core architecture with large mount of memory capacity. The optimization strategies include: (i) Memory access reorder in hash index by a *Hash-Join* transformation that takes advantage of cache sharing in multi-core architecture. It is worth noting that an efficient parallel algorithm on multi-core architecture should be aware of the underlying shared cache. (ii) A serialized hash index compression that reduces empty hash buckets, further enables to adopt longer hash key. The results indicate that improving performance by simply expanding hash key length is an inefficient way. More dedicated optimizations, such as *Hash-Join* and index compression, are needed to unleash the power of large memory system. The combination of these optimization strategies achieve a speedup of 4$\sim$11 times (6 times avg.) over the original parallel *PerM*.

Acknowledgment

This work is supported by the National Basic Research Program of China (NO.2012CB316502), Chinese Academy of Sciences (No.KGCX1-YW-13) and National Natural Science Foundation of China (NO.60803030, NO.60633040, NO.60925009, NO.60921002).

References

[1] Dalca AV, Brudno M, “Genome variation discovery with high-throughput sequencing data,” Brief Bioinform 2010;11:3-14.
[2] Pepke S, “Wold B, Mortazavi A. Computation for ChIP-seq and RNA-seq studies,” NatMethods 2009;6:S22-32.
[3] D. Zerbino and E. Birney. “Velvet: Algorithms for de novo short read assembly using de Bruijn graphs,” In Genome Research 18(5), pages 821-829, 2008.
[4] Chen Y, Souaiaia T, Chen T. “PerM: efficient mapping of short sequencing reads with periodic full sensitive spaced seeds,” Bioinformatics 2009;25:2514-21.
[5] Metzker ML. “Sequencing technologies-the next generation,” Nat RevGenet 2010;11:31-46.
[6] Kent, J.W., “BLAT: The BLAST-Like Alignment Tool,” Genome Res. 12(4), 656-664, 2002.
[7] David W. Ussery, Trudy M. Wassenaar and Stefano Borini, “Word Frequencies and Repeats”, Computing for Comparative Microbial Genomics Computational Biology, 2009, Volume 8, Part II, 137-150.
[8] Illumina, “HiSeq 2000 Sequencing System”, http://www.illumina.com/, June 2010.
[9] Sai P.M., Mahmut K., and Padma R., “Intra-application shared cache partitioning for multithreaded applications,” In Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP’10).
[10] Li H, Homer N, “A survey of sequence alignment algorithms for next-generation sequencing,” Brief Bioinform 2010, 11:473-83.
[11] H. Li, J. Ruan, and R. Durbin, “Mapping short DNA sequencing reads and calling variants using mapping quality scores,” Genome Research, vol. 18, no. 11, pp. 1851-1858, 2008.
[12] N. L. Clement, Q. Snell, M. J. Clement, P. C. Hollenhorst, J. Purwar, B. J. Graves, B. R. Cairns, and W. E. Johnson, “The GNUMAP algorithm: unbiased probabilistic mapping of oligonucleotides from next-generation sequencing,” Bioinformatics, vol. 26, no. 1, pp. 38-45, 2010.
[13] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg, “Ultrafast and memory-efficient alignment of short DNA sequences to the human genome,” Genome Biology, vol. 10, no.3, article R25, 2009.
[14] R. Li, C. Yu, Y. Li et al., “SOAP2: an improved ultrafast tool for short read alignment,” Bioinformatics, vol. 25, no. 15, pp.1966-1967, 2009.
[15] A. Smith, Z. Xuan, and M. Zhang, “Using Quality Scores and Longer Reads Improves Accuracy of Solexa Read Mapping,” BMC Bioinformatics, vol. 9, no. 1, p. 128, 2008.
[16] Rumble SM, Lacroute P, Dalca AV, Fiume M, Sidow A, et al. “SHRiMP: Accurate Mapping of Short Color-space Reads,”, PLoS Comput Biol 5(5), 2009.
[17] Ashwin M. Aji, Liqing Zhang, and Wu-chun Feng, “GPU-RMAP: Accelerating Short-Read Mapping on Graphics Processors,” In Proceedings of the 2010 13th IEEE International Conference on Computational Science and Engineering (CSE ’10).
[18] Alexandr Andoni, Piotr Indyk, “Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions,” Foundations of Computer Science, Annual IEEE Symposium on, pp. 459-468, 2006.
[19] David W. Williams, Jun Huan, Wei Wang, “Graph Database Indexing Using Structured Graph Decomposition,” Data Engineering, International Conference on, pp. 976-985, 2007.
[20] Changkyu Kim, Tim Kaldewey, Victor W. Lee, Eric Sedlar, and et al, “Sort vs. Hash revisited: fast join implementation on modern multi-core CPUs,” Proc. VLDB Endow., 1378-1389, August, 2009.
[21] D. Bozdag, C. C. Barbacioru, and U. V. Catalyurek, “Parallel Short Sequence Mapping for High Throughput Genome Sequencing,” in IPDPS’09: Proceedings of the IEEE International Symposium on Parallel & Distributed Processing. pp. 1-10, 2009.
[22] Nathan Clement, Mark J. Clement, Quinn Snell, W. Evan Johnson, “Parallel Mapping Approaches for GNUMAP,” in IPDPS’11: Tenth IEEE International Workshop on High Performance Computational Biology, May 2011.
[23] M. C. Schatz, “CloudBurst: Highly Sensitive Read Mapping with MapReduce,” Bioinformatics, vol. 25, no. 11, pp. 1363-1369, June 2009.
[24] Bozdag, D.; Hatem, A.; Catalyurek, U.V.; , “Exploring parallelism in short sequence mapping using Burrows-Wheeler Transform,” IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum, vol., no., pp.1-8, April 2010.
[25] http://www.ncbi.nlm.nih.gov/
[26] http://www.novogene.cn/