Optimization and Application extension for a Bloom filter based sequence classifier

Enze Liu

Degree project in
Computer Science
Second cycle
Stockholm, Sweden 2013
Optimization and Application extension for a Bloom filter based sequence classifier

Computer Science and Communication department

Royal Institute of Technology, Stockholm

Enze Liu

Supervisor Dr. Lars Arvestad

Examiner Prof. Jens Lagergren
Abstract

Nowadays, with the development of sequencing technologies, more sequencing reads are generated and involved in genomics research, which leads to a critical problem, how do people process these data rapidly and accurately? A data structure named Bloom filter which is initially developed in 1970 has been reused and applied more and more in Bioinformatics field for its relatively high storage efficiency and fast accessing speed. As an application of Bloom filter technique, FACS [1] system is a rapid and accurate sequence classifier. However, several bottlenecks have restricted its usage, for instance, neither supporting large query file nor fastq format files. Hence, in this report, an improved FACS system will be introduced, which includes a hashing system for FACS; making FACS become large query files (>2GB) and compressed files supported; making FACS become fastq file supported; making FACS system more user friendly etc. Moreover, the new paralleled FACS system (FACS 2.0) will be introduced and evaluated to prove that FACS 2.0 is at least 10 times faster and equally accurate compared with the original FACS system, Fastq_screen [7] and Deconseq [8] when doing sequence decontamination process. Last but not the least, the possibility of developing an adapter trimmer based on FACS system will also be analyzed in this report.

Key words: Bloom filter; Decontamination; Adapter trimming; Parallelization; Large query file (compressed and normal) supported;
## Content

1. Introduction ........................................................................................................... 1
   1.1 Goals .................................................................................................................. 2
      1.1.1 Speedup by threading ................................................................................. 2
      1.1.2 Making FACS practical to use ................................................................. 2
      1.1.3 Contamination assessment ....................................................................... 2
      1.1.4 Soft trimming ............................................................................................. 3
2. Background and Related work ......................................................................... 4
   2.1 Background ...................................................................................................... 4
      2.1.1 Contaminated sequence .......................................................................... 4
      2.1.2 Bloom filter ................................................................................................. 4
   2.2 Related work .................................................................................................... 6
      2.2.1 Bloom filter based Sequence Classifier ................................................ 6
      2.2.2 Bloom filter based de Bruijn graph ............................................................ 7
      2.2.3 Fastq_screen ............................................................................................. 7
      2.2.4 Deconseq ................................................................................................... 7
      2.2.5 Parallel computing ..................................................................................... 8
3. FACS optimization ............................................................................................. 9
   3.1 Main structure optimization ........................................................................... 9
   3.2 New Bloom filter library design ................................................................. 10
      3.2.1 New Hashing process design ................................................................... 10
      3.2.2 Main Architecture redesign ...................................................................... 11
      3.2.3 Seed generation optimization ................................................................. 11
      3.2.4 Element checking optimization ............................................................... 12
      3.2.5 Hash functions selection ........................................................................ 13
   3.3 Adaptive k-mer and match cutoff number ................................................. 14
   3.4 Performance Comparison with FACS prototype and optimized FACS ........ 15
4. Sequence Decontamination Investigation ..................................................... 17
   4.1 Architecture .................................................................................................... 17
      4.1.1 Quick Contamination Assessment process ........................................... 18
   4.2 New Scoring System ...................................................................................... 18
      4.2.1 System formulation ................................................................................... 18
      4.2.2 Consecutive Hits Bonus .......................................................................... 19
   4.3 Contamination probability value ............................................................... 23
      4.3.1 Basic concept ............................................................................................. 23
      4.3.2 P-value design .......................................................................................... 24
      4.3.3 p estimation in N (np, np(1-p)) ............................................................... 25
5. Parallelization for FACS system ...................................................................... 26
   5.1 Large file support ........................................................................................... 26
      5.1.1 Standard C library for big file support .................................................. 26
      5.1.2 File mapping ............................................................................................... 27
      5.1.3 Chunk mapping with File mapping ....................................................... 27
   5.2 Large compressed file .................................................................................... 27
5.3 Parallelization Area Investigation ................................................................. 28
5.4 MPI and OpenMP ........................................................................................... 29
5.5 SMP level parallel .......................................................................................... 29
  5.5.1 Main structure .......................................................................................... 29
  5.5.2 Mutual exclusion for multiple threads ....................................................... 31
5.6 Hybrid parallel design for clusters level parallel ........................................... 32
  5.6.1 Main structure .......................................................................................... 32
  5.6.1 Nodes synchronization ............................................................................. 33
5.7 Performance comparison among FACS_SC, FACS_SMP and FACS_hybrid.... 34
6. Overall Evaluation ............................................................................................. 36
  6.1 Introduction .................................................................................................... 36
  6.2 Experiment Result and Analysis ................................................................... 37
7. Adapter trimmer ................................................................................................ 41
  7.1 Architecture .................................................................................................. 41
  7.2 Performance evaluation for Prototype .......................................................... 42
    7.2.1 Synthetic query dataset ......................................................................... 42
    7.2.1 Performance analysis ............................................................................. 44
8. Conclusion and Future work ........................................................................... 46
  8.1 Conclusion ...................................................................................................... 46
  8.2 Future work .................................................................................................... 47
    8.2.1 True sequence classifier .......................................................................... 47
References ............................................................................................................... 48
Appendix A ............................................................................................................ 51
Appendix B ............................................................................................................ 53
Acknowledgement ................................................................................................. 55
List of Figures and Tables

Figure 2.1 Bloom filter example. ................................................................. 4
Figure 3.1 Process Flow chart for optimized FACS system .............................. 9
Figure 3.2 (a) k-mer Hashing process of PDbloom with 4 hash functions.(b) k-mer Hashing process of Newbloom with 4 hash functions.......................................................... 12
Table 3.3 Performance comparison between Power and Bit shift........................ 13
Figure 3.4 (a) Unique iteration number generated by different hash functions (b) Time cost by different hash functions........................................................................ 14
Table 3.5 Recommended values generated by adaptive value generating system.................. 15
Table 3.6 Testing parameters of comparing FACS 1.0 and 2.0 ............................... 16
Table 3.7 Time cost Comparison with FACS1.0 and FACS2.0 for building a bloom filter...... 16
Table 3.8 Time cost Comparison with FACS1.0 and FACS2.0 for using a bloom filter to check contamination................................................................. 16
Figure 4.1 Flow chart of FACS sequence decontamination system....................... 17
Table 4.2 Example of read scoring with old scoring system and new scoring system ........ 20
Figure 4.3 Performance Comparison of different scoring system with E.coli K-12 reference ...... 21
Figure 4.4 Performance Comparison of different scoring system with Human chromosome 8 reference...................................................................................................................... 22
Figure 4.5 Typical area representing p-value ........................................................ 24
Table 4.6 Hits with different reference filters using random generated dataset............... 25
Figure 5.1 Main flow chart of FACS decontamination program .............................. 28
Table 5.2 (a) Pseudo code of SMP based paralleled decontamination process (b) Pseudo code of multi-thread decontamination process...................................................... 30
Figure 5.3 Data flows in SMP based parallel decontamination process ................... 31
Table 5.4 Pseudo code of hybrid paralleled decontamination process...................... 32
Figure 5.5 Data flows in hybrid parallel decontamination process ............................. 33
Table 5.6 Testing environment and parameters of parallel comparison test ................. 34
Table 5.7 Performance Comparison with FACS_SC, FACS_SMP and FACS_hybrid............... 35
Table 6.1 Parameters of Comparison test with Deconseq, Fastq_screen and FACS 2.0........ 35
Table 6.2 Comparison experiment result of Deconseq, Fastq_screen and FACS2.0............ 37
Table 6.3 Comparison experiment result of Deconseq, Fastq_screen and FACS2.0............ 38
Table 6.4 Average time for handling single read with Deconseq, Fastq_screen and FACS 2.0.... 38
Table 6.5 Comparison of main features for Deconseq, Fastq_screen and FACS2.0............... 39
Figure 7.1 An example of adapters in sequencing process ........................................ 41
Figure 7.2 Flow chart of FACS adapter trimming system ......................................... 42
Figure 7.3 Average true SOLiD read length after removing all the adapters with E15 mouse dataset sequenced by Scilifelab................................................................. 43
Figure 7.4 (a) Performance of FACS based adapter trimmer without noise in query dataset..... 44
(b) Performance of FACS based adapter trimmer with random noise in query dataset....... 44
Figure 8.1 Family tree of FACS 2.0 system ...................................................... 47
Abbreviations

**FACS**: Fast and accurate sequence classification system

**PDbloom**: Bloom filter designed by Peter Alvaro and Dmitriy Ryaboy

**Newbloom**: Bloom filter designed by me; a variant from PDbloom

**k-mer**: a specific n-tuple or n-gram of nucleic acid or amino acid sequences that can be used to identify certain regions within biomolecules

**CPV**: Contamination probability value, measuring the probability that the sample has been contaminated

**ROC-curve**: Receiver operating characteristic (ROC), is a graphical plot which illustrates the performance of a binary classifier system as its discrimination threshold is varied.

**FDR**: false discovery rate

**SMP**: Symmetric multiprocessing. Usually refer to a parallel process on single CPU
1. Introduction

Nowadays, in genomic research, as an important application, sequence classification has been widely used, for instance, classifying DNA, RNA or protein sequences to the existing references can help people to learn their relevant function.

So far there are several sequence classifiers for DNA classification, for example, FACS system [1], which is originally designed for classifying reads in the query file to specific reference genome. According to the previous experiment result, by using a Bloom filter technique, along with k-mer based checking and evaluating approach, the FACS system becomes at least three times faster and more accurate than BLAT [9] and SSAHA2 [11] aligner especially with query data set larger than 50MegaBytes.

However, currently FACS algorithm has several significant drawbacks:

1. FACS cannot build a filter for a large reference genome due to a limitation of the original library that FACS had chosen. Any filter exceeding a size of 312MegaBytes cannot be successfully created, which narrows the reference genome size down to around 200MegaBytes.

2. Since the library of FACS is initially designed for a more general purpose than just sequence classification, it has not been optimized in many aspects, including its hash function, seeds generating for hash function, addressing method for hash function, and file I/O methods. Any of these could be the bottleneck for FACS.

3. FACS doesn’t take advantage of parallel processing techniques. Only one core in one CPU is actually running the FACS program at one time, which indicates that, if more than one core or even more than one CPU can be involved when running FACS program, then the speed would be significantly increased with parallelization and the efficiency would be dramatically improved.

4. Moreover, the Bloom filter based approach has shown the excellent capability and potential to be applied to more biological aspects where speed and storage space are critical. Hence, FACS system is supposed to be deeply developed and used more than for sequence classification.
In view of the reasons mentioned above, several potential research goals have been set up, including implementing threading for FACS; redesigning the Bloom filter core for FACS to make it practical to use; exploring the application possibility for sequence decontamination and adapter trimming.

1.1 Goals

1.1.1 Speedup by threading

The C implementation of FACS system has been shown to be significantly faster than the prototype FACS system. However, it would be interesting to see what kind of speedup would be possible by writing a threaded version of the software to make use of multicore CPU’s.

1.1.2 Making FACS practical to use

One basic objective for making FACS catch on is that it is well suited for most users. This is probably not the case due to several issues. For example, FACS currently reads FASTA-formatted sequences. This is not optimal in many applications today. When using FACS system, there are several parameters that need to be set which requires some understanding of how FACS works. Some straightforward default options with descriptive names should be developed to help end-users. Moreover, providing adaptive parameters and saving it along with Bloom filter is also necessary.

1.1.3 Contamination assessment

A common need in a genomics facility is to determine whether a sample has been contaminated or not. People do not want to send sequence data from bacteria along for analysis if they are working with e.g. a human sample, and vice versa, it is necessary to establish that a sample does not contain remains from the human that ran the sequencing.

There are approaches for doing this kind of filtering process, for example, Fastq_screen [7] has been developed to assess contamination. Although FACS can solve the task of removing
contamination, one application is to simply return a number between 0 and 1 that represents the contamination. The number could mean the fraction of reads that are contaminated, or it could be a number assessing the risk that there is contamination in the sample. Detail of this needs to be worked out in collaboration with end-users.

Input: A sequence library and a list of filters for suspected contaminants.
Output: A number in [0, 1] summarizing contamination level.

1.1.4 Soft trimming

In many genomics applications, DNA fragments are labeled with an added short DNA snippet before sequencing. This label may, for example, indicate from which batch or individual a DNA fragment comes from and it is excised after sequencing and before actual analysis. In the ideal case, the label is easy to remove because it is known already, but there are several occasions when the label becomes an obstacle.

1. The labels are often incompletely sequenced and people have to adapt the extent, which they remove by making a call on how much of the label was actually sequenced.

2. There might be read errors in the label.

3. The data is generated by external scientists and downloaded from a public repository. Surprisingly often, there are no indications of what the labels are and whom to contact about it.

The idea with this FACS mode is to guess a suitable trimming (5' or 3' on the read) based on what parts of a read map according to a FACS filter.

The input will be a set of reads and a filter, and the output is the same set of reads but with 5' or 3' ends removed according to suitable criteria which needs to be developed.
2. Background and Related work

2.1 Background

2.1.1 Contaminated sequence

According to the definition from NCBI VecScreen library [26], a contaminated sequence is one that does not faithfully represent the genetic information from the biological source organism/organelle because it contains one or more sequence segments of foreign origin. It can be from cloning vector, for instance, E.coli plasmid; or adapters, linkers or PCR primers; even from insertion sequences. If any of these cases happened, alien sequences can be cloned, and later be sequenced, which is definitely going to cause a lot of problems such as: higher error rate; pollution of public database; time and resources waste.

2.1.2 Bloom filter

Bloom filter is conceived in 1970, by Burton Bloom [2]. Generally said, a Bloom filter contains a bit vector \( m = \{1, \ldots, m\} \), where each element represents a bit, and a series of independent hash functions \( h_1, \ldots, h_k \). To insert a piece of data \( x \) into it, a series of elements \( h_1(x), \ldots, h_k(x) \) are set to one. To check whether a piece of data \( y \) is stored in the bloom filter or not, elements

![Figure 2.1 Bloom filter example. x, y, z means elements that have been put into the filter. W means element that doesn’t belong to the bloom filter.](image-url)
$h_1(y), \ldots, h_k(y)$ in $[m]$ needed to be checked. And this piece of data belongs to the filter only if all $h_1(y), \ldots, h_k(y)$ elements in the filter return 1.

The typical features of a Bloom filter are:

1. Great space efficiency. Unlike hash tables, a bloom filter doesn’t record the position info for a piece of data, instead, it only provides info of whether this data belongs to the filter or not. Hence, the space cost is much less compared to hash table.

2. No false negative rate. Once an element is in the filter, hits will always be shown when query filter with the same element.

3. False positive changes according to the number of elements stored in the filter. The more elements are there in the filter, the higher false positive rate the filter gets.

**Error Rate**

For a Bloom filter, the false positive rate of an element comes from all the bits that represent an element are by chance all set to 1.

The probability of k bits that represent an element return 1 is:

$$1 - \left(1 - \frac{1}{m}\right)^k$$  \hspace{1cm} (1)

Where m is the size of the bit vector, k is the number of hash functions.

The probability to make an element become 1 after n insertions becomes:

$$1 - \left(1 - \frac{1}{m}\right)^{kn}$$  \hspace{1cm} (2)

Where n is the number of elements to be inserted.

Therefore, the probability of having an element checking error is:

$$P = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k = (1 - e^{-kn/m})^k$$  \hspace{1cm} (3)

For a given $m$ and $n$, the value of $k$ that can minimize the value of $P$ is:

$$k = \frac{m}{n} \ln 2$$  \hspace{1cm} (4)

Iterate equation (4) into (3), we can get:

$$m = -\frac{n \ln P}{(\ln 2)^2}$$  \hspace{1cm} (5)

So we get the conclusion that $m$ only depends on the number of elements $n$ and the error rate $P$, which is important to us. If user provides error rate and the number of elements, then the size of Bloom filter and the optimal number of hash function $k$ can be calculated.
2.2 Related work

2.2.1 Bloom filter based Sequence Classifier

Stranneheim et al [1] has shown that the Bloom filter can be used for reads classification. Reads can be classified to the reference genome by examine if enough k-mers in the reads belong to the reference Bloom filter. Here are the main processes of their k-mer based classifier:

1. **Bloom filter building**
   a. **Initialization**
      Firstly, word size k and error rate will be taken from the user. Otherwise default values will be used. Number of elements can be roughly estimated by the size of reference. The size of Bloom filter and optimal hash number will then be calculated by using the formulas (5) and (3) respectively. Furthermore, a bit vector, which is actually simulated by a string, will be allocated in RAM.
   b. **Element insertion**
      After initializing an empty filter, a reference file will be read. For each read in the file, each time a sub sequence with word size k will be selected by using sliding window approach. Iteratively, each k-mer will be added to the filter.
   c. **Bloom filter saving**
      When all k-mers are inserted into the filter, the filter will then be saved as a binary file along with info such as number of elements and filter capacity.

2. **Sequence Matching**
   a. **Reference filter loading**
      The filter containing reference genome info will be loaded into RAM, stored as a Bloom filter.
   b. **Match cutoff**
      Match cutoff is a threshold used for distinguish whether a read is classified or not. In FACS system, match cutoff is calculated by the k-mer hits info (number of hits, position of hits, etc.).
   c. **Filtering and evaluating**
      Afterwards, the query file will be loaded into RAM as well. Then for each read in the file, query every k-mer in it against the reference Bloom filter to see whether the k-mer occurs in the filter or
not. The whole read will be then evaluated according to the number of hits (k-mer occurs in the filter) and the density of continuous hits, in order to be decided to be classified to the reference genome or not.

d. Result saving

Finally, all the reads will be saved into two files, one containing all classified reads and the other containing all the unclassified reads. RAM will be swept.

2.2.2 Bloom filter based de Bruijn graph

Jason Pell et al [3] have shown us another usage of Bloom filter in bioinformatics field. By using Bloom filter to save de Bruijn graph, RAM space requirement has been dramatically reduced while false positive rate is still maintained in an acceptable level. Moreover, they have provided an interface which allows Velvet [4] and Abyss [5] users to their Bloom filter based de Bruijn graph for assembling purpose.

2.2.3 Fastq_screen

Fastq_screen [7] is a Bowtie [6] based, simple but effective tool which can be used to search a query dataset against a set of reference to let user find out the originate of the query data. Based on the alignment scores provided by Bowtie, Fastq_screen can distinguish the detected reads which has a more match score to the reference genome than the threshold. Because of this feature, it can be used to quickly assess if there is any potential contamination within the dataset.

2.2.4 Deconseq

Deconseq [8] is a framework used for sequence decontamination, which takes alignment result from Bwa-sw [9], [10] sequence aligner, which is a Burrows-Wheeler based sequence aligner with Smith-Waterman approach.

The main process works like this: bwa-sw traverses the query prefix directed acyclic word graph and only keeps the best scoring nodes according to their Z-best strategy. And Deconseq, which is written in Perl, will take over the alignment info from bwa-sw and distinguish the possible
contaminated reads from the clean reads, saving the result into files.

2.2.5 Parallel computing

In the simplest sense, parallel computing is the simultaneous use of multiple compute resources (e.g. multiple CPU) to solve a computational problem. [27] By using parallel technique, more resources can be used to execute a program; hence, the computing time can be shortened. There are mainly three memory architectures for applying parallel process: shared memory, distributed memory and hybrid memory. For shared memory, it is relatively easy to apply a parallel process since all threads will share the same piece of memory, such as parallel process for multi-core CPU. For distributed memory, if a parallel process needs to be applied, then every thread needs to be executed independently. And message will be sent among threads for task synchronization. For a hybrid memory system, for instance, a cluster, there can be many ways of applying parallel process, and each will cost different resources and provide different performance.
3. FACS optimization

3.1 Main structure optimization

The current FACS system consists of two scripts: ‘Bloombuild’ and ‘facs’, which are two executable programs written in Perl. Both will call the same library, Libbloom when executed. To integrate them into one, I have redesigned the main structure, as the figure 3.1 shown below.

Figure 3.1 Process Flow chart for optimized FACS system

We can see that two individual processes: bloom building and sequence classification will be called from one main process, which is more convenient both for using and developing. And in this case, the command of using it looks like:
Rather than the previous:

```
./Bloombuild --b E.coli_k-12.fasta --k 21 --e 0.0005
./facs --b E.coli_k-12.bloom --q test.fastq --t 0.8
```

### 3.2 New Bloom filter library design

In order to build a Bloom filter for some large genome, for instance, Homo sapiens (Human) genome or Mus musculus genome, the filter has to be large enough to store all possible k-mers in the genome, which leads to the following features that need to be achieved.

1. Huge capacity. Reference genomes or query files larger than 4GB should be supported.
2. Low false positive rate. The lower the better.
3. Adaptive parameters. The program should automatically support the best k-mer and match cutoff for users.

#### 3.2.1 New Hashing process design

For storing one k-mer data into filter, several bits are needed depending on the error rate required. Meanwhile, for generating each bit, one hash function is needed. But in that case, there are two potential drawbacks:

1. Multiple hash functions are needed. For instance, in our case, to achieve error rate lower than 0.0005, 10 independent hash functions will be needed. It is not wise to hardcode such number of hash functions in the code.
2. Different hash functions have different false positive rate. Thus, if we use multiple types of hash functions, an overall error rate for the program cannot be precisely measured.

In reality, for convenient coding and equal hashing quality reason, a wise way to do that is to use only one function along with a large number of prepared seeds. Each time, only one hash function needs to be loaded, and the hash function can generate different numbers with different seeds even for the same string.
3.2.2 Main Architecture redesign

The Bloom filter that I have built is a variant based on the initial library Libbloom, which is developed by Peter Alvaro and Dmitriy Ryaboy and has been initially used by FACS1.0. There are several changes/improvements that I have done for our Bloom filter from Ryaboy and Alvaro’s, which are described as follows: (From here we use ‘PDbloom’ and ‘Newbloom’ to represent Peter and Dmitriy’s implementation and my implementation respectively).

3.2.3 Seed generation optimization

Both PDbloom and Newbloom uses only one hash function. But from figure 3.2 (a) and (b), the clear difference of hashing mechanism can be seen. Figure 3.2 (a) represents the way how PDbloom hashes a string while figure 3.2 (b) shows how Newbloom’s hashing process works. To make each k-mer hashing sub process unique, PDbloom uses seeds as an extra tail which will be added to the k-mer string in order to make the string become unique, hence, the hashing result becomes unique as well. But this approach slows the process down due to the time cost for adding seed into the k-mer string. Instead, in Newbloom, for each k-mer hashing process sub process, the hash function will take both the string and one seed as input and generate a hash number. In this case, no extra time is needed for merging string and seed.
3.2.4 Element checking optimization

In C/C++ program, since there is no data structure that can be used to directly represent bit type, people usually use char array to replace bit type. And each element in a char array has a length of one byte, equivalent to 8 bits.

Here is one of the critical lines in PDBloom:

\[
\text{element} = (\text{hash\_key} / 8); \quad (1)
\]

Where element is a char type array.

Here is another in PDBloom:

\[
\text{bit} = \text{pow}(2, (\text{hash\_key} \% 8)); \quad (2)
\]

These calculations are for setting the specific bit to 1.

Usually, without any optimization process, multiplication, division and power calculations cost at least ten times of clock cycles more than ‘Bit shift’ or ‘Bitwise AND’.

Potentially, multiplication, division and power calculations can be optimized. However, according to my testing result, GCC compiler has mechanism to optimize multiplication and division because when I replace (1) with \texttt{hash\_key} \ggg 3 and (2) with \texttt{hash\_key} \& 7, the performance is almost the same.
From the pseudo code, we can see that one power process is involved. Since this critical code fragments will be called several times once a k-mer is going to be hashed, replacing power with other efficient calculation could significantly increase the speed, which is, in our case, replace power with Bit shift, which makes the code fragment become:

\[
element = (\text{hash\_value} / 8); \quad (3)
\]

\[
\text{bit} = 1<<(\text{hash\_value} \mod 8); \quad (4)
\]

Here is result from a simple test measuring the difference of using bit shift and power.

<table>
<thead>
<tr>
<th></th>
<th>Total time for building E.coli K-12 filter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power</td>
<td>13.4s</td>
</tr>
<tr>
<td>Bit shift</td>
<td>8.8s</td>
</tr>
</tbody>
</table>

Table 3.3 Performance comparison between Power and Bit shift

Building a bloom filter for E.coli K-12 reference, with two different ways mentioned above, we can see that after optimizing power calculation with Bit shifting, the performance has been significantly improved.

### 3.2.5 Hash functions selection

A good hash function can provide a fast speed while maintaining a relatively low false positive rate. So choosing a fast and accurate hash function becomes critical. Hence, eight different kinds of hash functions are selected for a series of tests: Lookup8 [20], Jenkins [14], Cityhash [21], RShash [17], APhash [16], FNVhash [22], DEKhash [19] and SDBMhash [18]. Most of them are widely used worldwide for different purposes.

Among all 8 hash functions, ‘Lookup8’ and ‘Jenkins’ are hash functions developed by Bob Jenkins. Moreover, Jenkins function has been applied in PDbloom. Cityhash is developed and applied by Google corp. The other 5 are commonly used in all kinds of application.

In our experiment, E.coli K-12 genome is used as reference. Three word sizes k: 15, 18, 21, which appeared in our tool most frequently, have been chosen. The number of different types of k-mer is counted because the bigger this number is, the less false positive rate this hash function generates.

Figure 3.4 (a) illustrates this number with different word size k. Figure 3.4 (b) shows the relevant time cost for each hash function.
From these two graphs, we can see that although Lookup8 needs about 25% more time than the other hash functions, it also provides the smallest false positive rate, which is more important than computational time in scientific computing. Hence, Lookup8 has become our first choice of hash function.

Figure 3.4 (a) Unique iteration number generated by different hash functions (b) Time cost by different hash functions

### 3.3 Adaptive k-mer and match cutoff number

In FACS system, k-mer represents the word size k and match cutoff is a threshold of judging whether a read belongs to the reference genome or not. FACS1.0 doesn’t support adaptive word size k and match cutoff number. Only one set of default settings exits in the program. But for
different size of reference genomes, different word size k and match cutoff number should be chosen in order to achieve a best sensitivity and specificity. So in my implementation, I have built a function that generates the best word size k and match cutoff number. Moreover, word size k and match cutoff number will be stored along with the Bloom filter so that when the Bloom filter is loaded; these values will be updated automatically, which makes the program become more user-friendly. This cannot be found in FACS prototype. In fact, to use the FACS prototype, user has to manually provide the correct k-mer value every single time when doing a sequence alignment process.

The adaptive number system is based on empirical results that we have got during the developing and testing stage. The results are shown in table 3.5:

<table>
<thead>
<tr>
<th>Genome size</th>
<th>k_mer</th>
<th>Match cut off</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;1MB</td>
<td>15</td>
<td>0.4</td>
</tr>
<tr>
<td>1~10MB</td>
<td>15</td>
<td>0.3</td>
</tr>
<tr>
<td>10~20MB</td>
<td>16</td>
<td>0.4</td>
</tr>
<tr>
<td>20~50MB</td>
<td>17</td>
<td>0.3</td>
</tr>
<tr>
<td>50~100MB</td>
<td>18</td>
<td>0.3</td>
</tr>
<tr>
<td>100MB~500MB</td>
<td>19</td>
<td>0.4</td>
</tr>
<tr>
<td>&gt;500MB</td>
<td>20</td>
<td>0.3</td>
</tr>
</tbody>
</table>

Table 3.5 Recommended values generated by adaptive value generating system

3.4 Performance Comparison with FACS prototype and optimized FACS

So far, all the optimization I have done for the FACS prototype is mainly about increasing the processing speed. To verify whether the speed has been significantly raised or not, a test is set with both FACS prototype and optimized FACS. It can be described as follows:

For the convenient reason, from now on, FACS prototype will be short as FACS1.0, respectively, optimized FACS as FACS2.0
Testing environment:
Kalkyl cluster in UPPMAX

Competing Programs:
FACS1.0; FACS 2.0

Parameters:
k-mer 16, match cutoff 0.8 for E.coli K-12
k-mer 21, match cutoff 0.8 for Human chromosome 8

Reference Genome:
E.coli K-12
Human chromosome 8

Query dataset:
Illumina E.coli K-12 sequences 2,622,382 reads
Synthetic 454 dataset 10,000 reads

Table 3.6 Testing parameters of comparing FACS 1.0 and 2.0
Since both FACS1.0 and FACS2.0 are currently using the same scoring system, sensitivity and specificity are not counted in this test.

<table>
<thead>
<tr>
<th>Bloom build</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>E.coli K-12</td>
<td>Hum Chr 8</td>
</tr>
<tr>
<td>FACS1.0</td>
<td>26.668 s</td>
<td>959.249 s</td>
</tr>
<tr>
<td>FACS2.0</td>
<td>6.863 s</td>
<td>279.674 s</td>
</tr>
</tbody>
</table>

Table 3.7 Time cost Comparison with FACS1.0 and FACS2.0 for building a bloom filter

<table>
<thead>
<tr>
<th>Bloom check</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>Synthetic</td>
<td>Real illumine</td>
</tr>
<tr>
<td>FACS1.0</td>
<td>21.354s</td>
<td>N/A</td>
</tr>
<tr>
<td>FACS2.0</td>
<td>9.103 s</td>
<td>6 s</td>
</tr>
</tbody>
</table>

Table 3.8 Time cost Comparison with FACS1.0 and FACS2.0 for using a bloom filter to check contamination
N/A areas indicate another drawback of FACS1.0, which is it doesn’t support fastq formatted file.

From table 3.7 and 3.8, we can clearly see that after optimization, FACS2.0 is significantly faster than FACS1.0, no only when building a Bloom filter, but when detecting contamination using a bloom filter as well. From the result that has been shown above, a conclusion can be drawn, which is, the optimization work has made the FACS system become more efficient.
4. Sequence Decontamination Investigation

From the test result in chapter three, we know that the performance of FACS2.0 has been improved, including two critical features: speed and accuracy, which shows the possibility of applying FACS system for sequence decontamination. In order to do that, a new scoring system is needed for evaluating contamination level as well as a probability value for measuring the possibility of reads being contaminated.

4.1 Architecture

As shown in Figure 4.1, the main structure contains three sub main functions. The bloom build
and contamination removal processes are pretty much the same as the two main processes of sequence classifying.

**4.1.1 Quick Contamination Assessment process**

Query files coming directly from sequencing machine usually have a huge size and not all of them are potentially contaminated. To avoid doing unnecessary decontamination process, a quick contamination assessment process is developed, which has been shown as the middle process in Figure 4.1. Instead of checking every reads in the query data, it checks a certain proportion of all reads uniformly distributed in the query file. User can adjust the proportion value in order to balance the confidence and speed level. In addition, after the quick assessment is done, a ratio value between 0 and 1 will be returned as the contamination proportion, along with a probability score, measuring the probability of contamination.

**4.2 New Scoring System**

To measure whether a read is contaminated or not, a scoring system is needed for our k-mer approach. In this section, a new scoring system will be introduced. Meanwhile, as comparison, a series of tests has been done for both the new scoring system and the scoring system that has been used in FACS prototype.

**4.2.1 System formulation**

The initial idea of developing new scoring system is to develop a scoring system which supports a shorter k-mer to increase the sensitivity and specificity compared with Stranneheim’s [1] original scoring system. It can be described as follows:

Suppose in a read, a k-mer belongs to the reference filter, which can also be called as ‘a hit’, then in our program, the number of all bases in this k-mers will be counted. If this k-mer has an overlap area with other k-mer in this read, then the number of all bases in the overlap area will be counted as well.

In formula:
\[ AWS = k \times Nr_{Hit} \]

AWS means Accumulated weighted Score; \( k \) means the size of k-mer; \( Nr_{Hit} \) means Number of k-mer that hit the filter.

\[ MWS = kL - 2\left(\sum_{i=1}^{k-1} i\right) \]

MWS means Maximum weighted Score that could possibly occur in one read; \( L \) records read length.

\[ Missed\_Scores = k \times Nr_{NoHits} \]

\( Nr_{NoHits} \) Means the number of k-mers that don’t hit the filter.

And the normalized score can be calculated by:

\[ NormalizedScore = \frac{AWS}{MWS} \]

NormalizedScore is the threshold number for distinguishing whether a read is contaminated or not.

The goal of developing new scoring system is to keep the NormalizedScore stable while using a lower number of k-mer.

The goal is to make the NormalizedScore have higher value for correct read and lower value for alien read.

Finally, for every read in the query data, a match cutoff number will be calculated for judging contamination.

### 4.2.2 Consecutive Hits Bonus

This process is basically the same as the previous one except giving continuous hits extra bonus.

The idea of doing this is based on the fact that introducing C.H.B (Consecutive Hits Bonus) can effectively reduce the impact of random hits.

Consecutive Hits, short as C.H, is a concept for measuring continuous hits in one read. For instance, if we use sliding window and examine every k-mer from the beginning of the reads, then if two hits of k-mer happened in a row, then it will be count as one C.H.

With C.H.B process:

A.W.S formula is the same as above.

\[ CH = Nr_{Consecutive\_Hits} \]
$CH$ measures the number of consecutive hits.

And the $MWS$ becomes:

$$MWS = (k + 1) \cdot (L - 1) - \left( \sum_{i=1}^{k-1} i \right) \cdot 2 + 1$$

And Normalized Score becomes:

$$Score = \frac{AWS + CH}{MWS}$$

Here we present a typical example to compare the new scoring system and the previous that has been used in FACS1.0

Assume:

Three reads with the length of 105 are sequenced from E.coli K-12. One contains no error, one contains 5 mismatches spread out in the read, and one contains 5 mismatches gathered at the end of the read.

<table>
<thead>
<tr>
<th></th>
<th>old AWS</th>
<th>All</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read1</td>
<td>173</td>
<td>105</td>
<td>1.65</td>
</tr>
<tr>
<td>Read2</td>
<td>50</td>
<td>105</td>
<td>0.48</td>
</tr>
<tr>
<td>Read3</td>
<td>142</td>
<td>105</td>
<td>1.35</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>new AWS</th>
<th>All</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read1</td>
<td>1545</td>
<td>1545</td>
<td>1</td>
</tr>
<tr>
<td>Read2</td>
<td>181</td>
<td>1545</td>
<td>0.12</td>
</tr>
<tr>
<td>Read3</td>
<td>1239</td>
<td>1545</td>
<td>0.8</td>
</tr>
</tbody>
</table>

Table 4.2 Example of read scoring with old scoring system and new scoring system

From table 4.2, we can see that although both two types of scoring system give more score to the reads with more continuous hits, the new scoring system generates more significant difference. Normalized scores with Read2 and Read3, which indicates that the new scoring system can distinguish the difference of an alien read and a homolog read with mismatches much better, because usually, an alien read will have mismatches more spread within the read.

In order to see the performance in reality, a series of tests have been executed with different types of scoring system, including the old one, the new one and the new one with CHB. Here are the results:

Reference genome: E.coli K-12; Human chromosome 8

Query dataset: a fasta formatted synthetic query data used for performance investigation when FACS1.0 was evaluated, containing 1,794 E.coli reads and 55,137 Human chromosome 8 reads out of 100,000 reads in total, along with average read length of 269 and random 454 sequencing.
Figure 4.3 Performance Comparison of different scoring system with E.coli K-12 reference (a1) ROC graph for E.coli using Old scoring system (b1) FDR graph for E.coli using Old scoring system (a2) ROC graph for E.coli using New scoring system (b2) FDR graph for E.coli using New scoring system (a3) ROC graph for E.coli using New scoring system and CHB (b3) FDR graph for E.coli using New scoring system and CHB

The left parts are ROC graphs, where Y axis represents the true positive rate; X axis represents the false positive rate. The right parts are FDP (False discovery proportion), where Y axis represents the false positive rate, X axis represents the match cutoff value. By comparing the left part, we can clearly see that the false positive number rate of using new scoring system is significantly smaller than that of using old scoring system. And this number can be further reduced after applying CHB.
policy. Moreover, after applying new scoring system and CHB policy, we can use smaller word size k numbers and still maintain a very low FP rate, which can be drawn by comparing the right part. And by using smaller word size k, the sensitivity can be improved further, which can be seen from appendix A and B.

Figure 4.4 Performance Comparison of different scoring system with Human chromosome 8 reference (a1) ROC graph for human chromosome 8 using Old scoring system (b1) FDR graph for human chromosome 8 using Old scoring system (a2) ROC graph for human chromosome 8 using New scoring system (b2) FDR graph for human chromosome 8 using New scoring system (a3) ROC graph for human chromosome 8 using New scoring system and CHB (b3) FDR graph for human chromosome 8 using New scoring system and CHB

This set of graph 4.4 basically shows the same phenomenon as mentioned above. Additionally, from the right part, an optimal match cutoff area can be identified from the new scoring and CHB
graphs, which cannot be found from result of using Old scoring system.

In general, we can draw a conclusion that although the optimal case for both old scoring system and the new scoring system is pretty much the same, in most of the times, FP rate is sharply decreased, even with smaller word size k. A smaller word size k usually means a higher sensitivity. And the less the FP rate is, the higher the specificity becomes. So in another word, with new scoring system and CHB policy, both sensitivity and specificity are dramatically raised.

Additionally, from Figure 4.4 b2, we can find ‘unusual’ decrease which doesn’t follow the trend of all the other lines. When k-mer and match cutoff are respectively set to 21 and 1, only 267 reads are captured and 243 among them are true positive. This phenomenon happened due to the small sampling size, which could easily cause a sampling bias.

4.3 Contamination probability value

Under some exceptional circumstances, reads can be incorrectly identified as contaminated due to random k-mer hits coincidence. To measure the probability of this phenomenon, a contamination probability value, short as CPV, is established to tell the user how likely the contamination has happened in the query data.

4.3.1 Basic concept

The general idea of creating CPV is to use purely random generated reads to query against reference filters to see the chance of reads being accidentally captured as contaminated. Then take the random captured reads number as a standard. If a query data has a smaller captured reads number than standard ones, then it is very likely that the query data is clean, vice versa. But one main problem occurred during the investigation:

Due to the high specificity of our scoring system, random generated reads can hardly be identified. According to my test, no read is captured by our reference filters out of 2,000,000 reads in total, which indicates that it is not suitable to use random captured reads for calculating the relevant probability.

Counting hits number

Instead of counting random contaminated reads, recording random k-mer hits would be a more
rational choice as its bigger chance of happening and uniformly spread among the query data.

4.3.2 P-value design

For a k-mer querying against a bloom filter, only two results can be occurred: hit or unhit, and assume the probability of getting a hit is \( p \), then, for an unhit, it is \( 1 - p \). Hence, if we query a query dataset which contains a number of k-mers against the bloom filter, since each k-mer is independent, the result should follow a binominal distribution \( B(n, p) \) where \( n \) is the total number of k-mers within this dataset; \( p \) is the probability of getting one hit for one k-mer), which indicates for this query dataset, the probability of having \( k \) hits to a filter is:

\[
C_n^k p^k (1 - p)^{n-k}
\]

Thus, the probability of having no more than \( k \) hits to a filter becomes:

\[
\sum_{i=0}^{k} C_n^i p^i (1 - p)^{n-i}
\]

And the result of formula (2) becomes a p-value.

If we assume that the \( B(n, p) \) follows the curve in figure 4.5, then formula (2), as well as p-value represents the green area in this figure.

![Figure 4.5 Typical area representing p-value](image)

Since the \( n \) is usually large in our case, a reasonable approximation to \( B(n, p) \) can be described as \( N(np, np(1-p)) \), which is a normal distribution. Then by using cumulative distribution function, short as CDF, the estimated p-value can be calculated.
4.3.2 $p$ estimation in $N (np, np(1-p))$

To use cumulative function to calculate the $p$-value based on $N (np, np(1-p))$, $n$ and $p$ need to be estimated. $n$ is the total number of k-mers within a query dataset, which can be easily calculated.

For $p$, I estimate it in the following way:

<table>
<thead>
<tr>
<th></th>
<th>k-12</th>
<th>k-15</th>
<th>k-18</th>
<th>k-21</th>
</tr>
</thead>
<tbody>
<tr>
<td>Human chr 8</td>
<td>71071726</td>
<td>9347581</td>
<td>253767</td>
<td>8364</td>
</tr>
<tr>
<td>Chicken chr 5</td>
<td>76257025</td>
<td>18783005</td>
<td>56836</td>
<td>12089</td>
</tr>
<tr>
<td>Horse chr 29</td>
<td>60735883</td>
<td>4082324</td>
<td>28715</td>
<td>12855</td>
</tr>
<tr>
<td>Chicken chr 18</td>
<td>39972843</td>
<td>833521</td>
<td>19355</td>
<td>12464</td>
</tr>
<tr>
<td>E.coli K-12</td>
<td>50517595</td>
<td>194521</td>
<td>11748</td>
<td>9045</td>
</tr>
<tr>
<td>Fowlpox virus</td>
<td>6447514</td>
<td>36844</td>
<td>9968</td>
<td>8372</td>
</tr>
</tbody>
</table>

Table 4.6 Hits with different reference filters using random generated dataset. This dataset contains 400,000 random generated reads, with the average length of 260.

Table 4.6 has shown several representative reference genomes, with the size range from 1MB (Fowlpox virus) to 200MB (Human chromosome 8). $p$ is determined both by the k-mer size and the reference genome size. For example, if a query dataset has used human chromosome 8 as reference, and the k-mer size is 18, then the relevant $p$ is equal to:

$$\frac{253767}{((260-18+1)*400,000)}$$

which is the number of random hits divided by the total number of k-mer existing in this query file. And $p=0.0026$. 


5. Parallelization for FACS system

In this chapter, I am going to introduce two parallel methods for my previous decontamination and adapter trimming process for FACS 2.0: One is based on SMP parallel (parallel process executed on one CPU) and the other is for multi-CPU distributed process, and in the following text, there will be called as FACS_SMP and FACS_hybrid respectively.

5.1 Large file support

Nowadays, with high throughput sequencing technique, more and more data are stored in single files, which makes the file size becomes huge. Take Illumina Hiseq 2000 machine as an example: an illumine Hiseq has 6 lanes; each can generate 15 to 25 GB data per run, so in total, it can general up to 150GB (300 GB paired-end) data in just one run. In order to rapidly handle such a large amount of data, to take advantage of parallel technique becomes a highly reasonable option.

5.1.1 Standard C library for big file support

Usually, in standard C library, due to the addressing limits, only files smaller than 2GB can be read or written. But now, some new functions are made especially for 64-bit OS in order to allow it to become large file support. To use these functions, flags as follows have to be added:

Macro definition:

#define _LARGEFILE_SOURCE
#define _LARGEFILE64_SOURCE
#define _FILE_OFFSET_BITS 64

Flag for standard open() function:

O_LARGEFILE

Extra function for verifying opening large files:

ftruncate64().
5.1.2 File mapping

File mapping is the association of a file's contents with a portion of the virtual address space of a process. It allows user to fast access the data in the file without actually reading it into memory, which increases the accessing efficiency. To map a piece of data into RAM, a pipeline will be established to maintain the association between RAM and disk, and data will be directly copied to the user’s space, which makes the data be only copied once. However, for normal read/write process, data will be loaded to a buffer before given to user, which involves two times of copy process. And that’s why usually file mapping is much faster than regular read/write process.

5.1.3 Chunk mapping with File mapping

In my program, a normal large file will be divided into chunks. Every time only a chunk of data will be ‘mapped’ and processed. And the program keeps mapping and processing new chunks until all chunks are handled. This feature has been applied both in FACS_SMP and FACS_hybrid.

5.2 Large compressed file

Files that directly come from Illumina machine are usually compressed. Since they are usually large as well, it would be a time and space waste if every time files need to be decompressed before being filtered. To handle that, I import ‘Zlib’[23], which is a compression library for directly accessing compressed file. Not like the file mapping process, data will be read as a file stream; meanwhile, a buffer is essential for gathering data. When buffer is full, all data will be processed. The whole process will not stop until all data from the compressed file is buffered and handled.

By using Zlib, the SMP version is able to handle large files as well. Every time only a chunk of data will be read and processed.
5.3 Parallelization Area Investigation

The parallel redesign should be located where the CPU visits RAM most frequently in order to achieve the most efficiency. Figure 5.1 has shown the main flows of our decontamination tool. Three tasks (Yellow and green area) can be applied with multiple threads process. But the main difference between yellow and green area is that in yellow part, program tries to write data to the same piece of RAM, while in green part, program reads data from the same piece of RAM instead. Letting multiple threads write towards the same RAM could cause potential collision. Moreover, once a Bloom filter is built, it can be saved and used as many time as needed later. So multiple threads process is implemented for contamination detection and removal process, which are almost identical.

Figure 5.1 Main flow chart of FACS decontamination program
The parallel redesign should be located where the CPU visits RAM most frequently in order to achieve the most efficiency. Figure 5.1 has shown the main flows of our decontamination tool. Three tasks (Yellow and green area) can be applied with multiple threads process. But the main difference between yellow and green area is that in yellow part, program tries to write data to the same piece of RAM, while in green part, program reads data from the same piece of RAM instead. Letting multiple threads write towards the same RAM could cause potential collision. Moreover, once a Bloom filter is built, it can be saved and used as many time as needed later. So multiple threads process is implemented for contamination detection and removal process, which are almost identical.
5.4 MPI and OpenMP

For implementing parallel process, we have used two different libraries: OpenMP and MPI (OpenMPI/MPIch). OpenMP is a SMP (Single CPU) based parallel library, allowing all threads share the RAM. But it can only be implemented on SMP (Single node) systems. MPI is a more general parallel library that can work both on SMP systems and clusters. But with MPI, each thread usually execute their task independently, so all threads will have their own (distributed) independent memory (own stack, own heap etc.), which is a computational cost. For instance:

If we have 1GB data to handle, and 4 nodes (8 cores and 16GB RAM each) are available in the cluster. If we simply implement MPI and create 32 threads to handle 1GB data, then each threads needs to have their own copy of that 1GB data, which will cost 32GB in total. Not to mention the individual heap and stack cost and the time cost for thread communication and the complexity for synchronization.

In our case, this situation could really happen quite a lot. To make the work more fast and efficient, for SMP user, we suggest a single OpenMP approach, as Table 5.2 shown below. Correspondingly, I have provided a hybrid approach with OpenMP working on SMP level and MPI working on cluster level, which can be seen from Table 5.4.

5.5 SMP level parallel

5.5.1 Main structure
Table 5.2 (a) Pseudo code of SMP based paralleled decontamination process (b) Pseudo code of multi-thread decontamination process

The main reasons that OpenMP is used for SMP are:

1. With OpenMP library, threads can share the RAM all the time, which makes the task distribution and synchronization process becomes easier.

2. OpenMP generates multiple threads within single node, where threads can be quickly created, distributed and recycled. And after this session, when the parallel job is done, threads can be automatically released or recycled, which is quite convenient and efficient.
Figure 5.3 Data flows in SMP based parallel decontamination process. p1-p7 are chunks of a query file, p4-1…p4-4 are sub chunks. All chunks are divided by program.

Figure 5.3 shows the data flow for decontamination function using OpenMP. Moreover, it also explains how the task is distributed and processed by our tools under single CPU circumstance. Each time, a small file (size <2GB) will be read into RAM as query, it will be distributed later on. Meanwhile, a Bloom filter file will be loaded as reference. After that, multi threads will start from different area of the query file, doing decontamination and saving both clean and contaminated reads. Finally, all clean reads and contaminated reads will be saved into two file, which indicates that the decontamination process is finished.

A query file bigger than 2GB will be read chunk by chunk. For each chunk, same process will be done as the processes of handling small query file expect that, data will be temporary saved into temp files. Until all chunks has been read and processed, temp files will be merged, leaving only two files: the clean file and file with contaminated reads.

5.5.2 Mutual exclusion for multiple threads

In our case, once a read is examined, it will be saved in either clean string or contaminated string for later file saving. Imagine a multiple threads process like that: if two threads finish scanning two reads at the same time and both immediately try to save their reads into one string. A collision will definitely happen and cause two reads cannot be saved correctly one and another. To handle this issue, two kinds of thread lock are applied in our program: atomic lock and critical session:
Atomic lock is a tiny thread lock, which is often used in OpenMP program. Compared to thread locks like critical session, atomic lock provides the least of time loss caused by thread hang-up and release. But it can only be applied where the line only consists of simple variables (integer, float etc.) and simple calculation (plus, minus etc.) In our program, it is used in counting number of hits and so on.

Critical session is another kind of thread lock, which is usually applied to a whole area rather than one or two lines. In critical session, all thread will be hung up and executed one by one, which is definitely much slower than atomic lock. But for complicated computational process such as copy one string to a certain RAM area, it is the most reasonable choice. And it is used in pasting reads to clean/contaminated string in our program, which can also be seen from table 5.2.

### 5.6 Hybrid parallel design for clusters level parallel

#### 5.6.1 Main structure

```plaintext
input : one set of query reads
output: one set of clean reads and one set of contaminated reads
Initialize Nodes set $N \{N_1, ..., N_k\}$;
Other initialization;
load reference bloom filter $B$;
divide query data into chunks and save info into list $L$;
for $N_i \leftarrow N_1$ to $N_k$ do
    distribute a subset $L_i$ from $L$;
    SMP_decontamination($N_i, L_i$);
    save data into temporary files;
    if $N_i \neq N_1$ then
        synchronize info with $N_1$;
    end
end
Set up barrier to let all threads finish their tasks;
$N_1$ merge all temp files and returns statistics;
```

Table 5.4 Pseudo code of hybrid paralleled decontamination process.

SMP_decontamination() process is the process for single CPU, which has been shown in table 5.2 (a) and (b)

Even with single core, our program is able to handle a huge (millions of reads) query file with
OpenMP library. So what is the reason of importing MPI library? The answer is speed.

Firstly, on MPI level, for each available node, a piece of individual chunk would be mapping into the RAM. Secondly, each node will trigger an OpenMP process, distributing the task to every core in it and processing it. After that, temporary files will be saved. Finally, thread communication will be activated. Master thread will gather all the info about temporary files and finally it will merge them together. Just like the OpenMP process, only two files (clean file and file with contaminated reads) will be stored.

On this level, user has the option to control the number of nodes that they want to use. No matter how many nodes they choose, whole task will be adaptively distributed. For instance, for a 20GB query, if chunk size is 1GB, then for one node, it will have to process all 20 chunks. But for 5 nodes, each of them will only process 4 chunks.

5.6.1 Nodes synchronization

According to the hybrid design, for each chunk of data, a relevant clean file and a contaminated
file will be temporarily saved. Lastly, master node need to gather all the file names so that it can merge them back to two compete files. To do this, a node synchronization step is needed right before the program is finished. Since with MPI, threads usually work totally independently, threads synchronization relies on inter-thread communication, in our case, inter-node communication.

In our program, every time when a node finishes scanning all chunks in its subset, it will synchronize with master node to give the temporary file names to master node. Meanwhile, a barrier will be set right before the finalized area to make sure that all nodes have finished their tasks and synchronize the info with master node, which can also be seen in table 5.4.

**5.7 Performance comparison among FACS_SC, FACS_SMP and FACS_hybrid**

In order to prove that the SMP parallel and hybrid parallel frameworks are capable of improving the computational speed, a set of test has been executed to verify how long does FACS single node version, (short as FACS_SC), FACS_SMP and FACS_hybrid need to handle three different query datasets respectively.

**Testing environment**

<table>
<thead>
<tr>
<th>Kalkyl cluster in UPPMAX.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Quad-core Intel® Xeon 5520, eight cores per CPU</th>
</tr>
</thead>
</table>

**Query dataset:**

- Synthetic 454 data: fasta formatted, containing 1794 E.coli K-12 reads out of 10,000 reads in total, average read length 269, 48.3MegaBytes

- Illumina E.coli: fastq formatted, all reads coming from E.coli K-12 2,622,382 reads in total, average read length 58, 316.3MegaBytes

- Homo sapiens: Home Sapiens Nuclear receptor subfamily2, group F, average reads length 101, 14GB.

<table>
<thead>
<tr>
<th>Table 5.6 Testing environment and parameters of parallel comparison test</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>FACS_SC</td>
</tr>
<tr>
<td>FACS_SMP</td>
</tr>
<tr>
<td>FACS_hybrid</td>
</tr>
</tbody>
</table>
**Table 5.7 Performance Comparison with FACS_SC, FACS_SMP and FACS_hybrid**

Star: Only one node is used for handling such query data because query data is relatively small.

Pound: Four nodes have been used for handling query dataset (32 cores in total).

Compared FACS_SC with FACS_SMP in table 5.7, it can be seen that, with any query dataset, FACS_SMP is approximately three times faster than FACS_SC. And this number doesn’t go up when the query data size increases.

Compared FACS_SMP with FACS_hybrid, it seems that FACS_hybrid takes more time to handle small query dataset. That’s probably because when using MPI library, extra time has been spent on initializing step. But for the pound area in table 5.7, since the query dataset is relatively large, a more significant time difference is shown. Rather than one node used in FACS_SMP, four nodes are involved in FACS_hybrid, which makes it faster than FACS_SMP.

In conclusion, according to the experiment result and analysis above, it has been proven that the parallel redesign has significantly increased the processing speed of FACS system if query dataset is large enough.

Additionally, don’t like OpenMP library, MPI library is less adaptive simply because of its complex parallel process. Hence, I only have made FACS_SMP become the default parallel library and leave FACS_hybrid become an alternative option for users.
6. Overall Evaluation

6.1 Introduction

To evaluate the overall performance of FACS2.0, it is essential to introduce two softwares with similar function: Fastq_screen and Deconseq which are both based on popular sequence aligners, which have been proven as fast and highly reliable tools. And since the parallel redesign work in the previous chapter has been verified to become part of the standard FACS library, during this experiment, FACS_SMP will be used as FACS2.0 for the test.

Query dataset:

1. Synthetic 454 data: generated by MetaSim; contains 100,000 reads, with average length of 269; 1794 reads are from E.coli K-12.
2. Illumina E.coli: generated by Illumina HiSeq 1000; contains 2,622,382 reads with equal length of 130; all from E.coli K-12.
3. Illumina Mus musculus: generated by Illumina HiSeq 2000; contains 2,500,000 reads with equal length of 145; all from Lab rat E-15, none from E.coli K-12

Reference Genome:

E.coli K-12 genome.
Human hg19 genome

Testing environment

Kalkyl cluster in UPPMAX.

Parameters:

FACS2.0: Word size k 16, match cutoff 0.3
Deconseq: Alignment identity threshold 1; Alignment coverage threshold 1;
Fastq_screen: Bowtie align mode –n;

Table 6.1 Parameters of Comparison test with Deconseq, Fastq_screen and FACS 2.0
### 6.2 Experiment Result and Analysis

<table>
<thead>
<tr>
<th>Synthetic 454 data (100,000 reads)</th>
<th>Type</th>
<th>Sensitivity</th>
<th>Specificity</th>
<th>Time cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deconseq</td>
<td>99.1%*</td>
<td>94.98%</td>
<td>110.299 s</td>
<td></td>
</tr>
<tr>
<td>Fastq_screen</td>
<td>0.00%</td>
<td>0.00%</td>
<td>8.147 s</td>
<td></td>
</tr>
<tr>
<td>FACS2.0</td>
<td>98.89%*</td>
<td>99.11%</td>
<td>1.309 s</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>E.coli (2,622,382 reads)</th>
<th>Type</th>
<th>Sensitivity</th>
<th>Specificity</th>
<th>Time cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deconseq</td>
<td>93.06%</td>
<td>100.00%</td>
<td>847.136 s</td>
<td></td>
</tr>
<tr>
<td>Fastq_screen</td>
<td>97.52%</td>
<td>100.00%</td>
<td>55.454 s</td>
<td></td>
</tr>
<tr>
<td>FACS2.0</td>
<td>98.33%</td>
<td>100.00%</td>
<td>11.197 s</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mouse (2,500,000 reads)</th>
<th>Type</th>
<th>Sensitivity</th>
<th>Specificity</th>
<th>Time cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deconseq</td>
<td>100.00%</td>
<td>100.00%</td>
<td>531.257 s</td>
<td></td>
</tr>
<tr>
<td>Fastq_screen</td>
<td>100.00%</td>
<td>100.00%</td>
<td>24.935 s</td>
<td></td>
</tr>
<tr>
<td>FACS2.0</td>
<td>100.00%</td>
<td>99.81%</td>
<td>2.467 s</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Human Illumina (44,173,641 reads)</th>
<th>Type</th>
<th>Sensitivity</th>
<th>Specificity</th>
<th>Time cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deconseq</td>
<td>72.77%</td>
<td>100.00%</td>
<td>44770.9 s</td>
<td></td>
</tr>
<tr>
<td>Fastq_screen</td>
<td>Nah</td>
<td>nah</td>
<td>Nah</td>
<td></td>
</tr>
<tr>
<td>FACS2.0</td>
<td>94.24%</td>
<td>100.00%</td>
<td>1049.89 s</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.2 Comparison experiment result of Deconseq, Fastq_screen and FACS2.0 with synthetic 454 data, E.coli illumina data and Mouse data respectively using E.coli K-12 as reference
Table 6.3 Comparison experiment result of Deconseq, Fastq_screen and FACS2.0 with synthetic 454 data, E.coli illumina data and Human Illumina data respectively using Human hg19 as reference. Nah area indicates that Fastq_screen cannot handle a file larger than 4GB.

Table 6.2 shows the result of querying all three dataset against E.coli K-12 reference genome, all with E.coli K-12 genome as reference. The difference can be easily distinguished, which is, with all three query data types, FACS 2.0 provides at least the same performance on sensitivity and specificity while spends the least computational time cost (at least 4.5 times faster), especially when handing Illumina Mouse query dataset (at least 25 times faster).

Table 6.3 generally shows the same tread, but the gap of time cost between FACS 2.0 and Deconseq/Fastq_screen has been narrowed to 2.5 times. The possible reason is that the bloom filter for Human hg19 genome is much larger than hg19 reference for Deconseq/Fastq_screen, it takes more time for FACS 2.0 to load the hg19 reference filter. Once the query dataset is large enough, the gap will become significant again (up to 40 times). This phenomenon can be seen from the third sub graph (Human Illumina query) of Table 6.3.

The reasons that FACS 2.0 is much faster than Deconseq and Fastq_screen is Deconseq and Fastq_screen rely on the alignment scores provided by Bwa and Bowtie respectively before detecting any possible contamination. FACS 2.0 doesn’t need any pre information.

The reason that FACS 2.0 provides different performance with E.coli query dataset than Mouse query datasets is that it is much faster for a Bloom filter to identify an unhit than to identify a hit, using E.coli K-12 as reference, E.coli Illumina query dataset will get much more hits than Mouse Illumina query dataset, and that is why E.coli Illumina query dataset cost much more time for FACS 2.0.

Assume a query dataset consists of 50% clean reads and 50% contaminated reads, and then from the numbers in table 6.2 and table 6.3, the average processing time per read can be calculated:

<table>
<thead>
<tr>
<th>Deconseq</th>
<th>Fastq_screen</th>
<th>FACS 2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.449e-04 s</td>
<td>3.954e-05 s</td>
<td>5.295e-06 s</td>
</tr>
</tbody>
</table>

Table 6.4 Average time for handling single read with Deconseq, Fastq_screen and FACS 2.0

And according to our experiment recordings, for all three tools, the file loading time is approximately 7 seconds per hundred MegaBytes. But Deconseq has chosen a chunk reading policy, so for data size between 100MegaBytes ~ 4GB, it has to access the query file multiple
times, which could cost more time. And the average file writing time for both Deconseq and FACS2.0 are around 10 seconds per hundred MegaBytes. Since Fastq_screen doesn’t write clean or contaminated reads into files, I don’t count it in.

Another interesting phenomenon can be seen table 6.2, which is Fastq_screen seems not capable of correctly dealing with synthetic 454 data, even if a fastq version of this synthetic data has been created for Fastq_screen to scan. The possible reason is that Fastq_screen can only handle reads with the maximum length of 150bp, which is also the usual maximum length of Illumina reads. But the average length for synthetic 454 data is 267, which is a typical length of 454 reads.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Deconseq</th>
<th>Fastq_screen</th>
<th>FACS 2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependency</td>
<td>Bwa</td>
<td>Bowtie</td>
<td>FACS 1.0</td>
</tr>
<tr>
<td>Big query support (&gt;2GB)</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Quick Scan</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Decontamination</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Cross Examine</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Compressed file support</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Supported format</td>
<td>fasta</td>
<td>fastq</td>
<td>fasta</td>
</tr>
<tr>
<td>Supported platform</td>
<td>454 Illumina PacBio Ion Torrent</td>
<td>454 Illumina Ion Torrent</td>
<td>454 Illumina PacBio Ion Torrent</td>
</tr>
<tr>
<td>multi-thread support</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 6.5 Comparison of main features for Deconseq, Fastq_screen and FACS2.0

An important advantage of FACS2.0 has also been indicated by Table 6.5: by taking advantage of Bloom filter structure, carefully designed decontamination process, large query file supported system and parallel processing techniques, FACS2.0 can efficiently handle sequence files that directly coming from a lane of sequencing machines, which is usually compressed, containing hundreds millions of reads.

In reality, query file that is potentially contaminated usually contains a small proportion of alien reads. In that case, most of the k-mers in query file would get unhit when using Bloom filter to do decontamination work, which will makes FACS2.0 becomes much more faster when handling real query files. From table 6.2, we can clearly see that the when doing a decontamination for Mouse query dataset, FACS2.0 is incredibly 25 times faster than Fastq_screen and 100 times faster than Deconseq, and it is only 5 times and 20 times faster respectively when processing E.coli query
dataset with similar read number. Combine the characteristic that is mentioned above and the fact that we observed in the table, an obvious conclusion could be drawn, which is, FACS2.0 is an ideal tool for decontamination.
7. Adapter trimmer

Nowadays, when applying sequencing process, adapters are usually involved. For instance, in shotgun sequencing with cyclic-array methods, adapters are used to fragment genomic DNA [25]. And after sequencing, there is probably that reads still are with adapters attached on 5’ or 3’ side.

Figure 7.1 an example of adapters in sequencing process [24]

Figure 7.1 shows the typical case of planting adapters in sequencing process, where red area represents the real sequence fragments, black area represents the adapters. In other real cases, just like Figure 7.1, adapters are potentially planted on 5’ or 3’ side. So if we use a sliding window process both start from the 5’ area (direction from 5’ to 3’) and 3’ area (direction from 3’ to 5’), then all the alien bases which are actual adapters and not part of the reference genome will be removed, in that case, theoretically, the residue adapters can be removed. Because of the k-mer based sliding window approach, FACS system can also be potentially applied to trim the adapters on both the 5’ and 3’ area within a read. And in this chapter, I am going to introduce my exploration of developing adapter trimming function with FACS system.

7.1 Architecture

Since there is no existing adapter trimmer that can be drawn from, based on the sequencing structure of typical adapters and the theory mentioned before, I have designed a FACS based adapter trimmer. The main structure has been shown as follows:
7.2 Performance evaluation for Prototype

7.2.1 Synthetic query dataset

For the testing purpose, a synthetic query dataset has been built, which is based on E.coli K-12. Every read contains a fragment extracted from the reference genome, and filled with pure random bases pretended as adapters. Since all reads are set to the same length, the length of adapters for each read will be different.

Moreover, in real cases, according to Ekdahl and Farahani’s [27] research, the average length of
true fragments among the whole reads would generally follow a Poisson distribution, which can be seen from figure 7.3.

![Figure 7.3 Average true SOLiD read length after removing all the adapters with E15 mouse dataset sequenced by Scilifelab [27]](image-url)
7.2.1 Performance analysis

Figure 7.4 (a) Performance of FACS based adapter trimmer without noise in query dataset
(b) Performance of FACS based adapter trimmer with random noise in query dataset

Figure 7.4 (a) and (b) measures the performance of FACS based adapter trimmer under different parameters and query dataset. In this test, there are 1,000 reads in total. Generally said, as the minimum length of true fragment grows, more reads are correctly trimmed, which is an expected phenomenon. Meanwhile, the smaller the word size k, the more correctly trimmed reads are. The
result suggests us that for trimming the adapters within a read, a short word size k should be chosen.

Although from Figure 7.4 (a) and (b), we can see that the result is decent, but when handling real dataset, a lot of unacceptable over-trimmed reads occur. The possible reason is that for adapter trimming, sequences need to be cut accurately. Sometimes, only one base mismatch can lead to a total failure of trimming. And this case happens often in the real sequence case, which also indicates that using bloom filter along with k-mer approach cannot do adapter trimming properly.
8. Conclusion and Future work

8.1 Conclusion

The initial goal of this project is actually to optimize the original bloom filter library, and discover possible applications of FACS system. Finally, the decontamination application has been proved feasible; however the adapter trimming application is not due to the limitation of bloom filter and k-mer approach. Moreover, the parallel process, especially the hybrid FACS has been shown as an efficient parallel process for enforcing the performance of FACS system.

Here are the main achievements of my thesis project can be described as follows:

1. During the project, I have found that the FACS 1.0 system is not efficient enough, which mainly reflects on the k-mer hashing process and filter addressing process. To optimize that, I have redesigned the filter addressing process so that bit position can be addressed faster, and this can influence the performance dramatically. I replaced the old hash function (Jenkins hash) with a slower but more accurate and reliable one (Lookup8 hash) to make the Bloom filter generate less false positive hits. In addition, I have used fixed seeds approach to replace the previous random generated seeds approach so that the speed can be raised without affecting the accuracy.

2. I have investigated the possibility of applying FACS system to sequence decontamination. Furthermore, for decontamination application, I have developed a new scoring system for measuring whether a read is contaminated or not, as well as a new contamination probability score for evaluating the probability of a sequence being contaminated. Last but not least, I have created a new contamination quick scanning process, which allows FACS system to go through all the query sequence and quickly report the contamination proportion.

3. I have provided a new paralleled solution of FACS system to allow it to take advantage of parallel technology. Without any pre-setting, all cores in one CPU will be automatically distributed with equal tasks and involved in decontamination process. Additionally, a mixed framework with both OpenMP and MPI library has been developed so that both inner CPU (all cores within one CPU) parallel process and inter CPU (all available CPUS) parallel process can be executed at the same time. For instance, when users are running FACS2.0 on clusters, the program
can make the best use of every the available cores and every available CPUs to sharply increase the processing speed.

Figure 8.1 Family tree of FACS 2.0 system
From figure 8.1, all the variations in FACS 2.0 system and their relations can be seen.
Since the FACS system becomes mature, another extension has been done as well. A python interface has been built for FACS 2.0 so that people who are familiar with python can directly run FACS under python environment.

8.2 Future work

8.2.1. True sequence classifier
FACS prototype is originally designed as a fast sequence classification tool, which means reads will be classified to any reference genomes if the match cutoff reaches the threshold. A useful extension of FACS is to make it become a true aligner. Instead of the classifying current method, sequence will be classified to the reference genome which provides the highest match score so that the FACS will be the same functional as blast but much faster.
References


Appendix A

Figure X (From left to right)

(up-left) Total reads classified to E.coli Genome with OLD scoring system; (up-right) True positive number of classified reads with OLD scoring system; (mid-left) Total reads classified to E.coli Genome with NEW scoring system; (mid-right) True positive number of classified reads with NEW scoring system; (down-left) Total reads classified to E.coli Genome with NEW scoring system and CHB; (down-right) True positive
number of classified reads with NEW scoring system and CHB
Appendix B

Figure Y (From left to right)
(up-left) Total reads classified to Human chromosome 8 Genome with OLD scoring system; (up-right) True positive number of classified reads with OLD scoring system; (mid-left) Total reads classified to Human chromosome 8 Genome with NEW scoring system; (mid-right) True positive number of classified reads with NEW scoring system; (down-left) Total reads classified to Human chromosome 8 Genome with NEW
scoring system and CHB; (down-right) True positive number of classified reads with NEW scoring system and CHB
Acknowledgement

I would like to dedicate my appreciation to my examiner Prof. Jens Lagergren, my supervisor Dr. Lars Arvestad and my collaborators Dr. Henrik Stranneheim, Roman Valls Guimera and Hossein Farahani for their supervision and help. I could never finish this project without them. Also I would like to express my appreciation to my parents, my girlfriend Alin and all my friends for their generous support.