Chapter 2
JVM: Java Visual Mapping Tool for Next Generation Sequencing Read

Ye Yang and Juan Liu

Abstract We developed a program JVM (Java Visual Mapping) for mapping next generation sequencing read to reference sequence. The program is implemented in Java and is designed to deal with millions of short read generated by sequence alignment using the Illumina sequencing technology. It employs seed index strategy and octal encoding operations for sequence alignments. JVM is useful for DNA-Seq, RNA-Seq when dealing with single-end resequencing. JVM is a desktop application, which supports reads capacity from 1 MB to 10 GB.

2.1 Introduction

Over the past 5 years, tens of read mapping programs were published to copy with Illumina sequencing data. But there are some problems have to be pointed out. The first is the limitation of the operating system (OS). Most of programs is designed by C++ language and only can be used on Unix/Linux OS. The biologist is boring by using Unix/Linux OS. Based on a survey on OS user, more than 90 % of users are used to apply "Windows" OS; almost 7 % of users are willing to use "Mac" OS provided by Apple Inc. So the program meeting the need of Multi-OS is required. The second is the restriction of the memory usage. Traditional sequence alignment softwares like MAQ [1], BWA [2], SOAP [3] are high memory consumption programs, and it is difficult to run these programs on the laptop normally. Therefore

Y. Yang · J. Liu (✉)
School of Computer, Wuhan University, Wuhan 430072, Hubei, China
e-mail: liujuanjp@163.com; liujuan@whu.edu.cn

Y. Yang
Military Economy Academy, Wuhan, Hubei, China

the program with low memory consumption is needed. The last is the confusion of the parameter settings. There are so many parameters in most of the existed tools that it is difficult for a user to know how to set parameters to finish the alignment. In this work we present a new program JVM (Java Visual Mapping), trying to address to above three problems.

JVM is a desktop application program implemented with Java language, by which the user only needs mouse actions to fulfill the alignment. The best hit of each read which has zero number of sequence mismatch or gap will be reported. The read has multiple hits will be reported in the final list. JVM can handle reads around 11–1000 bp long, and can deal with single-end reads of FASTQ format files which produced by Illumina sequencing platform. JVM supports file sizes ranging from 1 MB to 10 GB. In order to run the program successfully, a Java Runtime Environment version 6.0 or later is required.

2.2 Problem Statement

Read document is a FASTQ format file with four lines per sequence. Line 1 begins with a '@' character and is followed by a sequence identifier and an optional description (like a FASTA title line). Line 2 is the raw sequence letters. Line 3 begins with a '+' character and is optionally followed by the same sequence identifier (and any description) again. Line 4 encodes the quality values for the sequence in Line 2, and must contain the same number of symbols as letters in Line 2 [4].

The genome sequence document is a series of characters, each character is either a nucleic acid represented as A, G, C, or T, or an unknown character, named N [5]. This document contains the genome chromosome information.

Read alignment (mapping) is the course of sequence mapping. JVM takes read query sequences with equal length and a database of reference genome sequence as input. Read alignment is just to locate the right places where reads have a perfect alignment to reference genomes. JVM finds all valid alignment that satisfied the constraint on zero error in the set of query sequence.

2.3 Preprocessing

To addresses the problem of too much of the memory spending, we adopt the following preprocessing strategies in JVM.

2.3.1 File Block

The size of a human reference genome document has around 2–3 GB. A read document generated from Illumina platform has a size from 2 to 4 GB. JVM first intelligently separate the read document into several parts, separate the reference document base on the information of chromosome name, and write each part into the disk. Then it maps each read block to a specified chromosome. The Fig. 2.1 illustrates the mapping strategy of choosing read and reference part.

Through this way, we can reduce the peak memory consumption when reading the large capacity files.

2.3.2 Octal Encoding and Sequences Compressing

Problem 1: octal encoding

JVM uses five octal digits to represent each base in read and reference document. The symbols A, C, G, T and N are encoded as 0, 1, 2, 3, 4 respectively. Take a string 'GGGANAACAT' as an example, this string is encoded as octal string: $(2220400103)_8$ (see Table 2.1).
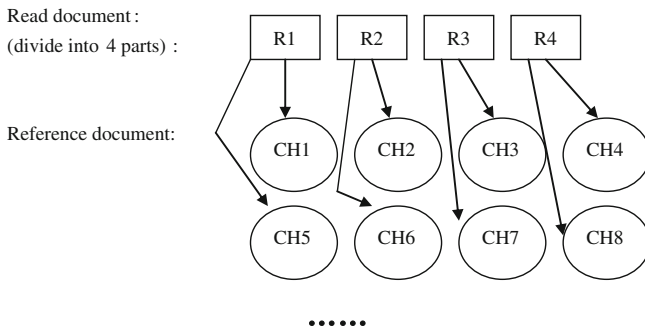

Read document:
(divide into 4 parts):

Reference document:

Fig. 2.1 A simulated image of the mapping strategy

Table 2.1 An example of octal encoding and string compressing

String S	GGGANAACAT
Octal encode	2220400103
Compressed value	306315331

Problem 2: sequences compressing

We denote a reference genome sequence as R = R(1, 2,…, m), the query read sequences as the set Q(q1, q2, …, qn), where m is the total bases of reference genome, n is the number of short read; we also let each sequence length be L(10<L<1,000). Alignment progress is the problem of mapping Q to R.

Reference compressing: We construct a new string P = P(1, 2,…, m) by using octal encoding. We partition the r into a set of factor F1, F2, …Fm − L + 1, where Fi = r(i, i + 1,…, i + L − 1), for 0 < i <=m − L + 1. In view of the range of integer type, we transform every 10 octal encoded number into a decimal value and store this numeric into an integer array. In this way, we can reduce memory cost five times when load a long sequences set into main memory. Table 2.1 give an example of the encoding and compressing progress.

Read compressing: Similarly, we can deal with the read document in the same way. We abstract the base sequences from the read and then encoding and compressing the base sequences into an integer array. We define the read array set as Q(1, 2,…n).

2.4 Method

JVM is a visualization tool that by using the mouse operation to complete read mapping and then create a file of ".SAM" format as the output result. In order to accelerate alignment, we take various measures to speed up the efficiency of JVM.

In the section of preprocessing, both reads and the reference sequences are converted to numeric data type using octal encoding for each base. We set the numeric reference as $F(i)(0 < i <= m$, and $i > 108)$, and numeric read as $Q(j)(0 < j <= n$, and $j > 107)$. As the progress of read alignment, we use the non-recursion quick sort algorithm combine with seed index strategy to complete ascending sort of $F(i)$. Then using the seed index to quickly position the $Q(j)$ to $F(i)$. At the end of this section, we would give the time complexity of JVM.

The steps of our method are as follows.

Step 1.

Save the reference and read documents into memory. We get a numeric reference set $F(i)$ and store each $F(i)$ into an integer array Ai[t], for $t = \lceil L/10 \rceil$. Then we add Ai[t] to a list T. In this way, reference document is compressed into list T, and reduces the size of reference document for five times. We use the same strategy to copy with the read document Q(n).

Step 2.

Build the seed index. We extract every Ai[0] (the first element of Ai[]) from list T. We call Ai[0] as seed index. Then we load Ai[0] into a hash index array B[m − L + 1]. The number of element in B[m − L + 1] almost equal with the number of bases in reference document. In other words, B[m − L + 1] has an big order of magnitude. Considering the memory overflow, we use non-recursive quick sort algorithm to sort the B[m − L + 1]. At the same time, we save the original position of every F(i) so as to keep the exact location of F(i). Non-recursive quick sort algorithm is based on the divide-and-conquer strategy and takes O(m log m) time to sort B[m − L + 1].

Step 3.

Index search and read alignment. We get the read sequence element array from Q(n). To find the best hit of Q(j)(0 <=j < n), we get the first element of each array that is Q j (0), then map it to reference array set B[m − L + 1]. To improve the ability of searching speed, we search the B[m − L + 1] base on the divide-and-conquer strategy. And this take O(log m) time to find the best hit.

For three steps, in step 1 can be done in O(mL + nL) time. Step 2 should be done in O(m log m) time. Step 3 runs in O(n log m). So the overall time complexity is O(m log m + n log m).

2.5 Result

2.5.1 Test by Simulated Dataset

To evaluate speed and accuracy of JVM, we compared JVM with MAPNEXT [6] and WHAM [7]. We had mapped a simulated dataset of 246,558 49 bp-long Illumina single-end resequencing reads. Our reference genome is a dataset simulated the structure of the zebra fish genome NCBI Zv9. To guarantee a fair comparison, we ran the three programs on a same virtual machine and set the mismatch parameter as 0. The OS of this machine is Linux CentOS.5.4. The configuration of this machine includes 2G of main memory, dual 2.00 GHz AMD Turion 64 2-core CPUs. We also test JVM on the Windows OS with the Java Virtual Machine memory with 1.6 G. Table 2.2 shows the performance of each program.

As the result in Table 2.2, JVM has the similar performance on both Linux and Windows OS. Although WHAM is much faster and has more numbers of read mapped to reference, it needs to write parameters and build an index on the reference genome. In addition, WHAM ignores the memory limitation from personal computer during the mapping progress. JVM has a better performance than MAPNEXT on both speed and mapping rate.

Table 2.2 Mapping 246,558 49 bp-long simulated reads to simulated the structure of the zebra fish genome NCBI Zv9

Program	Total time (s)	Read aligned
JVM (on Linux)	47	224
WHAM (on Linux)	0.57	953
MAPNEXT (on Linux)	53	184
JVM (on Windows)	54	224

2.5.2 Test by Real Datasets

We evaluate three programs on a computer with dual 2.00 GHz AMD Turion 64 2-core CPUs, and 4G of DDR2 main memory, running Linux OS. We choose two real datasets containing 20,099,013 and 17,680,937 Illumina single-end rese-quencing reads (length 49 bp), which were generated from mRNA-Seq of zebra fish. We call the two datasets as dataset 1 and 2. Two read files are two different growth stages of zebra fish. Concerning the time consumption and feature of JVM, we get the same part from dataset 1 and mapping to the zebra fish chromosome 25, the result of three programs show in Table 2.3. In the same way, we take out part of dataset 2 and mapping to the chromosome 22. It gives the performance of each program. We finally run JVM on Windows 7 OS with the same computer configuration in the previous tests.

In the article of WHAM, the author claimed and verified that WHAM was a very fast alignment method. It is often orders of magnitude faster than BOWTIE [8] and RBSA [9]. From the results shown in Table 2.3, total time consumption of WHAM is much less than JVM and MAPNEXT. That is, we also confirmed its conclusion by our experiment. Although JVM is not as fast as WHAM, JVM has a better performance on mapping number. JVM has mapped nearly 20 % more reads than WHAM.

JVM has great advantage over MAPNEXT on time consumption. JVM finished alignment in 235.670 s, while MAPNEXT done in 480.000 s. In terms of mapped reads, MAPNEXT has only 1001 reads mapping to chromosome 25, but JVM has found 37009 reads, it is dozens of times to MAPNEXT.

Table 2.3 Mapping 20,099,013 49 bp-long real reads to the zebra fish chromosome 25 (38,499,472 bp)

Program	Total time (s)	Read aligned
JVM (on Linux)	235.670	37009
WHAM (on Linux)	25.685	31571
MAPNEXT (on Linux)	480.000	1001
JVM (on Windows)	243.890	37009

Table 2.4 Mapping 17,680,937 49 bp-long real reads to the zebra fish chromosome 22 (42,261,000 bp)

Program	Total time (s)	Read aligned
JVM (on Linux)	246.32	53369
WHAM (on Linux)	28.213	44627
MAPNEXT (on Linux)	520.000	1407
JVM (on Windows)	254.762	53369

We also run JVM on Windows 7 OS, we get the same result as that on the Linux OS.

In order to valid the effectiveness of the results in Table 2.3, we not only adjust the read and reference document, but also reset parameters on indexing and mapping progress of WHAM and MAPNEXT. As it is indicated by Table 2.4, the same result can be concluded.

2.5.3 Conclusion and Discussion

As it is demonstrated by above analysis, our developed JVM does a better overall performance than MAPNEXT. And JVM can find more hit reads than WHAM. Based on the efficiency and sensitivity on alignment, we believe that further development and functionality research is not only necessary but also feasible.

We have to admit that JVM is still in the process of improving. As a feature of JVM different from other software, file block is the first target which should be deal with. Now that the order of read part document aligned to reference chromosome document is defined, as the example showing in Fig. 2.1, we can take parallel processing method to accelerate the alignment speed. We can regulate the number of parallel threats based on the total number of physical cores in test machine. So application of parallelization processing mechanism in JVM is the next work we should to do.

In addition, alignment is just the first step in analysing and processing the next generation sequencing data. Further researches based on JVM such as gene fusion and gene expression profiles will be launched.

References

1. Li H, Ruan J, Durbin R (2008) Mapping short DNA sequencing reads and calling variants using mapping quality scores. Genome Res 18:1851–1858
2. Li H, Durbin R (2009) Fast and accurate short read alignment with Burrows-Wheeler transform. Bioinformatics 25:1754–1760
3. Li R, Li Y, Kristiansen K, Wang J (2008) SOAP: short oligonucleotide alignment program. Bioinformatics 24:713–715

4. Cock P, Fields C, Goto N et al (2010) The Sanger FASTQ file format for sequences with quality scores and the Solexa/Illumina FASTQ variants. Nucl Acids Res 38:1767–1771
5. Frousios K, Iliopoulos CS, Mouchard L, Pissis SP, Tischler G (2010) REAL: An efficient REad ALigner for next generation sequencing reads. ACM, New York, pp 154–159
6. Bao H, Xiong Y, Guo H et al (2009) MapNext: a software tool for spliced and unspliced alignments and SNP detection of short sequence reads. BMC Genomics 10:S13
7. Li Y, Terrell A, Patel J (2011) WHAM: a high-throughput sequence alignment method. SIGMOD Conf 11:445–456
8. Langmead B, Trapnell C, Pop M, Salzberg SL (2009) Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. Genome Biol 10(3):R25
9. Papapetrou P, Athitsos V, Kollios G, Gunopulos D (2009) Reference-based alignment in large sequence databases. PVLDB 2(1):205–216