Genomics & Informatics

RESEARCH

Open Access

Fast and accurate short-read alignment with hybrid hash-tree data structure

Junichiro Makino[1,2], Toshikazu Ebisuzaki[3*], Ryutaro Himeno[1,4] and Yoshihide Hayashizaki[3,5]

Abstract

Rapidly increasing the amount of short-read data generated by NGSs (new-generation sequencers) calls for the development of fast and accurate read alignment programs. The programs based on the hash table (BLAST) and Burrows-Wheeler transform (bwa-mem) are used, and the latter is known to give superior performance. We here present a new algorithm, a hybrid of hash table and suffix tree, which we designed to speed up the alignment of short reads against large reference sequences such as the human genome. The total turnaround time for processing one human genome sample (read depth of 30) is just 31 min with our system while that was more than 25 h with bwa-mem/gatk. The time for the aligner alone is 28 min for our system but around 2 h for bwa-mem. Our new algorithm is 4.4 times faster than bwa-mem while achieving similar accuracy. Variant calling and other downstream analyses after the alignment can be done with open-source tools such as SAMtools and Genome Analysis Toolkit (gatk) packages, as well as our own fast variant caller, which is well parallelized and much faster than gatk.

Keywords Human whole genome analysis, Short read, Alignment (mapping), Variant calling, Hash, Tree

1 Introduction

Present-day sequencers such as Illumina NextSeq and BGI T7 can produce a full read of the human genome of around 50 persons (read depth of 30) in one day. This data corresponds to around 10TBp (base pair). This enormous amount of data requires a new level of computational power for read alignment and variant calling. The current "best practice" pipeline uses bwa-mem [7, 8] for alignment and gatk [11] for variant calling. The use of these tools on usual CPU-based servers would require a large cluster system to handle the output of a single sequencer, resulting in a significant increase in the total cost which would compromise the advantage of modern sequencers. Thus, it is of critical importance to improve the performance of read alignment and variant calling either by improving the hardware, software, or both.

There are many works to improve the performance of human genome analysis, mostly by using faster processors. Here we discuss a few recent achievements. A proprietary implementation of these tools on a machine with four NVIDIA V100 GPUs realized a speed of 175 min for human genome data with a read depth of 50 with similar accuracies with gatk [5]. This speed corresponds to 1.2 TBp/day. Thus, we can construct a system that has the performance of 10 TBp/day using $4 \times 8 = 32$ GPUs (or a somewhat smaller number with newer GPUs such as A100 or H100). Such a system would still be pretty expensive and would consume a large amount of electricity. This implementation is not a simple porting of bwa-mem and gatk but a newly

*Correspondence:
Toshikazu Ebisuzaki
ebisu@mail.jmlab.jp
[1] Advanced Accelerating Systems Co. Ltd, Deiki 1-28, B1312, Kanazawa-ku, Yokohama, Kanagawa 236-0021, Japan
[2] Department of Planetology, Graduate School of Science, Kobe University, 1-1, Rokkodai-cho, Nada-ku, Kobe 657-8051, Japan
[3] K.K. Dnaform, Ask Sanshin Building 3F, 2-6-29, Tsurumi-chuo, Tsurumi-ku, Yokohama, Kanagawa 230-0051, Japan
[4] Faculty of Health Data Science, Juntendo University, 6-8-1 Hinode, Urayasu, Chiba 279-0013, Japan
[5] Medical Technology Innovation Center, Juntendo University, 2-1-1 Hongo, Bunkyo-ku, Tokyo 113-8421, Japan

developed code highly optimized to NVIDIA GPUs. Therefore, the results of their system are not exactly the same as those of the best practice pipeline. As a result, they gave detailed discussions on the accuracy of their system.

An implementation of bwa-mem and gatk on Supercomputer Fugaku with a Fujitsu A64fx processor has been reported by Suzuki et al. [12]. The achieved performance is around 200Gbps/h, or 5TBp/day using 96 nodes of Supercomputer Fugaku. Thus, in principle, a 192-node A64fx system can process the data from one sequencer, but such a system is quite expensive and requires too much space and electricity. This implementation is a straightforward porting of bwa-mem and gatk to the A64fx processor of Supercomputer Fugaku with some modification of the source code to make use of the SVE SIMD instruction set of the A64fx processor. Thus, the calculation results are the same as those of the original bwa-mem/gatk combination, and there is no need for a detailed accuracy comparison. Since the gatk is not well parallelized, they have implemented a fairly complex scheduling algorithm in which multiple samples are processed in one batch, so that they could improve the parallel efficiency.

Illumina provides the hardware-based acceleration system, Dragen, which realizes a throughput of around 10 TBp/day. This system apparently offers the best price-performance ratio for human genome analysis. They offer both the software-only and hardware-accelerated versions of their systems.

In summary, it is certainly possible to construct computer systems that can process data from a modern sequencer in real time, but such systems are very expensive. To fully utilize the high performance of modern sequencers for the analysis of the human genome, it is necessary to significantly improve the performance of both the alignment and variant calling. The latter can be achieved by implementing the basic valiant calling algorithms in efficient parallel programs, while the former requires a fundamentally new algorithm if we are to achieve such improvement over existing best-performing software.

We describe such a new algorithm, the hybrid hash-tree algorithm. In this paper, we describe this new algorithm and compare its performance and accuracy with those of bwa-mem/gatk combination. The new algorithm achieved much better performance while retaining the accuracy comparable to that of bwa-mem/gatk.

2 Methods

2.1 Hash-based algorithm

The hybrid hash-tree algorithm is based on the original hash-based algorithm such as used in BLAST [1, 3]. A hash key is a fixed-length sequence of bases. With the hash-based algorithm, for each of all possible hash keys of a given length l, the locations in the reference sequence that match that key are recorded. For a read, we first use its first l base as a key. We store the locations of this key in the reference. Then, we shift the starting position within the read by a stride of s (typically $s = 5$), and store the locations of the new key. We repeat this procedure until we reach the end of the read.

Then we sort all candidate locations and search for the locations at which several different positions in the read match to the reference sequence. For example, if the read is a perfect copy of one sub-sequence of the reference sequence and if this sub-sequence appears in no other location, each hash of the read would appear in the corresponding location of the reference sub-sequence and there is no other place in the reference genome where all hashes of the read appear the corresponding location of the sub-sequence. Thus, we can determine the location of the sequence.

Let us consider the reference sequence of AGTCAC CAGAGATGGC with length 16 as an example. With $l = 2$, we have 15 possible starting locations of keys. If we encode ACGT as 0, 1, 2, 3, the locations and keys are as shown in Fig. 1.

Therefore, the hash table should express the data structure shown in Fig. 2. If the key is, for example, 0, it should return NULL, since there is no sequence AA in the reference.

If we have sequence AGAGA as a read, for the first AG (at position 0, when we count positions from the left and starting with zero), we find candidate locations 0, 7, and 9 as shown in Fig. 2. Here, we shift starting position by one and the key becomes GA, and get 8, 10. Since these

location	sequence	hash
0	AG	2
1	GT	11
2	TC	13
3	CA	4
4	AC	1
5	CC	5
6	CA	4
7	AG	2
8	GA	8
9	AG	2
10	GA	8
11	AT	3
12	TG	14
13	GG	10
14	GC	9

Fig. 1 Hash key values for 15 locations of reference sequence AGTCACCAGAGATGGC. First, second and third columns show the location, key value, and original string

sequence	hash	locations
AA	0	NULL
AC	1	4
AG	2	0,7, 9
AT	3	11
CA	4	3, 6
CT	5	5
CG	6	NULL
CT	7	NULL
GA	8	8, 10
GC	9	14
GG	10	13
GT	11	1
TA	12	NULL
TC	13	2
TG 14	12	
TT 15	NULL	

Fig. 2 Locations pointed by hash keys in the reference sequence of AGTCACCAGAGATGGC. "NULL" means there is no location for that key

locations correspond to position 1 in the read, to obtain locations which correspond to position 0 we should subtract one from these values to obtain 7, 9. For AG at position 2, we have 0, 7, 9 (and thus 5, 7 for the first position of the read), and 8, 10 for GA at position 3 in the read (and thus 5, 7 for the first position of the read). In this case, each key gives multiple values for the first position of the read, but position 7 appears for all four keys and it is the only value shared by all keys. Therefore, we know that AGAGA matches with the reference location 7 and the match is perfect (Appendix 1).

If there are SNPs in a read, the hash table gives different results for starting positions that cover the locations of SNPs. If there are inserts/deletions, the hash table gives different (but near) locations for starting positions before inserts/deletions and positions after. Thus, we can get some information on mutations.

This algorithm is quite robust, but the calculation cost per read can become very large. If we make, for example, hash keys of 15 bases, the number of possible values of keys is $4^{15} \simeq 10^9$, and the average number of locations per key is around three since the length of the human genome sequence is around 3×10^9. However, some keys appear in a very large number of locations, and that means such keys also appear in many reads. These frequently appearing keys cause a huge increase in the total calculation cost.

For example, if there is one hash key which appears in 10^4 places in the reference sequence, the probability that one read picks one of these 10^4 locations is $m \times 10^4/3 \times 10^9 = m/3 \times 10^5$, where m is the length of reads. Thus, if one read has the length of $m = 150$, around one in 2000 reads picks up this pattern, and its calculation cost can be 10^4 times higher than that of other reads. Of course, if there is only one such key, we could just ignore it. In practice, however, there is a spectrum of keys appearing in different numbers of locations. It is difficult to reduce the total calculation cost using the hash-based algorithm.

2.2 The suffix tree

In principle, if we could use much longer keys, we should be able to avoid this problem of too many matched locations. However, it is impractical to use the hash length longer than 15, since the amount of memory needed increases exponentially.

One solution for this problem is to use the suffix tree [13]. The suffix tree is a tree structure corresponding to the suffix array, and the suffix array is the alphabetically sorted array of all suffixes of the reference sequence. Thus, the suffix array is essentially the array of hash keys with a length the same as that of the reference sequence itself. We can regard the suffix tree as a convenient way to implement very long hash keys.

There are many different ways to make the data structure equivalent to the suffix tree [4]. Here we present a conceptually simple one for illustration purposes, which is not necessarily practical for actual reference sequence. From the reference sequence of AGTCACCAGAGA TGGC, we first make an array of sequences. The first element of the array is the reference itself, and the second element is the same reference but with the first base removed. For kth element, we remove the first $k-1$ bases, and thus we have n elements, where n is the length of the sequence Fig. 3.

Then we sort this array using the dictionary order to obtain the suffix array. The result is shown in Fig. 4a.

We can now construct a tree structure corresponding to the suffix array as shown in Fig. 4b. For one starting location of the read, we can go down the suffix tree to find the location(s) with the longest match. As a result, the number of match locations is dramatically reduced.

Though conceptually simple, the suffix tree has not been widely used for read alignment. One reason is that it requires a large amount of memory. The data size of the reference human genome sequence is around 1GB. The suffix array would need 12 GB or 24 GB (depending on whether one uses a 32-bit or 64-bit integer), and the suffix tree can easily consume more than 100 GB. Fifteen years ago, when the early versions of widely used genomics programs such as bwa-mem and gatk were designed and developed, the DRAM memory of more than 100 GB was very expensive. Moreover, machines that could house a large amount of memory were also very expensive since they had to have a large number of memory slots and thus must use expensive high-end server CPUs and very expensive motherboards.

```
AGTCACCAGAGATGGC
GTCACCAGAGATGGC
TCACCAGAGATGGC
CACCAGAGATGGC
ACCAGAGATGGC
CCAGAGATGGC
CAGAGATGGC
AGAGATGGC
GAGATGGC
AGATGGC
GATGGC
ATGGC
TGGC
GGC
GC
C
```

Fig. 3 The suffix array before sort for the reference sequence of AGTCACCAGAGATGGC

At that time, it was clearly unpractical to use the suffix tree, since there is an alternative data structure, Burrows-Wheeler Transform (BWT), which is extremely memory efficient. Thus, bwa-mem [7–9], which is currently the golden standard read aligner, adopted BWT as its basic algorithm and that was where its name, bwa-mem, came from (Burrows-Wheeler Alignment Tool, Maximal Exact Matches). To make use of computers with a small amount of memory then available, it was essential to use a memory-efficient data structure and the choice to use BWT made perfect sense.

As of 2023, desktop PC motherboards with just four memory slots can house 128GB of memory for less than 1000 USD. Thus, it might be time to rethink what is the best algorithm for the read alignment. We use the suffix tree itself instead of BWT.

The advantage of the suffix tree is that the algorithm is much simpler compared to the suffix array and BWT, and thus requires a smaller number of the main memory access. To extend the match by one base, the suffix tree algorithm needs to access just one tree node, which is usually a single instance of a class. In contrast, the suffix array and BWT require accesses to two locations in the suffix array and two more accesses to supporting data structures. Even though the calculation cost itself is not much different, the number of main memory accesses is much smaller for the suffix tree, since the data from one tree node usually fits in the one cache line, while several accesses required by BWT result in the accesses to multiple cache lines. Thus, the extension of the match with the suffix tree is much faster compared to that with BWT, on modern computers with the hierarchical cache structure.

2.3 The hybrid hash-tree algorithm

Even though the suffix tree is quite efficient, it is possible to further improve its efficiency by the following two modifications. The first one is to combine the tree search with the hash key search. We can replace the first l levels of the suffix tree with the hash key of the same length, and thus eliminate the first $l-1$ memory access. As we have stated above, $l = 15$ is practical with modern computer systems. This modification improves the search speed significantly since, for most cases, the initial hash key search reduces the candidate locations to just a few by single memory access, instead of following the tree structure 15 times. Figure 5 shows the hybrid data structure in the case of $l = 2$.

Another way to improve the efficiency of the suffix tree algorithm is to collapse multiple levels of a tree into a single level. For example, we can replace a two-level tree with four children at each level with a one-level tree with 16 children. Using this transformation, we can extend the match by two bases in one iteration, in other words, in one memory access, as far as the single node data fits into one cache line (typically 64 bytes). A tree node with 16 children can fit into 64-byte cache line, while that with 64 children does not. Therefore, instead of a suffix tree with each node corresponding to one base, we use a collapsed tree, with each node corresponding to two bases. This transformation is easy for the suffix tree, but not easy and might be impossible for the suffix array with BWT.

2.4 Search strategy

With the hash-based algorithm, such as used in BLAST, we obtain the location candidates for keys of length l in the read of length m with a stride of s. Thus, there are $(m-l-s+2)/s$ such keys. With our hybrid algorithm, we could use this same strategy, but it is obviously not ideal. When we find a rather long match, with the next starting position shifted by a stride of five or so we will certainly find a similarly long match. Also, for the majority of reads the match is either exact or containing just one SNP. Thus, the calculation cost of the hybrid algorithm can be $O(m^2)$. This is certainly not ideal.

We can avoid this problem by simply shifting the next starting point by the amount comparable to the match length itself. For example, if we shift the starting position by $p/2$, where p is the current match length, the total cost of matching calculation is reduced to $O(m)$. On the other hand, with this strategy, we can miss the longest match,

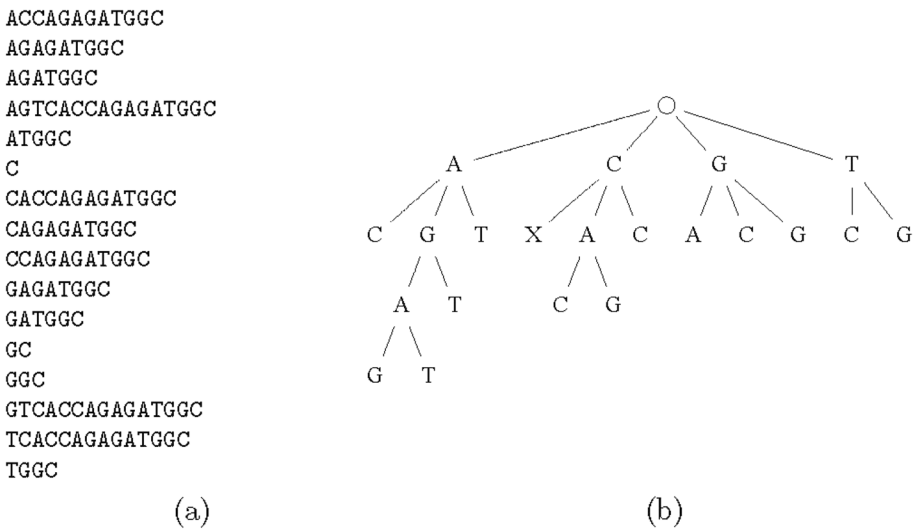

Fig. 4 The suffix array (**a**) and the corresponding suffix tree (**b**) for the reference sequence of AGTCACCAGAGATGGC. Here "X" in the tree means the end of the sequence

since the true longest match could start at the positions in the read we skipped.

A simple solution to this problem is to keep candidate locations with match lengths more than half of the apparent maximum match for the score calculation in the next stage. A longer match, if exists, starts somewhere between the current starting position and the next starting position. In the worst case, where the actual match is the shortest and appears at the leftmost position, it starts at the position next to the current starting position and extends by two bases after the end position of the current search. Therefore, if we start at the position shifted by $p/2$ from the current starting position, we can find the latter half of the longest match with length $(p+2)/2$.

For the search for chimeric alignments, our strategy can be problematic since the candidate regions for a chimeric alignment can be very similar and yet contain, for example, multiple SNPs. Actually, in this case, the longest exact match might not give the best matching location either, since the best match location might contain multiple SNPs and the length of the longest exact match location can be short. In such cases, it is necessary to make the shift length small so that we can find all short-match locations.

2.5 Extension and scoring

For the extension of the match, we use the usual Smith-Waterman-Gotoh (SWG) algorithm [6], and we take into account the Base Quality Score Recalibration procedure [2] when assigning the final scores to the matches. In our implementation, the actual code for the SWG algorithm

Fig. 5 The hybrid hash-tree tree for reference sequence of AGTCAC CAGAGATGGC

uses the AVX2 SIMD instruction set, so that we can take advantage of recent processors from both Intel and AMD.

2.6 Parallelization strategy

To make efficient use of modern CPUs, it is essential to make all steps of genome analysis well parallelized for a large number of cores, even when we just use a single desk-side workstation. This is because modern high-end processors have a large number of cores integrated into one package. For example, AMD EPYC 9000 series processors, announced in November 2022, have up to 96 cores in one package, and high-end servers can house two processor packages in one chassis, resulting in 192 cores in one computing node. It is, however, not easy to design a program whose performance scales well for more than 100 cores. Of course, large supercomputers have 1 million or more cores, and at least a few programs can make use of those huge numbers of cores. However, that is usually for extremely large-scale problems.

From the point of view of parallel processing, one advantage of human whole genome analysis is that there are lots of potential parallelism in all stages of processing. First of all, the alignment of a read can be done independently of those of all other reads. If we use the current

typical value of the read length of 150, a (pair of) fasta files for the read depth of 30 contains 100 G bases or around 300 M read pairs, and all of these 300 M read pairs can be processed in parallel.

There is, however, one practical issue. At least in the case of sample data, fasta files are usually provided as single big text files compressed with gzip. This means that it should be decompressed, and because gzip-compressed file can only be decompressed sequentially, this decompression can take time longer than the rest of the analysis. The actual sequencer should be able to generate many small fasta files for the data of one human genome since the actual reading process is highly parallel. In this paper, we assume that the input data are available as a number of small fasta files, where the total number of the fasta files is q.

Our new hybrid hash-tree algorithm requires a fairly large (around 100 GB) table to express the reference genome. Therefore, this table must be shared by processes that handle the reads in parallel. We therefore implemented the parallelization using OpenMP. In our implementation, one OpenMP-parallelized loop processes the small fasta files, one pair of files in one iteration. With OpenMP, we can specify how many threads are run in parallel at runtime. Since different threads process different files, there is very low parallelization overhead. Also, we carefully avoided any overhead caused by library calls and memory allocation/deallocation/garbage collection. Thus, as we will see in Section 3, we have achieved a very good parallel speedup. We are impressed by the fact that the Linux operating system can handle huge numbers of I/O requests quite well. This part of the program outputs candidates of matches for each read.

The SWG program, which performs the match through the Smith-Waterman-Gotoh algorithm and calculates the matching score, does not require large tables. Therefore, we implement this part as a single-thread program, which processes one file and generates SAM-format output. For parallel processing, we just run a fixed number of this program in parallel.

The output of the SWG program is then processed by the program for variant calling. The parallelism also exists for variant calling, although it is not of the level of reads, but of the level of the locations in the reference genome. Conceptually, what we do for variant calling is, for each base location in the reference genome, to see if the bases of reads aligned at its location contain SNP or other mutations. To do this, we need to be able to find all reads that cover the location we are looking at, and this is most easily done by sorting reads according to the aligned locations.

Usually sorting of reads is done using samtools. Here, again, the processing time of samtools sort can be longer than the rest of the processing, and it is important to speed up the sorting process. We have implemented parallel off-the-core bucket sort. We divide the reference locations into r regions ($r \sim 1000$ or more). The SWG program creates r output files and writes each SAM record to the file with the appropriate region. Since there are q small fasta files, we will have pq small SAM files. If we call one SAM file generated from fasta file i for reference region j as S_{ij} , we can obtain all SAM records for region j by combining all S_{ij} for $0 \leq i < q - 1$. The variant caller reads these files and sorts SAM record on the memory.

2.7 Variant calling

As we stated in the previous subsection, our variant caller accepts the SAM records in a specified range of the reference sequence and performs sorting. Then it performs so-called MarkDuplicates, and checks if there is SNP or INDEL for each location in the reference region.

2.8 Hardware and software platform

For all test calculations in this paper, we used a Linux server with AMD ThreadRipper 3990X 64-core processor, 256 GB of DDR4 main memory, and 4TB of PCIe gen3 M.2. SSDs.

The operating system is Ubuntu 20.4 LTS. We used gcc version 9.4.0 (Ubuntu 9.4.0-1ubuntu1 20.04.1) which comes with the operating system.

3 Results

3.1 Benchmark problem

As the benchmark problem, we used the data provided for PrecisionFDA Truth Challenge V2 [10]. We follow the challenge contest procedure and show the accuracy and timing results. In Appendix 2, we show the command parameters used for bwa-mem/gatk processing. We use bwa-mem/gatk for comparison since it is the current "golden standard" for genome analysis.

3.2 Accuracy

Table 1 gives the accuracy result, as was required in the PrecisionFDA Truth Challenge V2 contest. For each of the three sample data, HG002, HG003, and HG004, we present the result with bwa-mem/gatk, as well as two results with our system. One is with the high-accuracy mode, and the other with the low-accuracy, fast mode. As in the case of the original Truth Challenge V2 contest, some of the adjustable parameters for our system were tuned to give the best result for HG002, and the same set of parameters is used for HG003 and HG004. We can see that the final F-measure values of our high-accuracy results are slightly better than those of bwa-mem/gatk,

Table 1 Accuracy test results

	True positives	True calls	False-pos	False-neg	Precision	Sensitivity	F-measure
(a) HG002							
Gatk	3854404	3855258	54542	37091	0.9860	0.9905	0.9883
High	3843485	3845450	38814	48010	0.9900	0.9877	0.9888
Low	3840691	3841762	44070	50804	0.9887	0.9869	0.9878
(b) HG003							
	True positives	True calls	False-pos	False-neg	Precision	Sensitivity	F-measure
Gatk	3795148	3795094	53773	36846	0.9860	0.9904	0.9882
High	3785845	3786914	36724	46149	0.9904	0.9880	0.9892
Low	3783096	3783268	41959	48898	0.9890	0.9872	0.881
(c) HG004							
	True positives	True calls	False-pos	False-neg	Precision	Sensitivity	F-measure
Gatk	3819599	3819528	54716	37529	0.9859	0.9903	0.9881
High	3808201	3809240	39099	48927	0.9898	0.9873	0.9886
Low	3805555	3805753	43751	51573	0.9886	0.9866	0.9876

Table 2 HG002 SNP/INDEL result

	true positives	false positives	true positives calls	false negatives	precision	sensitivity	f-measure
Gatk SNP	3336055	45856	3337490	28373	0.9864	0.9916	0.9890
Gatk INDEL	518349	8686	517768	8718	0.9835	0.9835	0.9835
High SNP	3325657	29106	3329961	38771	0.9913	0.9885	0.9899
High INDEL	517828	9708	515489	9239	0.9815	0.9825	0.9820

Table 3 Time in minutes to process HG002 sample using our system

	High accuracy	Low accuracy
ass-match	55.82	28.32
ass-vcall	2.68	2.92
Total	58.50	31.23

Table 4 Time in minutes to process HG002 sample using bwa-mem/gatk

bwa-mem	124.40
MarkDuplicates	135.75
BaseRecalibrator	203.67
PrintReads	716.67
HaplotypeCaller	363.83
Total	1544.32

while those of low-accuracy results are slightly worse. The difference is, in all cases, quite small, and we can safely conclude that our system has achieved the accuracy of the same level as that of bwa-mem/gatk.

Table 2 shows the accuracy for SNPs and INDELs (non-SNPs), for the case of the HG002 sample. Our result is slightly better than that of bwa-mem/gatk for SNPs and slightly worse for INDELs. Again, the difference is very small.

Compared to other results presented in Olson et al. [10], our results (and that of bwa-mem/gatk) are of course not quite the best, even within non-deep-learning results, but are close to the average of them, since the results for Illumina data range from 97 to 99.7%.

3.3 Timing

Tables 3 and 4 show the elapsed time to process HG002 sample using our system and bwa-mem/gatk, respectively. In both cases, the time for each command and the total time are shown. For our system, times for both the high-accuracy and low-accuracy modes are shown. When we compare the total time, our low-accuracy mode is 49.5 times faster than bwa-mem/gatk, while our high-accuracy mode is 26.4 times faster. As we have stated earlier. bwa-mem uses less than 10% of the total time. If we compare the time for bwa-mem alone and that for our aas-match part, which does roughly what bwa-mem does, our low-accuracy mode is 4.4 times faster, and high-accuracy mode is 2.2 times faster than bwa-mem, respectively.

Table 5 Throughput of systems in terms of TBp/day

System	Processor	# of nodes	Performance (TBp/day)	Performance per node
This work	AMD threadripper 3990X	1	5.8	5.8
Franke and Crowgrey [5]	NVIDIA V100	4	1.2	0.3
Suzuki et al [12]	Fujitsu A64fx	96	5.0	0.05

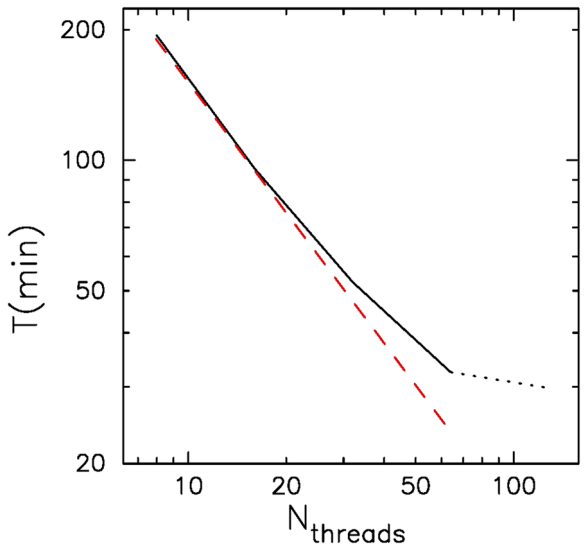

Fig. 6 Time in second T to process HG002 sample data as the function of the number of Threads, $N_{threads}$. Dashed line indidates ideal $1/N_{threads}$ scaling. For $N_{threads} > 64$, the result is shown in a dotted curve since the physical number of cores is 64

For the procedures after bwa-mem, with our system everything is done in a single command, aas-vcall, in less than 3 min. It effectively does the sorting of SAM records, MarkDuplicates, BaseRecalibration, and variant calling. Compared to gatk tools which take around 24 h in total, our system is around 500 times faster. If we compare the time for HaplotypeCaller only, our system is still 120 times faster. This difference is not due to any new algorithm, but primarily because everything is written in C language in such a way that there is no serious parallelization overhead.

Table 5 shows the throughput of systems in terms of TBp/day, for the present work, NVIDIA V100 [5] and Fujitsu A64fx [12]. We can see that our system is by far more cost-effective compared to systems reported in recent works. With our software, a single-processor workstation achieves a throughput comparable to (but faster than) those of 16 NVIDIA V100 GPGPUs or 96 Fujitsu A64fx processors.

3.4 Parallel efficiency

Figure 6 shows the time to process HG002 sample data as the function of the number of threads used. The dashed line indicates the ideal parallel speedup. We can see that processing speed is almost proportional to the number of threads, for up to 64 threads. For more than 64 threads, the gain is small since we used 64 cores.

The main reason why the speedup is somewhat less than ideal for a large number of cores is because of the limitation of the clock frequency. On AMD ThreadRipper 3990X processor, when only a small number of cores are active, their clock frequency can reach 4.3 GHz. However, when a large number of cores are used, the clock frequency goes down due to the limit in the power consumption. We found that the clock frequency was 3.9 GHz for 32 threads and 3.3 GHz for 64 and 126 threads. This reduced clock frequency explains why the parallel speedup is not ideal.

4 Discussions

In this paper, we present the algorithm, implementation, and performance of our fast aligner for short reads. Our system is two to four times faster than bwa-mem on the same computer system. The accuracy achieved for datasets used in Precision FDA Truth Challenge V2 is very close to that of bwa-mem/gatk for our low-accuracy mode (F-measure difference of 0.01–0.05%), and slightly more accurate for our high-accuracy mode.

We also developed a new implementation of the variant caller, which is used to obtain the above result. It can process Truth Challenge V2 samples in less than 3 min on a 64-core AMD ThreadRipper processor. Our variant caller is more than 100 times faster than gatk Haplotype-Caller.

Right now, our system has been tested only for germline mutations and is not well-tested for somatic mutations or structural variants. We are currently working on the efficient detection of structural variants. Here, bwa-mem does consume a significant fraction of the total processing time. Even so, the total speedup factor is rather limited, since variant callers for structural variants consume time comparable to that consumed by bwa-mem. Thus, to improve the total performance, it is necessary to improve the performance of the variant caller here as well.

There is still much room for performance improvement in our new aligner, and we hope to report on improving performance in the near future.

Appendix 1

Overview of the construction steps for the hybrid data structure

We construct the hybrid tree data structure in the following steps:

1. Make a single combined string of the genome, from chromosome data, so that one number can be used to specify both the chromosome and the location within that chromosome.
2. Make an array of 64-bit integers, in which each entry corresponds to one location in the combined genome string. This array corresponds to "locations" of Fig. 1.
3. Sort that array with the dictionary order of the genome string starting from the locations, with the maximum comparison length specified as "l" in the main text. This l is the length of the hash. In Fig. 1, this sorting is done using the "hash" values with $l = 2$. The sorted result is shown in Fig. 7.
4. Now we can count how many locations are assigned to each hash value, by comparing the hash values assigned to the consecutive entries in the sorted array (in Fig. 7), and assigning the locations with the same hash value to the hash value itself. The result of this counting is shown in Fig. 2. In the actual code, we can make this data structure by making an array of hash values and assigning the first location of that value in the sorted array to each of the hash values.
5. Finally, we need to construct the additional suffix-like tree structure for each hash value with two (or some given value) or more entries. This can be done through one of the usual algorithms to construct the suffix tree.

4	AC	1
0	AG	2
7	AG	2
9	AG	2
11	AT	3
3	CA	4
6	CA	4
5	CC	5
8	GA	8
10	GA	8
14	GC	9
13	GG	10
1	GT	11
2	TC	13
12	TG	14

Fig. 7 Index array after sort by hash values location hash value

Appendix 2

Command Scripts

Command Scripts for our system

```
> aas-preprocess \
       -q $DATADIR/HG003.novaseq.pcr-free.35x.R1.fastq.gz\
       -Q $DATADIR/HG003.novaseq.pcr-free.35x.R2.fastq.gz
> aas-match  -A -t 124
> aas-vcall -t 64
> gunzip test1.vcf.gz
> rtg bgzip test1.vcf
> rtg index test1.vcf.gz
> rtg vcfeval  --baseline  \
 $DATADIR/HG003_GRCh38_1_22_v4.2.1_benchmark-withchr.vcf.gz\
  --calls test1.vcf.gz --output rtg-results-HG003 \
  --template $DATADIR/GRCh38.sdf \
  --bed-region\
 $DATADIR/HG003_GRCh38_1_22_v4.2.1_benchmark_noinconsistent.bed
```

Listing 1: Commands for our system

Command Scripts for bwa-mem/gatk

```
> bwa mem  -t 60 -Y \
 $DATADIR/GRCh38/Homo_sapiens.GRCh38.dna.chromosome.giaborder.fa  \
 $DATADIR/HG002/HG002.novaseq.pcr-free.35x.R1.fastq.gz \
 $DATADIR/HG002/HG002.novaseq.pcr-free.35x.R2.fastq.gz  \
 | time samtools view -buhS \
   -t $DATADIR/GRCh38/Homo_sapiens.GRCh38.dna.chromosome.giaborder.fa.
      fai - \
   | time samtools sort -@ 60 -m 1000000000 \
   -T ./PE100.sort -o ./PE100.1.sorted.bam -
> time java -jar $PICARDDIR/picard.jar MarkDuplicates \
 I=./PE100.1.sorted.bam \
 O=./PE100.2.rmdup.bam \
 M=./PE100.2.rmdup.mat \
 REMOVE_DUPLICATES=false \
 TMP_DIR=./tmp
> java -jar $GATKDIR/GenomeAnalysisTK.jar \
 -T BaseRecalibrator \
 --disable_auto_index_creation_and_locking_when_reading_rods \
 -nct 62 \
 -R  $DATADIR/GRCh38/Homo_sapiens.GRCh38.dna.chromosome.giaborder.fa
     \
 -I ./PE100.2.rmdup.bam \
 -o ./PE100.4.bqsr.grp \
 -knownSites $DATADIR/dbsnp/All_20180418.vcf
> java -jar $GATKDIR/GenomeAnalysisTK.jar \
   --disable_auto_index_creation_and_locking_when_reading_rods \
   -T PrintReads \
   -nct 40 \
   -R  $DATADIR/GRCh38/Homo_sapiens.GRCh38.dna.chromosome.giaborder.fa
       \
   -I ./PE100.2.rmdup.bam \
   -BQSR ./PE100.4.bqsr.grp \
   -o ./PE100.4.bqsr.bam

> java -jar $GATKDIR/GenomeAnalysisTK.jar \
   --disable_auto_index_creation_and_locking_when_reading_rods \
   -R $DATADIR/GRCh38/Homo_sapiens.GRCh38.dna.chromosome.giaborder.fa\
   -T HaplotypeCaller -nct 40  \
   -I ./PE100.4.bqsr.bam --dbsnp $DATADIR/dbsnp/All_20180418.vcf \
   -stand_call_conf 30.0 --pcr_indel_model NONE \
   -variant_index_type LINEAR \
   -variant_index_parameter 128000 -o ./raw.snp_indel.vcf.gz
```

Listing 2: Commands for bwa-mem/gatk

Acknowledgements

This work was done under the research contract between Advanced Accelerating Systems Co. Ltd. and K. K. Dnaform.

Authors' contributions

Conceptualization: JM, TE, RH, and YH. Data curation: JM. Formal analysis: JM. Funding acquisition: YH and RH. Methodology: JM. Writing—original draft: JM. Writing—review and editing: JM and TE. All authors read and approved the final manuscript.

Declarations

Ethics approval and consent to participate

Not applicable.

Competing interests
The authors declare that they have no conflicts of interest.

Accepted: 10 June 2024
Published online: 29 October 2024

References

1. Altschul SF, Gish W, Miller W, Myers EW, Lipman DJ. Basic local alignment search tool. J Mol Biol. 1990;215(3):403–10 ISSN 0022-2836.
2. Caetano-Anolles D. Base quality score recalibration (bqsr). 2023.
3. Camacho C, Coulouris G, Avagyan V, Ma N, Papadopoulos J, Bealer K, Madden TL. Blast+: architecture and applications. BMC Bioinformatics. 2009;10(1):421 (ISSN 1471-2105).
4. Farach M. Optimal suffix tree construction with large alphabets. In: Proceedings 38th Annual Symposium on Foundations of Computer Science. 1997. p. 137–43.
5. Franke KR, Crowgey EL. Accelerating next generation sequencing data analysis: an evaluation of optimized best practices for genome analysis toolkit algorithms. Genomics Inform. 2020;18(1):e10.
6. Gotoh O. An improved algorithm for matching biological sequences. J Mol Biol. 1982;162(3):705–8 (ISSN 0022-2836).
7. Li H. Exploring single-sample SNP and INDEL calling with whole-genome de novo assembly. Bioinformatics. 2012;28(14):1838–44 (ISSN 1367-4803).
8. H. Li. Aligning sequence reads, clone sequences and assembly contigs with bwa-mem, 2013. URL https://arxiv.org/abs/1303.3997.
9. Li H, Durbin R. Fast and accurate short read alignment with Burrows-Wheeler transform. Bioinformatics. 2009;25(14):1754–60 (ISSN 1367-4803).
10. Olson ND, Wagner J, McDaniel J, Stephens SH, Westreich ST, Prasanna AG, Johanson E, Boja E, Maier EJ, Serang O, J´aspez D, Lorenzo-Salazar JM, A. Mu˜nozBarrera JM, Rubio-Rodr´iguez LA, Flores C, Kyriakidis K, Malousi A, Shafin K, Pesout T, Jain M, Paten B, Chang PC, Kolesnikov A, Nattestad M, Baid G, Goel S, Yang H, Carroll A, Eveleigh R, Bourgey M, Bourque G, Li G, Ma C, Tang L, Du Y, Zhang S, Morata J, Tonda R, Parra G, Trotta JR, Brueffer C, Demirkaya-Budak S, Kabakci-Zorlu D, Turgut D¨, OzemKalay, Budak G, Narcı K, Arslan E, Brown R, Johnson IJ, Dolgoborodov A, Semenyuk V, Jain A, Tetikol HS, Jain V, Ruehle M, Lajoie B, Roddey C, Catreux S, Mehio R, Ahsan MU, Liu Q, Wang K, Ebrahim Sahraeian SM, Fang LT, Mohiyuddin M, Hung C, Jain C, Feng H, Li Z, Chen L, Sedlazeck FJ and Zook JM. Precisionfda truth challenge v2: Calling variants from short and long reads in difficult-to-map regions. Cell Genomics. 2022 2(5):100129.
11. Poplin R, Ruano-Rubio V, DePristo MA, Fennell TJ, Carneiro MO, Van der Auwera GA, Kling DE, Gauthier LD, Levy-Moonshine A, Roazen D, Shakir K, Thibault J, Chandran S, Whelan C, Lek M, Gabriel S, Daly MJ, Neale B, MacArthur DG and Banks E. Scaling accurate genetic variant discovery to tens of thousands of samples. bioRxiv. 2018. https://doi.org/10.1101/201178.
12. Suzuki S, Ito S, K. Sakai S, Inada Y, Miyoshi I, Ishikawa H, and Miyano S. Optimization and performance evaluation of whole-genome analysys program genomon for super computer fugaku (in Japanese). Technical Report 18, RIKEN R-CCS, 2021.
13. Weiner P. Linear pattern matching algorithms. In: 14th Annual Symposium on Switching and Automata Theory (swat 1973). 1973. p. 1–11.

Publisher's Note