T2R2 東京工業大学リサーチリポジトリ
Tokyo Tech Research Repository

論文 / 著書情報
Article / Book Information

題目(和文)	大量メタゲノム情報に対するアミノ酸配列相同性検索の高速化
Title(English)	Faster Protein Sequence Homology Searches for Large-scale Metagenomic Data
著者(和文)	鈴木脩司
Author(English)	Shuji Suzuki
出典(和文)	学位:博士（工学）, 学位授与機関:東京工業大学, 報告番号:甲第9884号, 授与年月日:2015年3月26日, 学位の種別:課程博士, 審査員:秋山　泰,佐藤　泰介,宮崎　純,村田　剛志,関嶋　政和
Citation(English)	Degree:, Conferring organization: Tokyo Institute of Technology, Report number:甲第9884号, Conferred date:2015/3/26, Degree Type:Course doctor, Examiner:,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

Powered by T2R2 (Tokyo Institute Research Repository)

Faster Protein Sequence Homology Searches for Large-scale Metagenomic Data

Shuji Suzuki

Department of Computer Science
Graduate School of Information Science and Engineering
Tokyo Institute of Technology

Supervisor: Yutaka Akiyama

A Thesis Submitted for the Degree of *Doctor of Engineering*

February 20, 2015

Copyright © 2015 Shuji Suzuki

This dissertation partly used the published articles:

- © 2014 Suzuki *et al.* (Chapter 3)
- © 2014 Suzuki *et al.* (Chapter 4)
- © 2012 Suzuki *et al.* (Chapter 5)
- © 2012 IEEE (Chapter 7)

Abstract

Sequence homology search is an approach for establishing structural and functional similarity with existing genes or proteins using a variety of databases containing a large number of DNA and protein sequences and the associated biological information. Sequence homology search is used in metagenomics. However, because of improvements in DNA sequencing technology, the volume of sequence data and the number of queries used in this analysis have been increasing rapidly in recent years, and the speed of sequence homology search has become insufficient.

In this dissertation, we propose fast protein sequence homology search algorithms that can be applied to metagenomics using the latest DNA sequencing output. We used three approaches: development of novel protein sequence homology search algorithms, acceleration of protein sequence homology search with graphics processing unit (GPU), and parallelization of protein sequence homology search using modern supercomputing environments.

We propose a novel protein sequence homology search algorithm that finds similarities between a query and database sequences based on the suffix arrays of these sequences. We used a subsequence search method relying on a similarity-based optimal length. This algorithm designated as GHOSTX provides approximately 165 times faster protein sequence homology search than BLASTX in the analysis of metagenomic data. In addition, we propose a novel protein sequence homology search method based on database subsequence clustering, designated as GHOSTZ. This method clusters similar subsequences retrieved from a database to reduce alignment candidates based on triangle inequality, and its performance in the analysis of metagenomic data is approximately two times faster than that of GHOSTX.

In addition, we applied the GPUs and massively parallel computing systems, TSUBAME and the K computer, for protein sequence homology search and show that these approaches provide a significant acceleration of protein sequence homology search.

DNA sequencing technology is constantly improving, resulting in generation of vast amounts of sequence data. This explosion of sequence volume makes computational analysis with contemporary tools more difficult. Here, we offer the algorithms, which may provide a potential solution to this problem.

Acknowledgements

I would like to express my sincere gratitude to Professor Yutaka Akiyama for providing an excellent research environment, continuous guidance and encouragement throughout my research work. I also thank Associate Professor Takashi Ishida for his valuable comments, suggestions, and discussions of my research. Dr. Masanori Kakuta is one of the most important co-authors in my research articles; he provided insightful comments and suggestions. I also would like to thank all members of Professor Akiyama's laboratory for helpful discussions and wonderful time spent together. I am grateful to Ms. Kanako Ozeki, the secretary of Akiyama's laboratory, for her much appreciated support during my six years of laboratory life.

For the study on GHOSTZ-GPU (Chapter 6), I thank Mr. Brent Oster, Mr. Marc Hamilton, and Mr. Mark Berger at NVIDIA Corporation for providing an excellent research environment, insightful comments, and suggestions on the study. I wish to express my gratitude to my dissertation committee, Professor Taisuke Sato, Professor Jun Miyazaki, Associate Professor Tsuyoshi Murata, and Associate Professor Masakazu Sekijima at Tokyo Institute of Technology for their inspiring feedback and valuable comments on my research and dissertation. Professor Ken Kurokawa and Associate Professor Fumikazu Konishi at Tokyo Institute of Technology, Professor Hiroyuki Ogata at Kyoto University, and Dr. Jun Sese at the National Institute of Advanced Industrial Science and Technology also gave valuable suggestions related to my research.

I would like to express my sincerest gratitude to the Japanese Society for the Promotion of Science (JSPS) Research Fellow (DC1), CUDA COE Program by NVIDIA Corporation, and the Ministry of Education, Culture, Sports, Science and Technology (MEXT) Program for Leading Graduate Schools "Education Academy of Computational Life Sciences (ACLS)" led by Professor Yutaka Akiyama, for financial support of my research works. This study was also supported in part by a Grant-in-Aid for JSPS Fellows 24·8766.

I would like to thank the Young Researches Society for Bioinformatics (bioinfowakate) for valuable discussions and constant encouragement. I would also like to thank all young researchers who belong to the young researchers' community.

Last, but certainly not the least, I would like to extend special thanks to my parents Mitsuhiro Suzuki and Takako Suzuki. Without their kind support and constant encouragement, this work would not be possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Sequence homology search widely used in biological analyses, is an approach to establishing structural and functional similarity with existing genes or proteins using a variety of databases containing a large number of DNA and protein sequences and the associated biological information. This method is related to string search, which is an important problem in computer science.

1.1 String Search

String search identifies positions where one or several patterns are found within a text. Let Σ be an alphabet of size $|\Sigma|$, P be a pattern and T be a text. P and T are sequences of characters over Σ. The lengths of P and T are $|P|$ and $|T|$, respectively. A brute-force string search algorithm checks all positions of P within T. This algorithm requires $O(|P||T|)$ time. In practice, the positions of substrings similar to P within T are also required. This problem known as approximate string search is more complicated compared to string search because of mismatches or gaps. Approximate string search often uses edit distance [26] to measure similarity between P and a substring in T. Edit distance allows deleting, inserting, and replacing a character in both strings. The classical solution uses dynamic programming [44]. It requires $O(|T||P|)$ time. These string searches require long computation time, if large amount of string data is used. To accelerate string search, a number of algorithms with data structured as indexes such as a hash table and a suffix array are proposed [21, 36, 39]. For example, the Rabin-Karp algorithm [21] uses hashing to find P within T with an average computational time of $O(|P|+|T|)$. String search, including approximate string search, is useful in many applications such as spelling error correction [56], image compression

[34], and data mining [9].

1.2 String Search in Bioinformatics

String search, including approximate string search, is widely employed in biological analyses. DNA and protein molecules represent specific sequences of nucleotides and amino acids, respectively, and, therefore, can be expressed as strings of nucleotides and amino acids denoted as alphabet characters. DNA and protein sequences are composed of sets of 4 and 20 letters, respectively, and thus, can be easily analyzed by computational methods.

One of the methods used for approximate string search in biological analyses is sequence homology search, often employed for identifying evolutionary relationships among species. If two species have similar sequences, they may have a common ancestor, and these sequences are called homologous. Sequence homology search can be also used for estimating potential functions and structures of unknown biomolecules, because DNA and proteins of similar sequences often demonstrate similar structures and biological functions. Approximate string search in sequence homology analysis is based on sequence similarity to relate with evolutionary distance instead of edit distance. This similarity is often defined as the score of sequence alignment calculated with a more complex scoring scheme than edit distance. A match, mismatch, and gap penalty in edit distance are 0, 1, and 1, respectively. For DNA sequence, a match is given a positive score and a mismatch is penalized by a negative score, whereas, gap penalty varies depending on gap length. For protein sequence, a match and mismatch are defined for each character pairs. BLOSUM62 [18] is the matrix defining match scores and mismatch penalties for character pairs. One of the characteristics in the score matrix for protein sequence is a mismatch score, which is usually negative in approximate string search. However, in this score matrix, the score for a pair of different characters may be positive when the corresponding amino acids have similar properties. For example, the score for isoleucine (I) and leucine (L) is 2 in this score matrix. The differences in the scoring system prevent using an effective pruning approach for edit distance. A sequence homology search tool based on dynamic programming [41] can find the closest sequence similarity to a query in a database; however, it requires considerable computation time. Therefore, BLAST [4, 5], which is based on a heuristic algorithm, is often used in biological analyses.

Another method employing approximate string search in biological analyses is mapping, which determines the location of each short DNA fragment in a genome. In shotgun sequencing, DNA molecules are randomly broken into numerous small segments, and DNA sequencers generate information in the form of short fragments (reads) ranging from 100 to 1,000 base pairs (bp). Thus, to utilize known biological information, even when a reference genome is available, it is necessary to determine the location of each read in a genome. Many effective mapping programs, such as BWA [27, 28] and Bowtie [24, 25], have been developed for this purpose. Mapping uses the same sequence similarity approach as DNA sequence homology search; however, a read should be quite similar to the subsequence in a reference genome. This assumption of the similarity between a read and reference genome is used by a number of heuristic approaches incorporated into current mapping tools, which are estimated to be over 10,000 times faster than BLAST.

1.3 Current Explosion of Biological Sequence Data

The computation time of sequence homology search depends largely on the sizes of a query and database, which have been rapidly increasing in recent years because of the progress in DNA sequencing technology. These advancements have also reduced the cost of DNA sequencing. The cost of sequencing a genome equal by size to the human genome is shown in Fig. 1.1. The reduction in DNA sequencing cost is outpacing Moore's Law beginning January 2008; currently, DNA sequencing cost is approximately 24 times less than it was 5 years ago, which for a human-size genome would be 4,905 dollars. While DNA sequencing costs are decreasing, the data available for biological analyses are constantly increasing. DNA sequencing based on the latest technology is called next generation sequencing (NGS). One of NGS instruments, Illumina HiSeq2500, can produce approximately 1 terabase (Tb) sequence data in a single run.

Similarly, the computation time for mapping also depends on the size of a query and reference genome and is increasing along with the progress in sequencing technology. However, current mapping tools can use effective heuristic approaches, and, therefore, can perform fast mapping. On the other hand, sequence homology search cannot use heuristic approaches for requiring more search sensitivity; thus, the analysis based on sequence homology search such as metagenomics still takes long computation time.

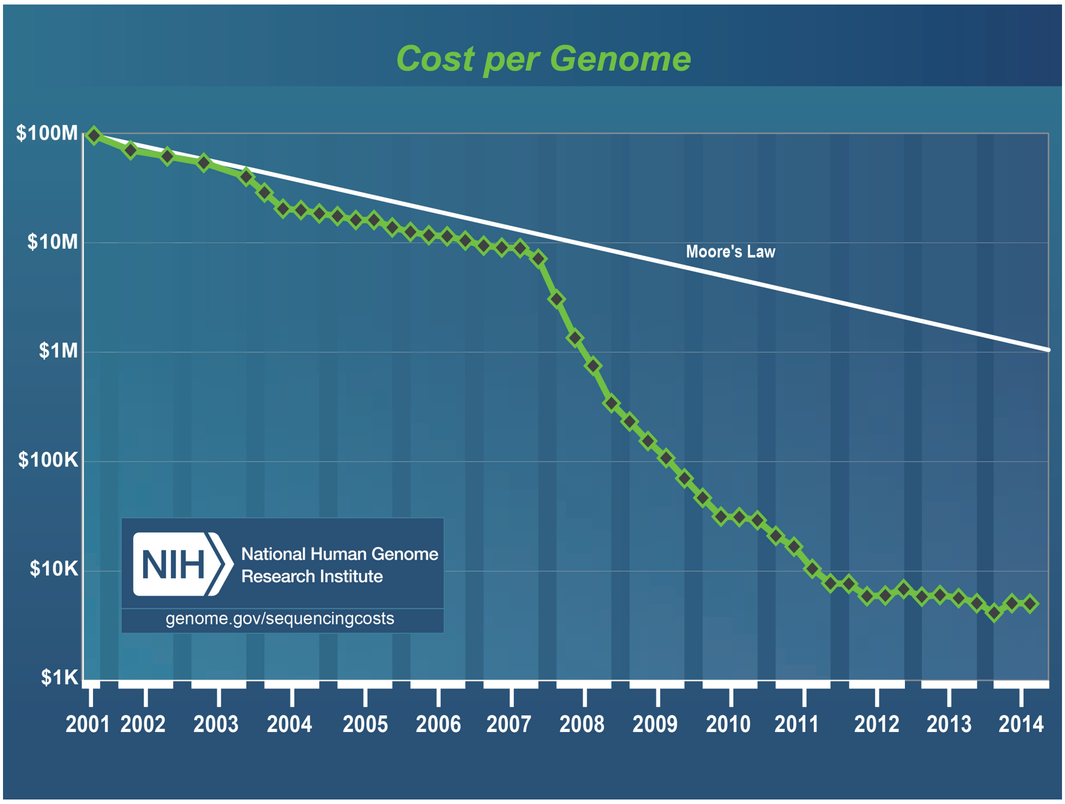


Figure 1.1: Cost per genome [1]. (Courtesy: National Human Genome Research Institute)

1.4 Metagenomic Analysis

Metagenomics, which is the study of the genomes of uncultured microbes obtained directly from microbial communities in their natural habitats, has recently become more popular because of the rapid improvement of DNA sequencing technologies. Microbes live in every environment, including soil, ocean, and human body, and their community structure is defined by environmental conditions, while they, in turn, affect biological characteristics of their habitat. It is, therefore, very important to understand structure-functional relationship between microbial populations and their environment.

Conventional genomic analysis is usually performed for a single microorganism, which should first be cloned and obtained as a pure culture. However, such an approach cannot be applied to a wide variety of microorganisms existing in nature and thus, fails to represent the true status of microbial population structure and biological interactions within microbial community. Metagenomics based on DNA sequences directly obtained from mixed microbial populations in their natural habitats provides a more comprehensive approach to the problem. In metagenomic analysis, environmental samples usually contain DNA sequences from many different species, whereas the reference database often does not contain closely related genomes. Thus, more sensitive approaches than mapping are required. In a typical metagenomic analysis, reads are translated into protein sequences and assigned to protein families by running protein sequence homology searches against publicly available databases such as KEGG [20, 19], COG [50, 51], and Pfam [12]. The BLASTX program, which is one of implementations of the BLAST algorithm, is widely used for such binning and classification searches. To identify homologues that may not have high DNA sequence identities, BLASTX translates query DNA sequences into protein sequences, and then performs protein sequence homology search against a protein sequence database, because protein sequences are often more similar than the original DNA sequences [53, 23]. When a DNA sequence is translated in BLASTX, each codon is converted into a corresponding amino acid using three possible reading frames in a single DNA strand, and a double-stranded DNA molecule is thus translated into six protein sequences.

However, the search speed provided by BLASTX has become insufficient for the analysis of currently available large sequence data. If we perform BLASTX with the data produced by Hiseq2500 and stored in the KEGG GENES database on 1 CPU core in metagenomic analysis, it is estimated to require 1 million CPU days. Several currently available sequence homology search tools are faster than BLASTX. For example, BLAT [22] is approximately 50 times faster than BLASTX in protein sequence homology search; however, the search sensitivity of BLAT is much lower than that of BLASTX and is often insufficient for metagenomic analysis. Recently, novel protein sequence homology search tools such as RAPSearch [58] have been developed. RAPSearch has a sufficient sensitivity for metagenomics and provides faster homology search than BLASTX or BLAT, because it uses a reduced amino acid alphabet [38] and a suffix array [36]. In addition, RAPSearch2 has been improved to use hash tables instead of suffix arrays, which makes it more memory-efficient [59].

However, several large metagenome projects such as the Human Microbiome Project (HMP) [52], the Metagenomics of the Human Intestinal Tract (MetaHIT) [43], and the Earth Microbiome Project [16] have recently produced unprecedentedly large amounts of sequence information. For instance, HMP has sequenced 681 whole human metagenome shotgun samples. In addition, the number of reference sequences in the databases would continue to grow in parallel with further progress in sequencing technologies. For example, the size of the National Center for Biotechnology Information (NCBI) non-redundant protein database (nr) [7] have increased from approximately 4.1 billion amino acids in 2010 to approximately 16.7 billion amino acids in 2014. Therefore, the speed of homology searches needs to be increased to facilitate metagenomic analysis.

1.5 Purpose of Study

In the present study, we describe the development of protein sequence homology search algorithms that can be applied to metagenomic analysis with NGS output. Fig. 1.2 shows our approaches to improve protein sequence homology search. For the increase of analyzed sequence data, we developed and implemented fast protein sequence homology search algorithms. In addition, we accelerated protein sequence homology search with the graphics processing unit (GPU) and parallelized the search using modern supercomputing environments.

1.6 Summary of Contributions

The contributions of this thesis are classified into three categories: (i) development of novel protein sequence homology search algorithms, (ii) acceleration of protein sequence homology search with the GPU, and (iii) parallelization of protein sequence homology search using modern supercomputing environments. We now describe these contributions in more detail.

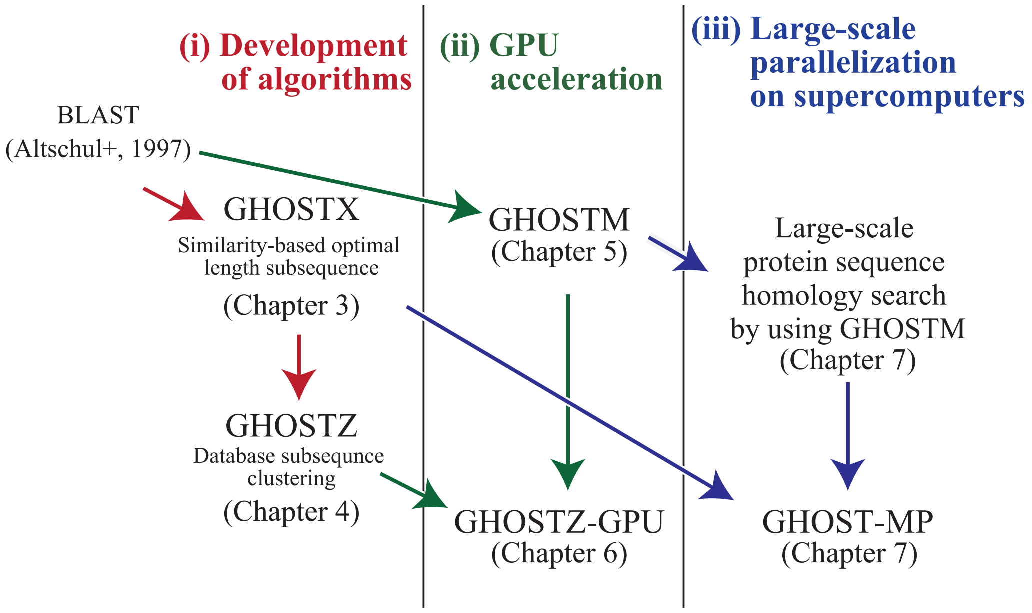

(i) Development of algorithms
(ii) GPU acceleration
(iii) Large-scale parallelization on supercomputers
BLAST (Altschul+, 1997)
GHOSTX
Similarity-based optimal length subsequence
(Chapter 3)
GHOSTM
(Chapter 5)
Large-scale protein sequence homology search by using GHOSTM
(Chapter 7)
GHOSTZ
Database subsequnce clustering
(Chapter 4)
GHOSTZ-GPU
(Chapter 6)
GHOST-MP
(Chapter 7)

Figure 1.2: Relationship among approaches to improve protein sequence homology search.

(i) **Development of novel protein sequence homology search algorithms**

(1) We propose a protein sequence homology search algorithm that finds similarities between query and database sequences based on suffix arrays of these sequences. We used a seed search method relying on a similarity-based optimal length. In the algorithm, only seeds with a sufficient match score are searched based on a given score matrix. Thus, the algorithm can effectively exclude seeds with sufficient length but insufficient similarity. We designated it as GHOSTX. When we evaluated GHOSTX performance with metagenomic data,**GHOSTX demonstrated an approximately 131–165 times faster search than BLASTX**.

(2) We propose a protein sequence homology search method based on database subsequence clustering, and designated it as GHOSTZ. This method clusters similar subsequences retrieved from a database to reduce alignment candidates based on triangle inequality. This database subsequence clustering method provides an approximately two-fold increase in speed without a significant decrease in search sensitivity. When we evaluated GHOSTZ performance with metagenomic data, **it was approximately 213–285 times faster than BLASTX**.

(ii) **Acceleration of protein sequence homology search with GPU**

(1) We have developed a protein sequence homology search algorithm suitable for GPU calculations. We implemented it as a GPU system, and designated it as GHOSTM. GHOSTM first searches for positions of sequence alignment candidates retrieved from a database using a hash table and then calculates the scores of local alignments around the candidate positions before calculating similarity. **The system with 1 and 4 GPUs achieved calculation speed that was approximately 130 and 407 times, respectively, higher than BLASTX with 1 CPU core**. The system with 1 and 4 GPUs also showed higher search sensitivity and performed calculations approximately 4 and 15 times, respectively, faster than BLAT with 1 CPU core.

(2) We have developed a GPU version of GHOSTZ. Several calculations such as distance calculation, ungapped extension, and gapped extension are the bottlenecks in GHOSTZ. We mapped these processes to the GPU and designated the version as GHOSTZ-GPU. In this version, we optimized memory access in GPU calculation. In addition, GHOSTZ-GPU uses CPU-GPU heterogeneous computing to improve utilization efficiency of the CPU and GPU. When we used metagenomic data, **GHOSTZ-GPU with 12 CPU cores and 3 GPUs was approximately 5.1–7.1 times faster than GHOSTZ with 12 CPU cores.**

(iii) **Parallelization of protein sequence homology search in modern supercomputing environments**

(1) We have developed a large-scale system for analyzing vast amounts of metagenomic data obtained by NGS. This system enables us to analyze NGS-generated metagenomic data in real time by utilizing huge computational resources provided by TSUBAME 2.0. We used GHOSTM to analyze metagenomic data in this system, and show that **the system could process about 60 million reads per hour with 2,520 GPUs (840 computing nodes).**

(2) We have developed a method for parallel protein sequence homology search on massively parallel computing systems. This method provides fast protein sequence homology search with database indexes and hierarchical parallel search and allows large-scale metagenomic analysis. Its parallel efficiency

and search speed were evaluated using two massively parallel computing systems, TSUBAME 2.5 and the K computer. **The method could process over 10,000 CPU cores approximately 89 times faster than mpiBLAST**, a tool to perform BLAST in a parallel computing system.

1.7 Thesis Organization

The remaining chapters of this thesis are organized as follows: Chapter 2 reviews studies on sequence homology search focusing mainly on protein sequences. Chapter 3 describes a novel protein sequence homology search algorithm GHOSTX that finds subsequence similarity between query and database sequences based on suffix arrays. Chapter 4 presents a novel protein sequence homology search method GHOSTZ based on database subsequence clustering. Chapters 5 and 6 discuss GPU implementations and execution results using the GPUs. Chapter 7 discusses the application of the novel protein sequence homology search algorithm to massively parallel computing systems and the execution results in supercomputing environments. Conclusions are presented in Chapter 8 together with future work. In addition, Appendix A describes the comparison of search sensitivity with score based on E-value.

This thesis is based on the following publications by the author: [47, 46, 48, 49].

Chapter 2

Sequence Homology Search

2.1 Introduction

Sequence homology search is a method of searching sequence databases by using alignment to a query sequence. The analysis of sequence similarity is essential for identifying evolutionary relationships among species; it can also be used for estimating potential structures and functions of biomolecules. In biological analyses, the highest score of sequence similarity between a query and database sequences obtained by sequence alignment is generally used.

2.2 Sequence Alignment

Sequence alignment is an arrangement of DNA or protein sequences inclusive of gaps. The score of an alignment is calculated based on a scoring scheme consisting of scores for each substitution pattern and gap penalty. For DNA sequences, an exact match gives a positive score and a mismatch is penalized by a negative score. For protein sequences, substitution score matrices such as PAM [10] or BLOSUM [18] are often employed. Substitution scores in these matrices are defined for each character (amino acid) pair. Several amino acids have similar physicochemical properties and are often evolutionary conserved. Thus, substitution scores for such pairs are higher than for the other pairs. For example, isoleucine (I) and leucine (L) have similar physicochemical properties and the substitution score for this pair is 2 in BLOSUM62, while substitution scores for most pairs are negative. (Fig. 2.1). The alignment with the highest score is called optimal sequence alignment.

Alignment methods are classified into two categories, global alignment and local

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	4	−1	−2	−2	0	−1	−1	0	−2	−1	−1	−1	−1	−2	−1	1	0	−3	−2	0
R	−1	5	0	−2	−3	1	0	−2	0	−3	−2	2	−1	−3	−2	−1	−1	−3	−2	−3
N	−2	0	6	1	−3	0	0	0	1	−3	−3	0	−2	−3	−2	1	0	−4	−2	−3
D	−2	−2	1	6	−3	0	2	−1	−1	−3	−4	−1	−3	−3	−1	0	−1	−4	−3	−3
C	0	−3	−3	−3	9	−3	−4	−3	−3	−1	−1	−3	−1	−2	−3	−1	−1	−2	−2	−1
Q	−1	1	0	0	−3	5	2	−2	0	−3	−2	1	0	−3	−1	0	−1	−2	−1	−2
E	−1	0	0	2	−4	2	5	−2	0	−3	−3	1	−2	−3	−1	0	−1	−3	−2	−2
G	0	−2	0	−1	−3	−2	−2	6	−2	−4	−4	−2	−3	−3	−2	0	−2	−2	−3	−3
H	−2	0	1	−1	−3	0	0	−2	8	−3	−3	−1	−2	−1	−2	−1	−2	−2	2	−3
I	−1	−3	−3	−3	−1	−3	−3	−4	−3	4	2	−3	1	0	−3	−2	−1	−3	−1	3
L	−1	−2	−3	−4	−1	−2	−3	−4	−3	2	4	−2	2	0	−3	−2	−1	−2	−1	1
K	−1	2	0	−1	−3	1	1	−2	−1	−3	−2	5	−1	−3	−1	0	−1	−3	−2	−2
M	−1	−1	−2	−3	−1	0	−2	−3	−2	1	2	−1	5	0	−2	−1	−1	−1	−1	1
F	−2	−3	−3	−3	−2	−3	−3	−3	−1	0	0	−3	0	6	−4	−2	−2	1	3	−1
P	−1	−2	−2	−1	−3	−1	−1	−2	−2	−3	−3	−1	−2	−4	7	−1	−1	−4	−3	−2
S	1	−1	1	0	−1	0	0	0	−1	−2	−2	0	−1	−2	−1	4	1	−3	−2	−2
T	0	−1	0	−1	−1	−1	−1	−2	−2	−1	−1	−1	−1	−2	−1	1	5	−2	−2	0
W	−3	−3	−4	−4	−2	−2	−3	−2	−2	−3	−2	−3	−1	1	−4	−3	−2	11	2	−3
Y	−2	−2	−2	−3	−2	−1	−2	−3	2	−1	−1	−2	−1	3	−3	−2	−2	2	7	−1
V	0	−3	−3	−3	−1	−2	−2	−3	−3	3	1	−2	1	−1	−2	−2	0	−3	−1	4

Figure 2.1: BLOSUM62

alignment.

2.2.1 Global Alignment

In global alignment, every character in a given sequence is aligned. Global alignment is computed by using similarity to relate to evolutionary distance; it is most useful when sequences are similar and of roughly equal size. Global alignment is calculated by Needleman-Wunsch algorithm [40]. The example of global alignment is shown in Fig. 2.2.

2.2.2 Local Alignment

Local alignment aligns similar regions within sequences. This alignment is used when sequences are generally dissimilar but contain similar regions or motifs. Local alignment is calculated by the Smith-Waterman algorithm [45] or Gotoh algorithm [17]. The example of local alignment is shown in Fig. 2.3. The alignment is only performed for central regions that are similar in these sequences. Local alignment used to identify similar regions or sequence motifs related to potential functional, structural,

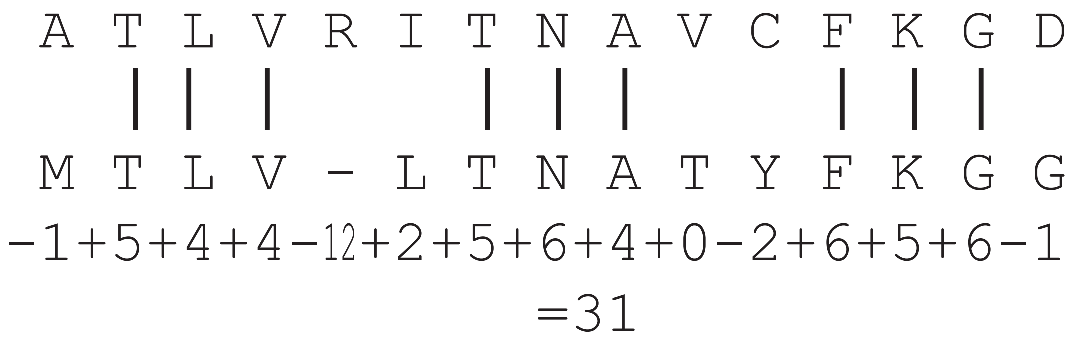


Figure 2.2: The example of global alignment. When we used BLOSUM62 and −12 as gap penalty, the score of this global alignment is 31.

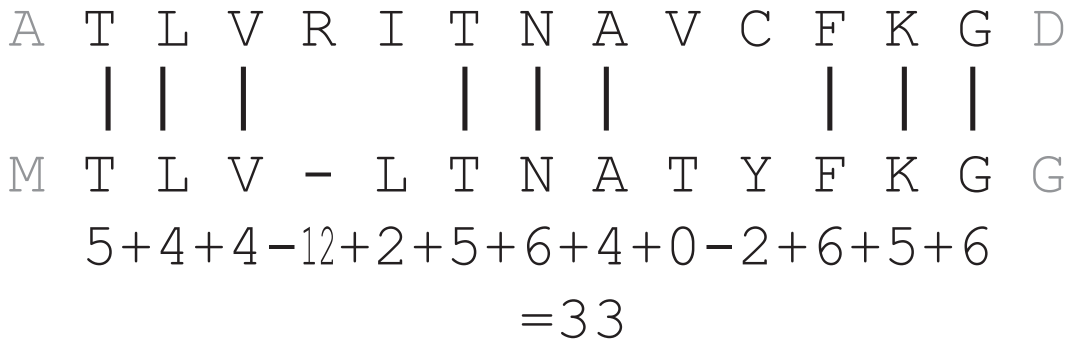


Figure 2.3: The example of local alignment. When we used BLOSUM62 and −12 as gap penalty, the score of this local alignment is 33.

and evolutionary similarity, is performed by a number of sequence homology search tools such as BLAST, BLAT, and RAPSearch.

2.3 Dynamic Programing for Sequence Alignment

The Needleman-Wunsch algorithm, Smith-Waterman algorithm, and Gotoh algorithm are based on dynamic programming to calculate optimal sequence alignment. However, when sequences are long, these algorithms take significant computation time. Let M be the length of one sequence and N the length of the other sequence. The computation time of these algorithms is $O(MN)$.

2.3.1 Smith-Waterman Algorithm

The Smith-Waterman algorithm [45] is used for local alignment. Let A and B be two sequences. A and B are given below.

$$A = a_1a_2...a_M \tag{2.1}$$

$$B = b_1b_2...b_N \tag{2.2}$$

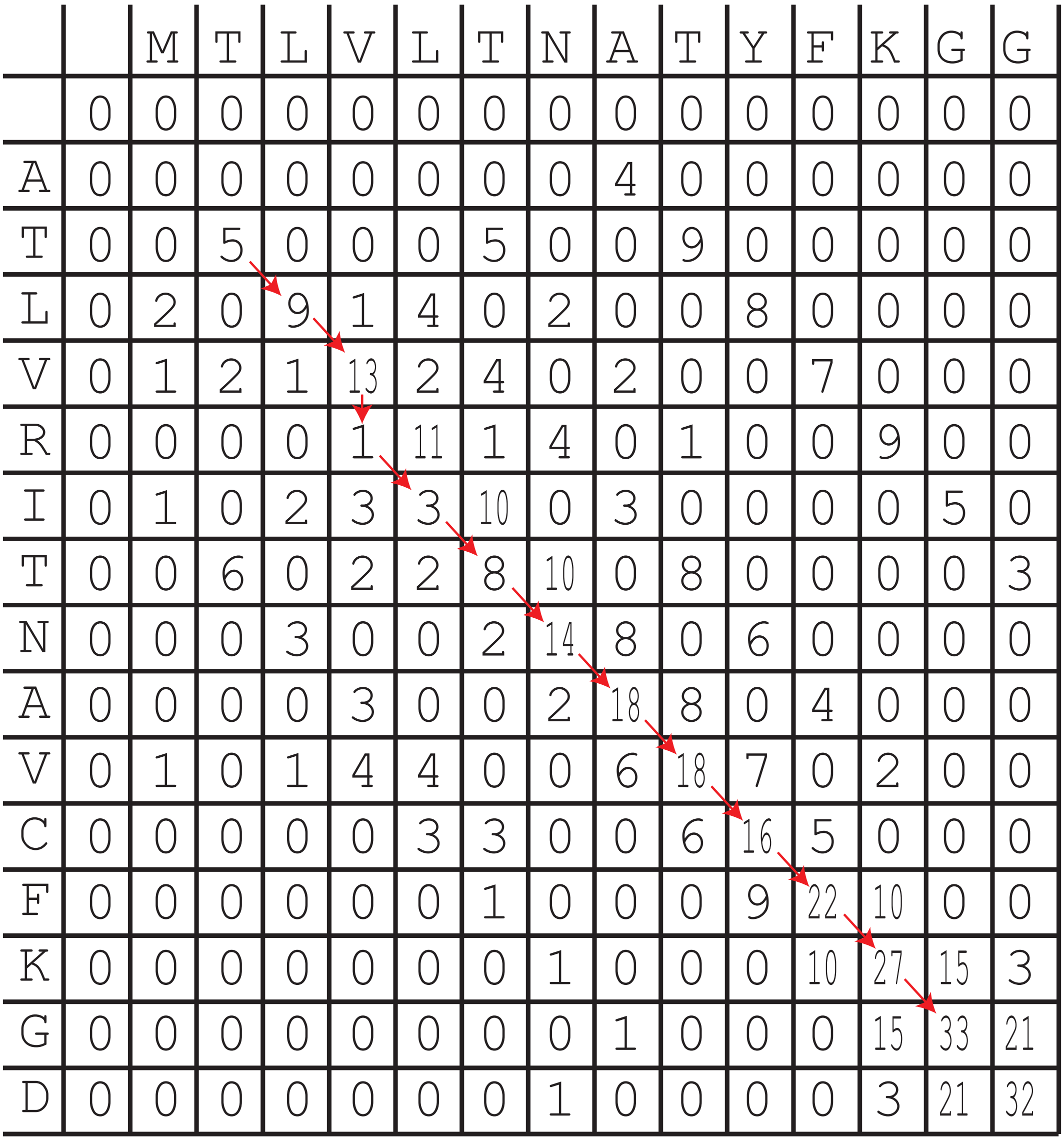
Let $S[i,j]$ be the score of the highest scoring local path ending at $[i,j]$ between $A[1,i]$ and $B[1,j]$. Let $s(a_i,b_j)$ be the substitution score, when a_i changes to b_i. This algorithm applies linear gap penalty, which is gl, where g is the penalty of a gap and l is gap length. By definition, we have $S[0,0] = 0$, $S[i,0] = 0$, and $S[0,j] = 0$. With these initializations, $S[i,j]$ for $i \in 1,2,...,M$ and $j \in 1,2,...,N$ can be computed following recurrence.

$$S[i,j] = \max \begin{cases} 0 \\ S[i-1,j] - g \\ S[i,j-1] - g \\ S[i-1,j-1] + s(a_i,b_j) \end{cases} \tag{2.3}$$

The largest value of $S[i,j]$ is the score of the optimal alignment between A and B. If the optimal alignment is also needed, the path for the alignment is computed by a traceback algorithm that recovers this alignment using S. An example of S for sequences in Fig. 2.3 is shown in Fig. 2.4.

2.3.2 Gotoh Algorithm

For aligning biological sequences, affine gap penalties are considered more appropriate than linear gap penalties discussed in Section 2.3.1. An affine gap penalty is defined as $o + el$, where o is the penalty of gap opening, e is the penalty of gap extensions, and l is the length of the gap. This gap penalty is most widely used in biological analyses. The Gotoh algorithm [17] was proposed to apply affine gap penalties in computing local alignment. The Gotoh algorithm used two more matrices, I and D, to distinguish gap extensions from gap openings. $I[i,j]$ is the score of the optimal alignment between $A[1,i]$ and $B[1,j]$ ending with an insertion. $D[i,j]$ is the score of the optimal alignment between $A[1,i]$ and $B[1,j]$ ending with a deletion. Let $S[i,j]$ be the score of the optimal alignment between $A[1,i]$ and $B[1,j]$. By definition, we have $S[0,0] = 0$,



		M	T	L	V	L	T	N	A	T	Y	F	K	G	G
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	0	0	0	0	0	0	0	0	4	0	0	0	0	0	0
T	0	0	5	0	0	0	5	0	0	9	0	0	0	0	0
L	0	2	0	9	1	4	0	2	0	0	8	0	0	0	0
V	0	1	2	1	13	2	4	0	2	0	0	7	0	0	0
R	0	0	0	0	1	11	1	4	0	1	0	0	9	0	0
I	0	1	0	2	3	3	10	0	3	0	0	0	0	5	0
T	0	0	6	0	2	2	8	10	0	8	0	0	0	0	3
N	0	0	0	3	0	0	2	14	8	0	6	0	0	0	0
A	0	0	0	0	3	0	0	2	18	8	0	4	0	0	0
V	0	1	0	1	4	4	0	0	6	18	7	0	2	0	0
C	0	0	0	0	0	3	3	0	0	6	16	5	0	0	0
F	0	0	0	0	0	0	1	0	0	0	9	22	10	0	0
K	0	0	0	0	0	0	0	1	0	0	0	10	27	15	3
G	0	0	0	0	0	0	0	0	1	0	0	0	15	33	21
D	0	0	0	0	0	0	0	1	0	0	0	0	3	21	32

Figure 2.4: An example of S in the Smith-Waterman algorithm when we used BLOSUM62 and $g = -12$. Red arrows show the path of the optimal alignment.

$S[i,0] = 0$, $S[0,j] = 0$, $I[0,j] = -\infty$, and $D[i,0] = -\infty$. With these initializations, $I[i,j]$, $D[i,j]$ and $S[i,j]$ for $i \in 1, 2, ..., M$ and $j \in 1, 2, ..., N$ can be computed following recurrence.

$$I[i,j] = \max \begin{cases} I[i,j-1] - e \\ S[i,j-1] - o - e \end{cases} \tag{2.4}$$

$$D[i,j] = \max \begin{cases} D[i-1,j] - e \\ S[i-1,j] - o - e \end{cases} \tag{2.5}$$

$$S[i,j] = \max \begin{cases} 0 \\ I[i,j] \\ D[i,j] \\ S[i-1,j-1] + s(a_i, b_j) \end{cases} \tag{2.6}$$

The largest value of $S[i,j]$ is the score of the optimal alignment between A and B. If the optimal alignment is also needed, the path for the alignment is computed by a traceback algorithm that recovers this alignment using S in the same way as the Smith-Waterman algorithm.

2.3.3 SSEARCH

SSEARCH [41] is a sequence homology search tool based on the Smith-Waterman and Gotoh algorithms. Therefore, sequences with the highest similarity to a query sequence can be always found in databases using SSEARCH. However, most of current biological analyses that use protein sequence homology search need to utilize large sequence databases and high number of queries. Therefore, this tool does not meet current demands.

2.4 Seed-and-Extend Strategy

The seed-and-extend strategy provides a fast method to perform sequence homology search and is, therefore, frequently employed in current sequence homology search tools [4, 5, 22, 35, 58, 59]. The Smith-Waterman algorithm and Gotoh algorithm need long computation time, especially when large sequence data are used. To solve this problem, the seed-and-extend strategy applies heuristics; therefore, sequence alignment computed by this algorithm may not be optimal, but is sufficiently accurate.

The strategy consists of two main steps: seed search and extension. In seed search, subsequences from query and database sequences are constructed, and pairs of exactly matched or similar subsequences between a query and database are searched. These pairs of subsequences are called seeds. In extension, sequence alignment is performed for the areas flanking the seeds and their alignment scores are calculated.

The seed-and-extend strategy is faster than the Smith-Waterman and Gotoh algorithms, because it uses only the regions around seeds instead of entire sequence. Seed search finds similar regions within query and database sequences by searching short subsequences. Because sequences with similar regions often display structural and functional homology, these regions are often matched in optimal alignments. Therefore, the seed-and-extend strategy reports the alignment quite similar to an optimal alignment.

2.4.1 BLAST

The BLAST algorithm was proposed at 1990 by Altschul *et al.* [4]. Currently, BLAST, which employs the seed-and-extend strategy, is widely used in various biological fields [53, 23, 13, 42, 11]. Two versions of the BLAST algorithm have been proposed. In the first version, which computes local alignment without gaps [4], seed search identifies exact or similar subsequence pairs as seeds in both query and database sequences. A subsequence has a fixed length and is called "word" in the BLAST algorithm. Words are searched by the Aho-Corasick algorithm [3] within BLAST, which finds strings in a text by using a deterministic finite automaton (DFA). BLAST constructs a DFA from query sequences. By the Aho-Corasick algorithm, words can be found in $O(|Q| + |DB| + z)$, where $|Q|$ is query length, $|DB|$ is database sequence length and z is the number of word occurrences. For DNA sequences, an exact match between words is used. For protein sequences, BLAST employs neighborhood words, which are subsequences similar to each word [4]. A vast variety of neighborhood words are used in seed search to increase search sensitivity.

The extension which calculates the alignment without gaps is called ungapped extension. The calculation of ungapped extension is faster than that of gapped extention, because ungapped extension does not consider insertions and deletions. However, it is still a more extensive computation process compared with the other processes and needs to be accelerated. For this, the X-dropoff has been introduced [4]. Ungapped extension process is terminated, when the score of the current state is dropped more than the X-dropoff value below the current maximum score.

The first version of BLAST [4] was fast, but many analyses require considering gaps. Therefore, the second version of BLAST (also called Gapped BLAST) [5] that could compute local alignment with gaps has been developed. The second version uses the two-hit search method to reduce useless seeds in protein sequence homology search. In two similar protein sequences, there are a significant number of seeds within a relatively short distance on the same diagonal. When matches repeat in the alignment, the seeds are on the same diagonal, as shown in Fig. 2.4. Therefore, this BLAST version takes the seeds into the next step when two non-overlapping seeds occur within a given distance from each other on the same diagonal (Fig. 2.5). In the figure, no seeds are on the diagonal with $Seed_0$; therefore, this seed is not taken to the next step. On the other hand, $Seed_1$ and $Seed_2$ are on the same diagonal and the distance between them is short enough; therefore, they are merged into one and taken to the next step.

Gapped extension takes considerable computation time, which can be decreased by reducing the number of seeds produced by seed search. However, it causes a loss of a significant number of seeds, which can result in lower sensitivity. Therefore, this BLAST version performs ungapped extension with the X-dropoff. If the ungapped extension score of a seed excesses a threshold, the seed is extended with gaps. To accelerate gapped extension, X-dropoff is also used in this version of BLAST. Gapped extension is terminated when the score of the current state is dropped more than the X-dropoff value below the found maximum score. Fig. 2.6 shows an example of gapped extension with X-dropoff. Green regions correspond to the cells with the score calculated in gapped extension with X-dropoff. Both Smith-Waterman and Gotoh algorithms require scores of the entire region to be calculated. However, with the introduction of X-dropoff, the regions requiring score calculation become smaller.

As the heuristic approach, BLAST is faster than SSEARCH and has high search sensitivity. However, BLAST search speed has become increasingly insufficient for current biological analyses of sequence homology searches based on large sequence data. Therefore, novel sequence homology search algorithms, such as BLAT and RAPSearch, have been proposed.

2.4.2 BLAT

The BLAT algorithm was proposed at 2002 by Kent [22] to solve the problem of search speed encountered by using BLAST. As a result, BLAT is approximately 50 times faster than BLAST in protein sequence homology search [22]. BLAT also applies the seed-and-extend strategy, but builds a hash table of a database and stores in mem-

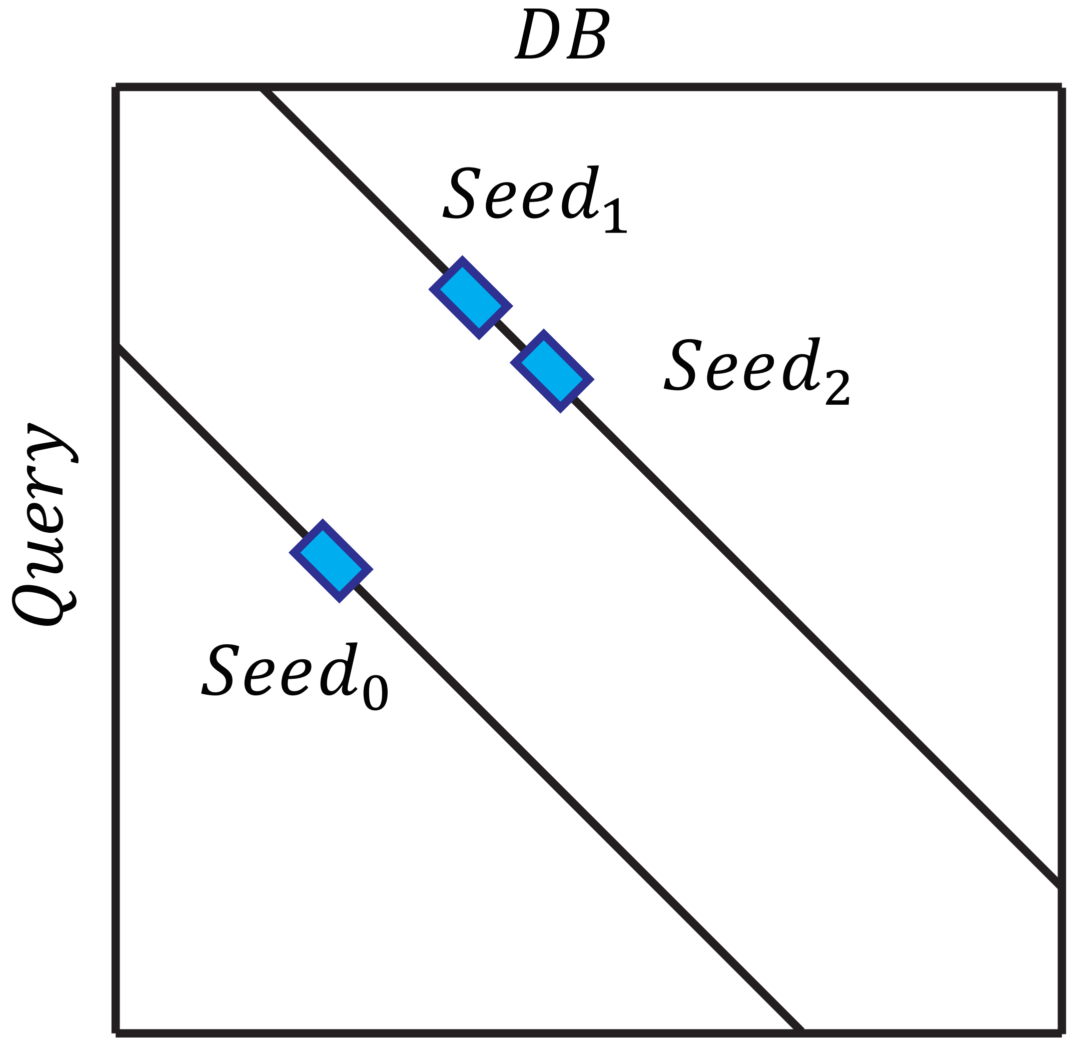


Figure 2.5: Explanation of two hits method.

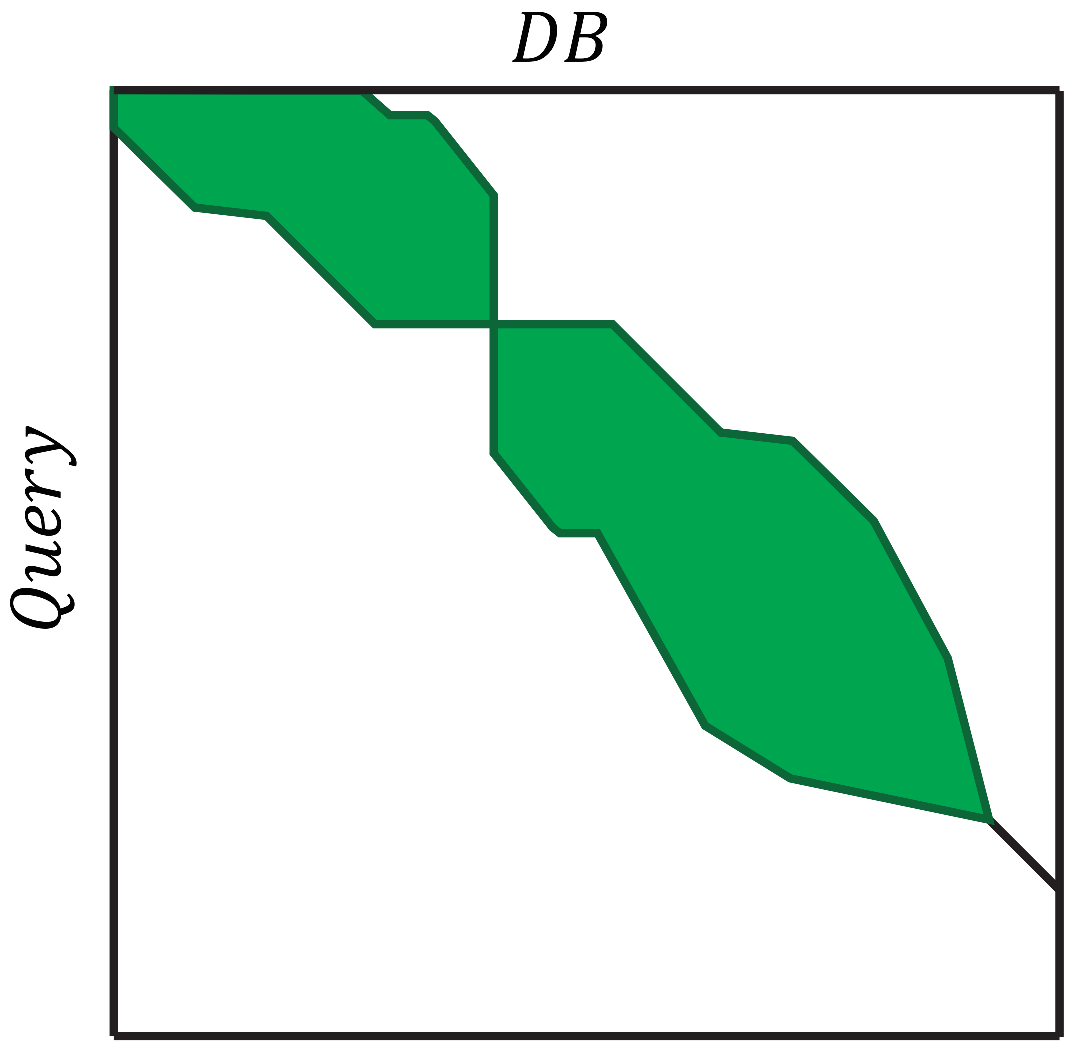


Figure 2.6: Example of gapped extension with the X dropoff.

ory. The table consists of non-overlapping fixed-length subsequences with corresponding positions in the database, thus reducing memory size and accelerating sequence homology search; it requires a few GB of RAM and is affordable for many users. However, the sensitivity of BLAT search is much lower compared to that of BLAST and is insufficient for several analyses, including metagenomics.

2.4.3 RAPSearch

RAPSearch [58] first proposed in 2011 [58], is one of the state-of-the-art protein sequence homology search tools for metagenomic analysis, which performs faster and more sensitive homology search than BLAT. RAPSearch also applies seed-and-extend strategy, and uses a reduced amino acid alphabet [38] and a suffix array [36] in seed search. In addition, the second version of RAPSearch has been improved to use hash tables instead of suffix arrays, making it more memory-efficient [59]. RAPSearch is approximately 20–90 times faster than BLAST, and has higher search sensitivity than BLAT. However, RAPSearch utilizes the frequencies of database subsequences in seed search, which may change the results when sequences are added to the database. Considering continuous increase in analyzed sequence data due to the advancements in sequencing technologies, a faster protein sequence homology search tool is still required.

Chapter 3

A Protein Sequence Homology Search Algorithm Using a Query Suffix Array and a Database Suffix Array

3.1 Introduction

In this study, we developed a fast protein sequence homology search algorithm using suffix arrays [36] of both queries and database sequences for its seed search process. We used a seed search method relying on a similarity-based optimal length. In the algorithm, only seeds with a sufficient match score are searched, based on a given score matrix. Thus, the algorithm can effectively exclude seeds with sufficient length but insufficient similarity. We designated this algorithm as GHOSTX.

3.2 Methods

3.2.1 Overview of GHOSTX Algorithm

GHOSTX adopts the seed-and-extend strategy used in BLAST. GHOSTX consists of three main steps: a seed search, an ungapped extension, and a gapped extension. The flow of GHOSTX is shown in Fig. 3.1. Initially, GHOSTX finds seeds that are subsequences of database sequences similar to the subsequences of a query sequence. Next, GHOSTX makes alignments by extending those seeds without gaps, and then

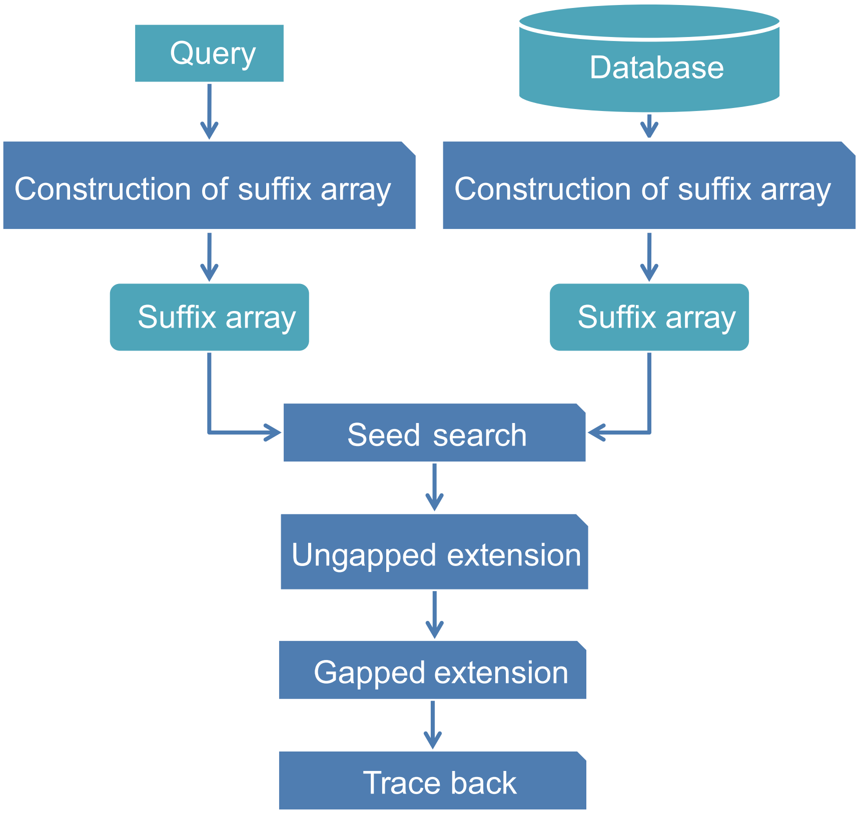


Figure 3.1: The flow of GHOSTX.

similar, nearby seeds are brought together by a chain filter. Finally, GHOSTX makes alignments from seeds with gaps. In BLAST, the gapped extension step requires heavy calculation, but BLAST efficiently decreases the number of gapped extension candidates through its seed search and ungapped extension steps. As a result, the seed search and the ungapped extension steps are the most computationally intensive parts of BLAST. The seed search and the ungapped extension steps consume approximately 75% of the computation time of BLAST, while approximately 20% of the time is spent on the gapped extension [54]. Thus, reducing the computation time for the seed search and ungapped extension steps is effective for achieving acceleration. To accelerate the search seed step, GHOSTX uses suffix arrays for both the query sequences and the database sequences.

Our seed search method using a suffix array effectively reduces the computation time of the seed search step. As a result, the ungapped extension step then becomes the bottleneck. Thus, for further acceleration, we have to decrease the number of ungapped extensions. It would be easy to decrease the number of ungapped extension candidates by using longer seeds. However, if this is done, significant matches can be missed, and search sensitivity becomes lower. Consequentially, a sophisticated method is required for accelerating search speed, while still maintaining search sensitivity. Therefore, GHOSTX does not fix the length of a seed in the seed search step, but rather it extends the length until the matching score exceeds a given threshold. In comparison, BLAST searches with seeds of fixed lengths, and if one seed is discovered near another, BLAST performs ungapped extensions around it. BLAST seed hits with low matching scores using fixed length seeds, such as an exact match of “AAA”, whose score is only 12 based on the BLOSUM62 score matrix, are treated equally with seed hits with high matching scores, such as an exact match of “WWW”, whose score is 33. However, hits with lower scores tend to be false. Consequently, GHOSTX extends such seeds to check whether they are reliable, thus GHOSTX can use a higher score threshold than BLAST, without losing its search sensitivity. As a result, GHOSTX can reduce the number of ungapped extensions and gapped extensions needed, thereby reducing computation time after the initial seed search step.

3.2.2 Suffix Array

A suffix array is the list of indexes of all suffixes of a string in a lexicographically sorted order. A text $T[0, n] = t_0 ... t_{n-1}$ is a sequence of symbols and the length of T is $|T| = n$. Each symbol is an element of an alphabet Σ ($|\Sigma|$ of protein is 20). $T[i] = t_i$ and $T[i, i+j] = t_i ... t_{i+j-1}$ are subsequences. The suffix array of T is SA_T, that is an array of pointers to all the suffixes of T in lexicographical order. Therefore, if $i < j$, then $T[SA_T[i], n] < T[SA_T[j], n]$. A suffix array can be constructed in linear time. An exact search based on a binary search for pattern, whose length is m, can be performed as $O(m \log(n))$ with the suffix array of T.

3.2.3 Seed Search

For two suffix arrays, we can find all the local matches using dynamic programming [15]. However, calculating all alignments using dynamic programming requires a huge amount of computation time. In GHOSTX, therefore, we introduce two methods to prune the search space.

Here, the sequences $DB_0, DB_1, ..., DB_{N-1}$ in a database are connected with inserting delimiters to transform them into a long single sequence $DB = \#DB_0\#DB_1\#...DB_{N-1}$ (marked by the special symbol #). SA_{DB} is the suffix array of DB, and SA_Q is the sequence of query Q. The pair of subsequences DB and Q, $\{DB[i, i+l], Q[j, j+l]\}$ is the seed. Here, we want to find a seed whose score is more than the threshold T_{seed} based on these two suffix arrays. Fig. 3.2 shows the pseudo-code of the seed search method, and Fig. 3.3 shows a pseudo-code for the search method of one character using a suffix array. In Fig. 3.2, sp_Q, ep_Q, sp_{DB} and ep_{DB} are positions on SA_Q and SA_{DB}, and GHOSTX gets the positions of subsequences from suffix arrays by using these positions. If the score of a pair of subsequences $\{X_{DB}, X_Q\}$ exceeds threshold T_{seed}, GHOSTX keeps the pair as a seed (line 22 in Fig. 3.2); otherwise, GHOSTX checks all pairs of extended subsequences $\{X_{DB}c', X_Qc\}$ $(c, c' \in \Sigma)$ (line 25 in Fig. 3.2). Thus, the maximum number of new pairs of subsequences is $|\Sigma|^2$. Using the suffix arrays of a query and a database, GHOSTX can find a subsequence efficiently. Fig. 3.4 shows the example for the seed search. If {A, A} is found, GHOSTX searches the query sequence and the sequences in a database for extended subsequences AA, AR, ..., AV. And then, GHOSTX checks all pairs of extended subsequences that are found {AA, AA}, {AA, AR},..., {AV, AV}. GHOSTX repeats this step. However, the search takes a long time if the max seed length $length_{max}$ is large, because the size of the seed search space is $O(|\Sigma|^{2length_{max}})$. Thus, the search space must be pruned.

GHOSTX uses two methods to prune the search space (line 24 in Fig. 3.2). First, let $score_{max}$ be the sum of the exact match score of all query subsequence characters (line 16 in Fig. 3.2), score be the score of the pair of the query and database subsequence (line 20 in Fig. 3.2), and D be the upper limit of $score_{max} - score$. If $score \leq score_{max} - D$, GHOSTX does not extend the subsequence in the pair. For example, if GHOSTX checks {AA, AR} and uses BLOSUM62 score matrix, $score_{max}$ of this pair is $4 + 5 = 9$ and score of this pair is $4 - 1 = 3$. If $D = 4$, GHOSTX does not extend the subsequences in this pair. Second, if the score of a subsequence pair is not more than 0, GHOSTX does not extend it. If $x < y < z$, the score of the subsequence pair $\{DB[i, i+y], Q[j, j+y]\}$ is less than 0, and the score of the subsequence pair $\{DB[i, i+z], Q[j, j+z]\}$ exceeds the threshold T_{seed}, then GHOSTX finds another pair $\{DB[i+x, i+z], Q[j+x, j+z]\}$ whose score exceeds T_{seed}. Therefore, GHOSTX examines only those pairs with scores greater than 0. For example, if GHOSTX checks {A, R} and uses the BLOSUM62 score matrix, the score of this pair is -1. Therefore, GHOSTX does not extend the subsequences in this pair. Consequently, GHOSTX can find long seeds quickly using these pruning methods. In addition, GHOSTX uses a depth-first search for the implementation of this algorithm to save memory. With a breadth-first search, the depth of the recursion in a seed search is proportional to the exponential of $length_{max}$, and thus it is difficult to check all pairs of subsequences.

```
$SeedSearchCore(Q, SA_Q, DB, SA_{DB}, sp_Q, ep_Q, sp_{DB}, ep_{DB}, score_{max}, score, length)$
if $length < length_{max}$ then
    Let $result_Q$ be the array whose length is $|\Sigma|$.
    Let $result_{DB}$ be the array whose length is $|\Sigma|$.
    Let $S$ be the score matrix.
    for all $c \in \Sigma$ do
        $sp, ep \Leftarrow SASearchNextCharacter(Q, SA_Q, sp_Q, ep_Q, c, length)$
        $result_Q[c] \Leftarrow sp, ep$
    end for
    for all $c \in \Sigma$ do
        $sp, ep \Leftarrow SASearchNextCharacter(DB, SA_{DB}, sp_{DB}, ep_{DB}, c, length)$
        $result_{DB}[c] \Leftarrow sp, ep$
    end for
    for all $c \in \Sigma$ do
        $sp, ep \Leftarrow result_Q[c]$
        if $sp \leq ep$ then
            $score'_{max} \Leftarrow score_{max} + S[c, c]$
            for all $c' \in \Sigma$ do
                $sp', ep' \Leftarrow result_{DB}[c']$
                if $sp' \leq ep'$ then
                    $score' \Leftarrow score + S[c, c']$
                    if $T_{seed} \leq score'$ then
                        store $sp, ep, sp'ep'$
                        continue
                    else if $score' > score'_{max} - D \bigwedge score' > 0$ then
                        $SeedSearchCore(Q, SA_Q, DB, SA_{DB},$
                          $sp, ep, sp', ep', score'_{max}, score', length + 1)$
                    end if
                end if
            end for
        end if
    end for
end if
$SeedSearch(Q, SA_Q, DB, SA_{DB})$
$SeedSearchCore(Q, SA_Q, DB, SA_{DB}, 0, |Q| - 1, 0, |DB| - 1, 0, 0, 0)$
```

Figure 3.2: Pseudo-code for seed search algorithm using suffix arrays.

```
$SASearchNextCharacter(T, SA, sp, ep, c, length)$
$sp_{tmp} \Leftarrow sp - 1$
$ep_{tmp} \Leftarrow ep$
while $sp_{tmp} + 1 < ep_{tmp}$ do
    $m \Leftarrow \frac{sp_{tmp}+ep_{tmp}}{2}$
    if $T[length + SA[m]] < c$ then
        $sp_{tmp} \Leftarrow m$
    else
        $ep_{tmp} \Leftarrow m$
    end if
end while
if $T[length + SA[ep_{tmp}]] \neq c$ then
    $sp_{tmp} \Leftarrow 1$
    $ep_{tmp} \Leftarrow 0$
    return $sp_{tmp}, ep_{tmp}$                              ▷ not found $c$
end if
$sp \Leftarrow ep_{tmp}$
$sp_{tmp} \Leftarrow sp$
$ep_{tmp} \Leftarrow ep + 1$
while $sp_{tmp} < ep_{tmp} - 1$ do
    $m = \frac{sp_{tmp}+ep_{tmp}}{2}$
    if $T[length + SA[m]] > c$ then
        $ep_{tmp} \Leftarrow m$
    else
        $sp_{tmp} \Leftarrow m$
    end if
end while
if $T[length + SA[sp_{tmp}]] \neq c$ then
    $sp_{tmp} \Leftarrow 1$
    $ep_{tmp} \Leftarrow 0$
    return $sp_{tmp}, ep_{tmp}$                              ▷ not found $c$
end if
$ep \Leftarrow sp_{tmp}$
return $sp, ep$
```

Figure 3.3: Pseudo-code for search algorithm using a suffix array.

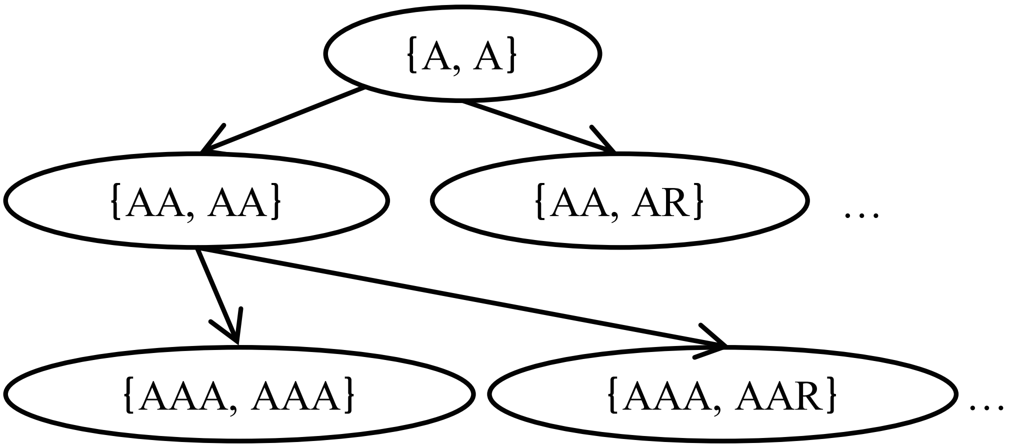


Figure 3.4: An example seed search.

However, the depth of recursion in *SeedSearchCore* is $O(length_{max}|\Sigma|^2)$ based on a depth-first search. Therefore, using this depth-first search strategy can save memory.

Even when using a binary search, this seed search approach was originally a bottleneck in GHOSTX. To accelerate the process, GHOSTX searches parts of seeds using an auxiliary data structure. GHOSTX stores the search results for all subsequences whose length is less than 6 on a table after the construction of the database index. In the seed search, GHOSTX can find the search result for a subsequence without performing a binary search on the suffix array of a database, if the length of the subsequence is shorter than 6. If we store the search results for longer subsequences, we can make the process more efficient. However, the table requires more memory depending on the length of the subsequence. If the length $length_{subsequence}$ of a subsequence is extended by 1, the size of table increases by $O(|\Sigma|^{length_{subsequence}})$. Thus, GHOSTX only stores the search results for the subsequence whose length is less than 6.

3.2.4 Ungapped Extension and Chain Filtering

Decreasing the number of seeds is critical for the acceleration of a search. However, higher T_{seed} values cause an increase in the number of significant hits missed, so it is difficult to use high T_{seed} values without sacrificing sensitivity. Therefore, GHOSTX performs an ungapped extension, which extends seeds without any gaps and excludes low-score extended seeds, after the seed search step, as in BLAST. In the ungapped extension step, GHOSTX uses X-dropoff [4].

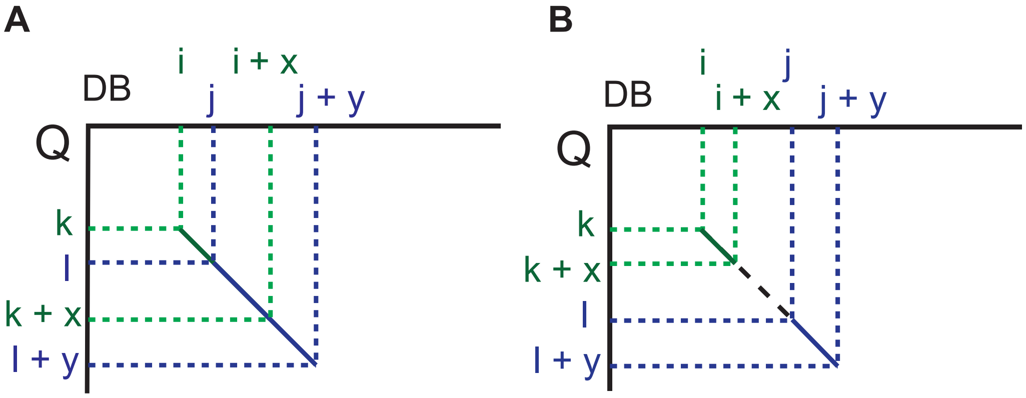


Figure 3.5: Conditions for reducing seeds in chain filtering.

Some seeds may overlap with others after the seed search and the ungapped extension step. In particular, if there is a sequence highly similar to a query in the database, many seeds that overlap with others are found, and almost identical alignments are often obtained from these overlapped seeds. Thus, it is necessary to merge such overlapped seeds to reduce the number of gapped extensions. Therefore, GHOSTX uses a chain filtering technique. There are two cases in which the seeds are filtered out, as shown in Fig. 3.5. First, if two seeds $\{DB[i, i+x], Q[k, k+x]\}$ and $\{DB[j, j+y], Q[l, l+y]\}$ overlap as shown in Fig. 3.5A, GHOSTX combines these overlapped seeds together into one. Second, if two seeds $\{DB[i, i+x], Q[k, k+x]\}$ and $\{DB[j, j+y], Q[l, l+y]\}$ do not overlap but the score exceeds the dropoff used for the ungapped extension step, as shown in Fig. 3.5B, GHOSTX also merges the overlapped seeds.

3.2.5 Gapped Extension

Those seeds judged as meaningful by the chain filter are extended with gaps. In the gapped extension, GHOSTX employs X-dropoff in the same way as BLAST [5]. In BLAST gapped extension, the process stops if the score is much lower than the best score, which saves computation time. GHOSTX also employs this technique and uses the same X-dropoff.

3.2.6 Database Division

GHOSTX requires a large amount of memory in its protein sequence homology search. Memory size depends on database size. However, computing systems generally have relatively small memory sizes compared with current database sizes. Therefore, GHOSTX divides a database into several chunks, each of whose size is l_{DB}, before it constructs its indexes. GHOSTX sequentially searches each database chunk, and merges its results with the results of previous chunk searches, when this chunk division is performed before the construction of its database indexes. GHOSTX dramatically reduces working memory requirements using this approach.

3.3 Results and Discussion

3.3.1 Datasets and Conditions

To evaluate the performance of our tool, we compared its search sensitivity and computation time to NCBI BLASTX (version 2.2.28+), BLAT (version 34 standalone) and RAPSearch (version 2.12). We used the binaries of BLASTX and BLAT downloaded from web sites. We used RAPSearch compiled with GCC (version 4.3.4) and the -O3 optimizing option. We also compiled GHOSTX using GCC with the -O3 optimizing option. We used a database obtained from KEGG [20, 19] GENES protein sequences as of May 2013. This database contained approximately 10,000,000 protein sequences, with a total size of approximately 3,600,000,000 residues (3.9 GB). We also used another database obtained from NCBI nr that contained approximately 25,000,000 sequences, approximately 8,600,000,000 residues (14.8 GB), to check our algorithm's dependency on database size. For the query sequences, we used 2 query sets: one from human microbiome metagenomic sequences (SRS011098), and the other of soil microbiome metagenomic sequences (SRR444039). SRS011098 was obtained from the Data Analysis and Coordination Center for Human Microbiome Project (HMP-DACC) [52] web site. We used the whole metagenomic shotgun sequencing data from SRS011098. SRR444039 was obtained from the DNA Data Bank of Japan (DDBJ) Sequence Read Archive. 10,000 randomly selected DNA short reads were used from both sets, SRS011098 and SRR444039. We performed the analyses on a workstation with two Intel Xeon 5670 processors (2.93 GHz, 6 cores) and 54 GB of memory.

3.3.2 Relationship between GHOSTX Parameters and Sensitivity and Computation Time

GHOSTX has two parameters for its seed search, threshold of the seed search T_{seed}, and an upper mismatch score D. These parameters affect the performance of GHOSTX. Therefore, we first searched for optimal parameters. To determine the best parameters, we used $T_{seed} = 22, 24, 26, 28, 30, 32$ and D = 1, 4, 7. To evaluate search sensitivity, we used the search results obtained using local alignment by SSEARCH [41] as the correct answer. Because it does not use any heuristics, it returns an optimal local alignment. We analyzed the performance of the particular parameter in terms of the fraction of its results that corresponded to the correct answers. When the subject sequences that had the highest score by SSEARCH and each particular method corresponded on each query, the query was deemed correct. Table 3.1 shows the sensitivity and computation time of each different parameter. As shown in the table, when T_{seed} is large or D is small, the sensitivity of GHOSTX is low and its computing speed is fast. This is because the search space in the seed search is small and the number of seeds is small. However, when T_{seed} is small or D is large, the sensitivity of GHOSTX is high and its computing speed is slow. This is because the search space in the seed search is large and the number of seeds is large. We selected $T_{seed} = 30$ and $D = 4$ as default parameters that have a good balance between sensitivity and computation time. We used those parameters in the following evaluations.

3.3.3 Evaluation of Search Sensitivity

To evaluate search sensitivity, we evaluated sensitivity the same way as we evaluated the relationship between GHOSTX parameters and their sensitivity and computation time. To evaluate the software, we executed the BLASTX program with the command line options "-outfmt 6 -comp_based_stats 0", which instructed the program to output in tabular format, without using composition-based statistics [6], because composition-based statistics are not available in SSEARCH. We used default parameters for the other options. The BLAT program does not include a function to translate DNA reads to protein sequences. Therefore, we translated the DNA reads into protein sequences based on all six potential frames using a standard codon table before executing BLAT. We executed the BLAT program with the command line option "-q=prot -t=prot -out=blast8", which instructed the program to run the queries and database as protein sequences, and to output data in the BLAST tabular format. We could not execute BLAT when we used nr as a database because our machine has insufficient memory for

Table 3.1: Relationship between GHOSTX parameters and sensitivity and computation time. The first, second third, and fourth columns show T_{seed}, D , the sensitivity, and the computation time.The sensitivity is calculated as the ratio of correctly searched queries whose E-values $< 10^{-5}$.

T_{seed}	D	Sensitivity	Computation time (sec.)
22	1	0.978	615.0
22	4	0.985	2099.0
22	7	0.984	7066.6
24	1	0.977	474.4
24	4	0.984	1284.4
24	7	0.985	4312.9
26	1	0.976	316.1
26	4	0.984	779.9
26	7	0.985	3035.2
28	1	0.971	199.4
28	4	0.982	518.4
28	7	0.985	2472.6
30	1	0.963	151.2
30	4	0.980	401.9
30	7	0.985	2472.6
32	1	0.957	119.5
32	4	0.977	344.1
32	7	0.984	2101.9

the execution. Therefore, we only executed BLAT with KEGG GENES. We executed the RAPSearch program with 2 cases. One case used the default options and the other used the command line option "-a T", which instructed the program to perform a fast mode search. For GHOSTX, we used the following parameters: threshold of the seed search $T_{seed} = 30$, upper mismatch score $D = 4$, and size of the database chunk $l_{DB} = 2$ GB. The other parameters used are the same as BLASTX defaults. In Fig. 3.6, GHOSTX shows lower sensitivity than BLASTX, especially for those hits with E-values above 10^{-5}. However, hits with such high E-values are not used in practice because they are unreliable. For instance, Trunbaugh *et al.* used hits with E-values below $1.0 \times E^{-5}$ [53], and Kurokawa *et al.* used hits with E-values below $1.0 \times E^{-8}$ [23]. We used the single-value search sensitivity calculated as the ratio of correctly searched queries to all queries with the E-values $< 1.0 \times E^{-5}$ to compare search sensitivity of GHOSTX with that of other tools. Table 3.2 showing search sensitivity for each program indicates

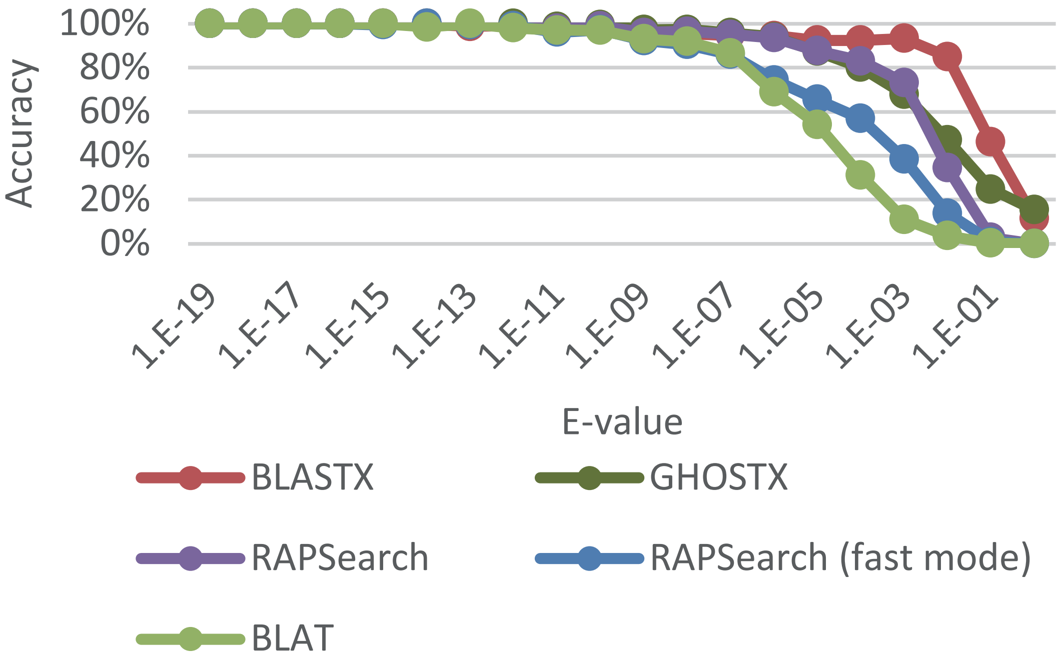


Figure 3.6: Search sensitivity of each tool with KEGG GENES. The vertical axis shows the percentage of correct answers that correspond to the correct answers for each method. The horizontal axis shows the E-value of the alignments.

Table 3.2: Search sensitivity. The search sensitivity is calculated as the ratio of correctly searched queries whose E-values $< 10^{-5}$.

	Sensitivity
GHOSTX	0.98
RAPSearch	0.97
RAPSearch in fast mode	0.93
BLAT	0.93
BLASTX	0.97

that the sensitivity of GHOSTX was clearly better than that of BLAT and RAPSearch in fast mode, and almost equal to that of RAPSearch and BLASTX. Therefore, we believe that GHOSTX has search sensitivity sufficient for most practical analyses.

3.3.4 Evaluation of Computation Time

We ran each method with the same commands as for the evaluation of search sensitivity to measure computation time. We used 2 query sets, 10,000 randomly selected

Table 3.3: Computation time with SRS011098 and KEGG GENES (3.9 GB). The first, second, and third columns show the name of each program, the computation time, and the acceleration in processing speed relative to BLASTX using 1 thread, respectively.

	Computation time (sec.)	Acceleration ratio
GHOSTX	401.9	152.6
RAPSearch	649.5	94.4
RAPSearch in fast mode	91.2	672.2
BLAT	1409.7	43.5
BLASTX	61314.1	1.0

Table 3.4: Computation time with SRR444039 and KEGG GENES (3.9 GB). The first, second, and third columns show the name of each program, the computation time, and the acceleration in processing speed relative to BLASTX using 1 thread, respectively.

	Computation time (sec.)	Acceleration ratio
GHOSTX	362.7	151.8
RAPSearch	553.2	99.5
RAPSearch in fast mode	64.8	849.6
BLAT	1265.3	43.5
BLASTX	55045.0	1.0

DNA short reads from SRS011098 and SRR444039, and we used KEGG GENES as our database. Table 3.3 and Table 3.4 show the computation time for each program. As shown with each query set, GHOSTX showed accelerations of approximately 153 and 152 times with respect to BLASTX, and approximately 3.5 and 3.5 times with respect to BLAT. Additionally, GHOSTX was approximately 1.6 and 1.5 times faster than RAPSearch. GHOSTX outperforms BLASTX in reducing computation time. The computation time acceleration is caused by the use of a suffix array for its seed search and ungapped extension steps. GHOSTX was slower than RAPSearch in fast mode. However, the sensitivity of RAPSearch in fast mode is clearly lower than GHOSTX.

We also checked the dependency of computation time on the database size for each program by using a larger database. Table 3.5 and Table 3.6 show the computation times and accelerations for NCBI nr. GHOSTX showed a better acceleration ratio against BLASTX, as compared with the KEGG GENES database (approximately 165 times and 131 times, respectively). This indicates that these programs can efficiently handle an increase in database size in the future. In contrast to GHOSTX's acceleration

Table 3.5: Computation time with SRS011098 and NCBI nr (14.8 GB). The first, second, and third columns show the name of each program, the computation time, and the acceleration in processing speed relative to BLASTX using 1 thread, respectively.

	Computation time (sec.)	Acceleration ratio
GHOSTX	1020.1	165.2
RAPSearch	1564.4	107.7
RAPSearch in fast mode	223.8	752.8
BLAT	N/A	N/A
BLASTX	168488.0	1.0

Table 3.6: Computation time with SRR444039 and NCBI nr (14.8 GB). The first, second, and third columns show the name of each program, the computation time, and the acceleration in processing speed relative to BLASTX using 1 thread, respectively.

	Computation time (sec.)	Acceleration ratio
GHOSTX	1003.5	130.8
RAPSearch	1404.1	93.4
RAPSearch in fast mode	223.8	586.2
BLAT	N/A	N/A
BLASTX	131213.3	1.0

as compared with BLASTX, GHOSTX's acceleration ratio was 1.5 and 1.4 times as fast as RAPSearch with the larger database, and almost the same when using the smaller KEGG GENES database. Thus, the acceleration ratio of GHOSTX to RAPSearch would not significantly change regardless of the size of a database.

Additionally, to compare the computation time of each step, seed search, ungapped extension and gapped extension in GHOSTX, we obtained execution profiles of BLASTX and GHOSTX. These functions of GHOSTX are not used inline expansion to obtain execution profile. These execution profiles were obtained from the calculation using 10,000 DNA short reads in SRS011098 as queries and KEGG GENES as a database. Table. 3.7 shows execution profiles of GHOSTX and BLASTX. Seed search of GHOSTX was faster than that of BLASTX. Because, GHOSTX uses effective seed search with suffix arrays of query and database. Currently, seed search is one of the primary bottlenecks in protein sequence homology searches. Thus, we consider that this effective speed up in seed search contributes to significant increase in search speed observed.

We also measured the computation time of preprocessing, including database index-

Table 3.7: Computation time of seed search, ungapped extension and gapped extension. The percent of computation time of each step to total computation time is in bracket.

	GHOSTX	BLASTX
Seed search	146.0 (49%)	57759.5 (86%)
Ungapped extension	33.1 (11%)	7674.3 (11%)
Gapped extension	97.5 (32%)	875.1 (1%)
Others	24.4 (8%)	1009.8 (2%)
Total computation time	301.0 (100%)	67318.7 (100%)

Table 3.8: Computation time of the preprocessing including indexing with KEGG GENES (3.9 GB) and NCBI nr (14.8 GB). The first, second, and third columns show the name of each program, the computation time with KEGG GENES, and the computation time with NCBI nr.

	Computation time with KEGG GENES (sec.)	Computation time with NCBI nr (sec.)
GHOSTX	1589.2	4415.2
RAPSearch	1914.2	4210.5
BLASTX	637.6	1678.9

ing, for GHOSTX, BLASTX, and RAPSearch. Table 3.8 shows the computation time for preprocessing. Preprocessing in GHOSTX requires computation time almost equal to RAPSearch. However, protein sequence homology search computation time is generally much larger than that required for the database construction phase when a huge amount of DNA reads sequenced by NGSs are processed. Moreover, preprocessing is only performed when a database is updated. Therefore, we think preprocessing is not a problem in practice.

3.3.5 Evaluation of Memory Size

While GHOSTX can search for homologues more efficiently than BLASTX, GHOSTX requires more memory. GHOSTX uses approximately 18 GB of memory for constructing the indexes of a typical database, and approximately 13 GB for the protein sequence homology search itself, when a 2 GB database chunk is used. However, using a smaller database chunk size can decrease the amount of memory required. Table 3.9 shows the relationships between the amount of memory required to construct the indexes and

Table 3.9: Comparison with memory size for KEGG GENES (3.9 GB) of each size of the database chunks. The first, second, and third columns show the size of the database chunk, the used memory size for constructing index (GB), and the used memory size for protein sequence homology search (GB).

Chunk size	Memory size for constructing index (GB)	Memory size for protein sequence homology search (GB)
512 MB	4.6	4.2
1 GB	9.2	7.2
2 GB	18.2	13.3

Table 3.10: Comparison with Computation time for KEGG GENES (3.9 GB) of each size of the database chunks. The first, second, and third columns show the size of the database chunk, the computation time, and the acceleration in processing speed relative to GHOSTX with 2GB database chunks, respectively.

Chunk size	Computation time (sec.)	Acceleration ratio
512 MB	526.9	0.8
1 GB	452.7	0.9
2 GB	401.9	1.0

protein sequence homology search versus the size of a database chunk. The required memory size of GHOSTX is almost linearly increased in proportion to the size of a database chunk. If a database is divided into more chunks, the required memory size becomes smaller accordingly. Therefore, with smaller database chunk sizes, GHOSTX can be executable even on a general PC. Of course, there is a trade-off between database chunk size and search speed. Protein sequence homology search computation times increase as the size of a database chunk becomes smaller. This is so because the same suffix array search has to be performed for each respective chunk, and the number of suffix array searches increases as a result. However, the situation is not dire; as shown in Table 3.10, the search speed of GHOSTX with 512 MB chunks is approximately 20% slower than that with 2 GB chunks. The maximum size of a database chunk is 2 GB in GHOSTX, because the maximum size of a 32 bit integer is 2 GB.

3.4 Summary

We have developed an efficient algorithm for performing protein sequence homology searches, and have implemented it as GHOSTX. GHOSTX has sufficient search sensitivity for practical analyses. It uses an extremely efficient seed search algorithm, employing database and query suffix arrays, to achieve a well over 100 times faster protein sequence homology search than BLASTX. GHOSTX is also almost 1.4–1.6 times faster than RAPSearch, which is one of the fastest protein sequence homology search tools available, even though GHOSTX is slightly more accurate.

Chapter 4

A Protein Sequence Homology Searches with Clustering Subsequences Technique

4.1 Introduction

In this study, we developed a faster protein sequence homology search algorithm using database subsequence clustering. Current protein sequence homology searches require long computation time to extend alignments without gaps, because seed searches tend to produce a large number of seeds [54]. However, only a small number of seeds generate ungapped extension scores that are higher than the score threshold, and the wasted computation time increases the overall time required for ungapped extensions. Our algorithm clusters subsequences derived from a database and reduces non-representative seeds within these clusters to minimize the computation time spent on ungapped extensions. In this study, we developed a novel fast protein sequence homology search algorithm that uses hash tables, and applied our subsequence clustering method to the index to further accelerate protein sequence homology search. We designated this algorithm as GHOSTZ.

4.2 Methods

The flow of GHOSTZ, which adopts the seed-and-extend strategy used in BLAST is shown in Fig. 4.1. Subsequences are extracted from a database and similar subsequences are clustered. Then, hash tables, which contain indexes for the subsequences

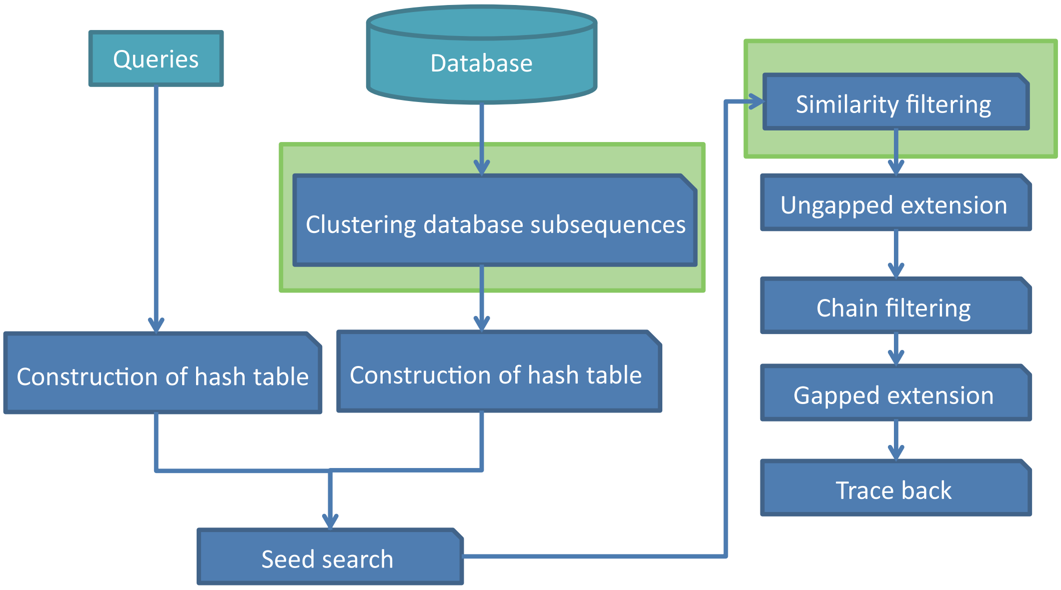


Figure 4.1: Flow of the proposed protein sequence homology search method based on database subsequence clustering. Clustering of database subsequences and similarity filtering (green box) are included in this method.

and the clusters, are constructed. The protein sequence homology search method uses the hash tables to select the seeds for alignments from representative sequences in the clusters. The distance between a query subsequence and a cluster representative is calculated, and the lower bounds for the distance between the query subsequence and other members of the cluster are computed based on triangle inequality as shown in Fig. 4.2. If the computed lower bound is less than or equal to the distance threshold, the seed is selected for ungapped extension to investigate the homology between the query and the sequences of the cluster. This filtering using the lower bounds of the distance is referred to here as “similarity filtering”. Finally, chain filtering is used to bring similar extended seeds together, and gapped extension is performed to align the extended seeds with gaps.

In the database subsequence clustering and seed search processes, the protein sequences of a query and database are converted to a reduced amino acid alphabet to increase search sensitivity. We used a 10-letter reduced amino acid alphabet (A, {K, R}, {E, D, N, Q}, C, G, H, {I, L, V, M}, {F, Y, W}, P, {S, T}) derived from the BLOSUM62 matrix [38]. This reduced amino acid alphabet has been used successfully in a previous study on protein sequence homology searches [58]. For ungapped and gapped extensions, alignments were performed with the standard 20-letter amino acid

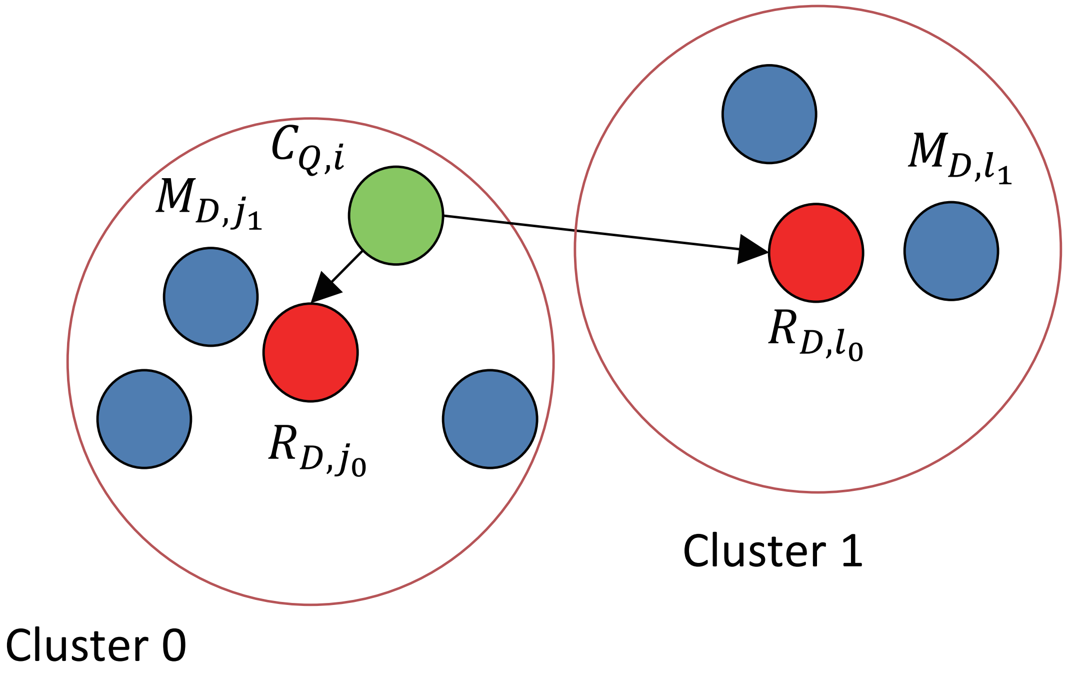


Figure 4.2: Example of similarity filtering. $C_{Q,i}$ is the query subsequence. R_{D,j_0} and R_{D,l_0} are the representative subsequences in cluster 0 and cluster 1, respectively. The lower bound of the distance between $C_{Q,i}$ and the member subsequence M_{D,j_1} in cluster 0 is calculated from the distance $d(C_{Q,i}, R_{D,j_0})$. When the lower bound of $d(C_{Q,i}, M_{D,j_1}) \leq T_{distance}$, the seed for $C_{Q,i}$ and M_{D,j_1} is taken to the next step. The lower bound of the distance between $C_{Q,i}$ and the member subsequence M_{D,l_1} in cluster 1 is calculated from the distance $d(C_{Q,i}, R_{D,l_0})$. When the lower bound of $d(C_{Q,i}, M_{D,l_1}) > T_{distance}$, the seed for $C_{Q,i}$ and M_{D,l_1} is not taken to the next step.

alphabet.

4.2.1 Database Subsequence Clustering and Construction of Hash Tables

Our database subsequence clustering approach was developed for efficient sequence homology searches. In this method, subsequences in a database are clustered to be used in similarity filtering; however, we do not cluster subsequences used in seed searches, but instead use longer subsequences, which overlap seed subsequences. To avoid term confusion, we use “subsequence for seed” for subsequences used in general seed searches denoted by S, and “subsequence for clustering” for subsequences used in the database subsequence clustering and similarity filtering denoted by C. All subsequences for clustering depend on a subsequence for seed. Therefore, GHOSTZ first builds a hash table of subsequences for seed, and then determines which subsequences should be used for clustering based on the hash table. Database subsequence clustering is performed using these subsequences. Therefore, we will first describe the construction of hash tables containing subsequences for seed, and then the construction of subsequences for clustering and database subsequence clustering.

Here, the text $T = T[0, n] = t_0...t_{n-1}$ denotes a sequence of symbols, and the length of T is $|T| = n$. Each symbol is an element of an alphabet Σ ($|\Sigma|$ of protein is 20). $T[i] = t_i$ and $T[i, i + j] = t_i...t_{i+j-1}$ are subsequences. The sequence of a query is Q. The sequences D_0, D_1,...,D_{N-1} in a database are connected by inserting delimiters to transform them into a single long sequence $D = \#D_0\#D_1\#, ..., \#D_{N-1}\#$ (marked by the special symbol #). A seed is a pair of identical or similar subsequences of Q and D. $S_{Q,i} = Q[i, i + l]$ and $S_{D,j} = D[j, j + l]$ is the subsequence of Q and D for a seed, and $\{S_{Q_i}, S_{D,j}\}$ is a seed. The hash table used to identify subsequences for seed stores a pair of hash values of $S_{D,i}$ and the starting point i of $S_{D,i}$.

In the BLAST-like seed-and-extend strategy based algorithms, search speed can be increased by decreasing the number of seeds. The number of seeds can be decreased if longer subsequences are used for seeds, because this decreases the number of randomly matched cases. However, this also causes a decrease in the search sensitivity. Thus, tolerances are required in the matching to retain sufficient search sensitivity. In BLAST, the length of the subsequence for seed is three and neighborhood words are identified [5]. A neighborhood word is a subsequence similar to each subsequence [4]. BLAST uses a large variety of subsequences of each subsequence in a seed search to increase search sensitivity using neighborhood words, which, however, are ineffective for longer subsequences for seed because of high variation in neighborhood words.

GHOSTZ identifies long subsequences by employing a reduced amino acid alphabet in the seed search. In subsequences, the conventional amino acid alphabet is converted to a reduced amino acid alphabet, and then the hash value for this subsequence is calculated. Using the reduced amino acid alphabet, the variety of subsequences for each original subsequence becomes one. In addition, the reduced amino acid alphabet allows GHOSTZ to find longer subsequences without a significant decrease in search sensitivity. In GHOSTZ, the length of a subsequence for seed is determined by a sum of the match scores for this subsequence. Because the frequency of each amino acid in subsequences varies, the probability of finding each particular subsequence is different. Therefore, different subsequences may have different lengths. Score definition has been previously proposed to calculate matches between reduced amino acid alphabets [37]. However, in the present study we used a simpler definition. We defined the match scores of the groups of reduced amino acid alphabets by the largest match score in the group based on the original score matrix. For example, in the BLOSUM62 score matrix, the match scores of amino acids F, Y, and W, are 6, 7, and 11, respectively; thus, the match score for the group including F, Y, and W is 11. To avoid insignificant hits, only subsequences with scores that exceed score threshold T_{seed} are hashed as subsequences for seed. For example, when $T_{seed} = 39$, “HDGLNP” is not used in seed search because its score is 38 and does not exceed T_{seed}. However, “HDGLNPA” is used in seed search because its score is 42, which exceeds T_{seed}. Furthermore, in our implementation, the length of subsequences for seed is restricted to 6–8 residues, because a perfect hash function is used.

After building the hash table of subsequences for seed, the subsequences for clustering are constructed and database subsequence clustering is performed as follows: if i is the starting point of $S_{D,i} = D[i, j]$ and L is the length of the subsequence used for clustering, then let $C_{D,i} = D[i - L/2, i + L/2]$ be the subsequence for clustering. For clustering, the subsequence for clustering with i as the center is used instead of a subsequence for seed with i as the starting point. The relationship between $C_{D,i}$ and $S_{D,i}$ is shown in Fig. 4.3. If $C_{D,i}$ has delimiters, $C_{D,i}$ are not used for clustering, because $C_{D,i}$ contains the subsequence of several sequences in the database. $C_{D,i}$ becomes a member of a cluster if it has the same hash value of $S_{D,j}$ as the cluster representative $C_{D,j}$ and the distance between the representative of a cluster $C_{D,j}$ and $C_{D,i}$ is lower than or equal to the distance threshold $T_{cluster}$. Hamming distance, which is the number of mismatches between sequences, is used to measure this distance. To reduce the computation time required for clustering, a greedy algorithm similar to

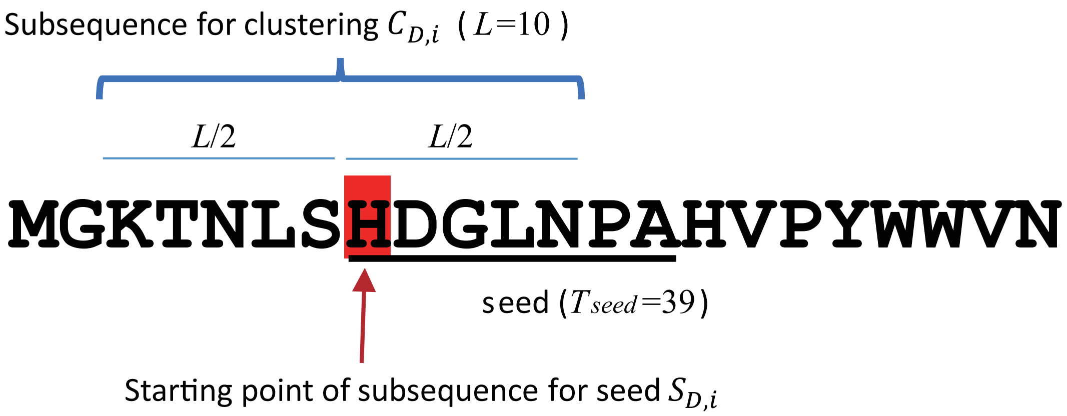

Subsequence for clustering $C_{D,i}$ (L=10)
$L/2$
$L/2$
MGKTNLSHDGLNPAHVPYWWVN
seed (T_{seed}=39)
Starting point of subsequence for seed $S_{D,i}$

Figure 4.3: Relationship between a subsequence used for clustering and the starting position of the seed.

CD-HIT [29, 14] was employed. The algorithm for database subsequence clustering is shown in Fig. 4.4. In this algorithm, the first subsequence sampled always becomes a cluster representative. All subsequences are compared with each cluster representative, and a subsequence becomes a new cluster representative if it is not a member of any other cluster. Before running database subsequence clustering, we recommend that similar sequences are arranged close to each other in the input file using a clustering tool such as CD-HIT, which allows the clustering algorithm to cluster subsequences more efficiently. After subsequence clustering, the results are used to construct three tables to be used as indexes for seed searches. The B_e hash table stores hash values of $S_{D,i}$ and starting points i of $S_{D,i}$ for the representatives of clusters containing only one member. The B_r hash table stores the hash values of $S_{D,i}$, their cluster IDs, and starting points i of $S_{D,i}$, which are representatives of a cluster (not stored in B_e). The B_m table stores mapping from the cluster IDs to the starting points i of $S_{D,i}$, which $C_{D,i}$ are the members of that cluster. These three tables are used for seed search. Examples of B_e, B_r, and B_m are shown in Fig. 4.5A, B, and C.

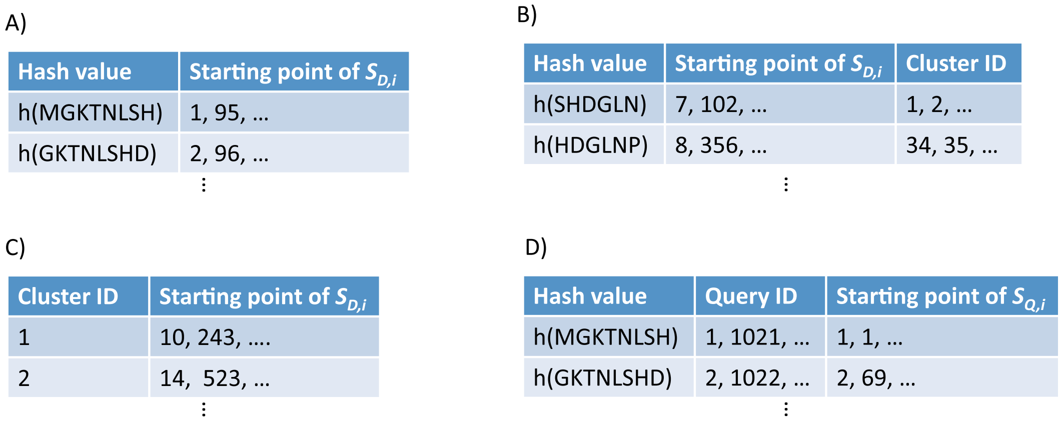
4.2.2 Seed Search and Similarity Filtering

Seed search is performed with B_e, B_r, B_m, and the hash table of queries. The hash table of the queries is constructed before seed search. This hash table contains the hash values of $S_{Q,i}$, query IDs, and starting points of subsequences for the corresponding hash values. An example of a hash table of queries is shown in Fig. 4.5D.

In seed search, seeds of query subsequences and representative subsequences in the

```
Let $D$ be the concatenated sequence of a database.
Let $L$ be the length of $C_{D,i}$.
Let $T_{cluster}$ be the distance threshold of clustering.
Let $clusters$ be the results of clustering.
build hash table $H$ from $D$
for each hash value $h$ in $H$ do
    Let $flags_R$ be the flag for representative subsequences.
    Let $members$ be the lists of members.
    for $i$ in $H.find(h)$ do
        build subsequence $C_{D,i}$
        $flag_M \leftarrow false$                    ▷ The flags for member subsequences
        if $C_{D,i}$ does not have delimiters then
            for $j$ in $H.find(h)$ do
                if $j \geq i$ then
                    break
                end if
                build subsequence $C_{D,j}$
                if $flags_R[j]$ & $(d(C_{D,i}, C_{D,j}) \leq T_{cluster})$ then
                    add $C_{D,i}$ to $members[j]$
                    $flag_M \leftarrow true$
                    break
                end if
            end for
        end if
        if $flag_M$ then
            $flags_R[i] \leftarrow false$
        else
            $flags_R[i] \leftarrow true$
        end if
    end for
    store $flags_R$ and $members$ to $clusters$
end for
build the tables $B_e, B_r$ and $B_m$ from $H$ and $clusters$
```

Figure 4.4: Pseudo-code for database subsequence clustering.



A)

Hash value	Starting point of $S_{D,i}$
h(MGKTNLSH)	1, 95, …
h(GKTNLSHD)	2, 96, …
⋮	

B)

Hash value	Starting point of $S_{D,i}$	Cluster ID
h(SHDGLN)	7, 102, …	1, 2, …
h(HDGLNP)	8, 356, …	34, 35, …
	⋮	

C)

Cluster ID	Starting point of $S_{D,i}$
1	10, 243, ….
2	14, 523, …
	⋮

D)

Hash value	Query ID	Starting point of $S_{Q,i}$
h(MGKTNLSH)	1, 1021, …	1, 1, …
h(GKTNLSHD)	2, 1022, …	2, 69, …
	⋮	

Figure 4.5: Examples of data structures. A) An example of a hash table B_e. B_e stores the hash values of $S_{D,i}$ and the starting points i of $S_{D,i}$ that are the representatives $C_{D,i}$ of clusters with only one member. B) An example of a hash table B_r. B_r stores the hash values of $S_{D,i}$, their cluster IDs, and starting points i of $S_{D,i}$, which are the representative $C_{D,i}$ of a cluster (not stored in B_e). C) An example of a table B_m. B_m that stores the mapping from the cluster IDs to the starting points i of $S_{D,i}$, which $C_{D,i}$ are the members of that cluster. D) An example of a hash table of the queries, which contains the hash values of $S_{Q,i}$, query IDs, and starting points i of subsequences for the corresponding hash values.

database are found using B_e and B_r. If the seeds are from B_e, ungapped extension is performed because there are no other subsequences in the cluster. If the seeds are from B_r, similarity filtering is performed. Then, the hamming distance between a query and database subsequence is calculated. Given two sequences S_1 and S_2, $d(S_1, S_2)$ denotes the distance between S_1 and S_2, which should satisfy the following triangle inequality:

$$d(S_1, S_2) \leq d(S_1, S_3) + d(S_2, S_3) \tag{4.1}$$

If $C_{Q,i}$ is the subsequence of the query, $M_{D,j}$ ($C_{D,j}$) is the sequence of a cluster member, and $R_{D,k}$ ($C_{D,k}$) is the subsequence of a representative cluster member, then the lower bound of the distance between $R_{D,i}$ and $M_{D,j}$ from this inequality will be:

$$d(C_{Q,i}, M_{D,j}) \geq d(C_{Q,i}, R_{D,k}) - d(R_{D,k}, M_{D,j}) \tag{4.2}$$

The lower bound of the distance between $C_{Q,i}$ and $M_{D,j}$ is calculated, and the seed is extended without gaps if the lower bound of the distance is less than or equal to the distance threshold $T_{distance}$. The relationships among the query, cluster representative, and cluster members are shown in Fig. 4.6. The pseudo-code for the seed search and similarity filtering is shown in Fig. 4.7.

4.2.3 Ungapped Extension

Gapped extension generally requires long computation time; therefore, most protein sequence homology search algorithms perform ungapped extension before gapped extension. We used ungapped extension to filter candidate seeds in the output from seed search. Only seeds with ungapped extension scores that exceed the score threshold $T_{ungapped}$ are stored and extended with gaps after ungapped extension is complete. In ungapped extension, the X-dropoff used in BLAST [4] is applied to accelerate extension process. The $T_{ungapped}$ and other parameters for ungapped extensions are the same as the default parameters in BLAST.

For the efficient memory access in ungapped extension, seed searches are performed for multiple queries simultaneously. If the hash values of query subsequences are the same, their starting points are packed using a hash table. Then, ungapped extension is performed for the queries that have identical hash values in sequential order, because it increases the cache hit ratio when accessing the positions of sequences in the database (lines 9–31 in Fig. 4.7).

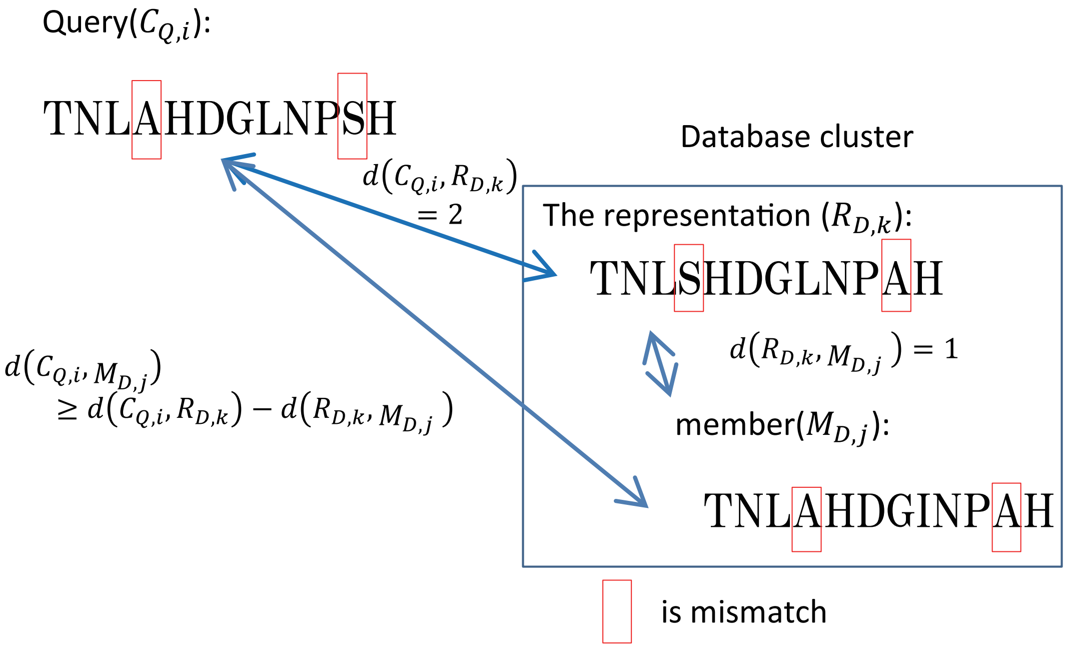

Query($C_{Q,i}$):
TNLAHDGLNPSH
$d(C_{Q,i}, R_{D,k}) = 2$
Database cluster
The representation ($R_{D,k}$):
TNLSHDGLNPAH
$d(R_{D,k}, M_{D,j}) = 1$
member($M_{D,j}$):
TNLAHDGINPAH
$d(C_{Q,i}, M_{D,j}) \geq d(C_{Q,i}, R_{D,k}) - d(R_{D,k}, M_{D,j})$
is mismatch

Figure 4.6: Relationships among the query subsequence, representative cluster subsequence, and member of the cluster that satisfies the triangle inequality.

4.2.4 Chain Filtering and Gapped Extension

Chain filtering is performed after ungapped extension because some seeds overlap. Therefore, the number of gapped extensions can be reduced by merging overlapping seeds. After chain filtering, the seeds are extended with gaps using X-dropoff [5].

4.2.5 Execution of the Protein Sequence Homology Search Method without Subsequence Clustering

The flow of the protein sequence homology search without subsequence clustering is shown in Fig. 4.8. This method is almost identical to that used in GHOSTZ, except that subsequence clustering and similarity filtering are not used for seed search. The method without subsequence clustering was used to evaluate the reduction in computation time achieved by subsequence clustering. Here, query subsequences are searched against all the subsequences in the database using hash tables. Next, all seeds are directly extended using the ungapped extension process. Finally, chain filtering is performed to merge similar seeds, and gapped extension is used to extend seed sequences.

```
Let $hash_{queries}$ be the hash value of queries.
Let $positionslist_{queries}$ the list of positions of $hash_{queries}$ of each query.
Let $B_r$ be the hash table for the representative subsequences in a database.
Let $B_m$ the inverted table for the member subsequences in a database.
Let $T_{ungapped}$ the threshold for ungapped extension.
Let $List_Q$ the list of query data passed through similarity filtering.
$pairs_{id,position} \leftarrow B_r.find(hash_{queries})$
for $id_{cluster}, p_r$ in $pairs_{id,position}$ do
    for $i_{query}, p_{query}$ in $positionslist_{queries}$ do
        $d \leftarrow CalculateDistance(i_{query}, p_{query}, p_r)$
        if $SimilarityFiltering(d)$ then
            add $i_{query}, p_{query}$ to $List_Q$
        end if
        if $d \leq T_{distance}$ then
            build $seed_r$ by using $i_{query}, p_{query}, p_r$
            $score \leftarrow UngappedExtention(seed_r)$
            if $score > T_{ungapped}$ then
                store $seed_r$
            end if
        end if
    end for
    $positions_m \leftarrow B_m.find(id_{cluster})$
    for $p_m$ in $positions_m$ do
        for $i_{query}, p_{query}$ in $List_Q$ do
            build $seed_m$ by using $i_{query}, p_{query}, p_m$
            $score \leftarrow UngappedExtention(seed_m)$
            if $score > T_{ungapped}$ then
                store $seed_m$
            end if
        end for
    end for
    clear $List_Q$
end for
```

Figure 4.7: Pseudo-code for seed search, similarity filtering, and ungapped extension in the case of multiple cluster members.

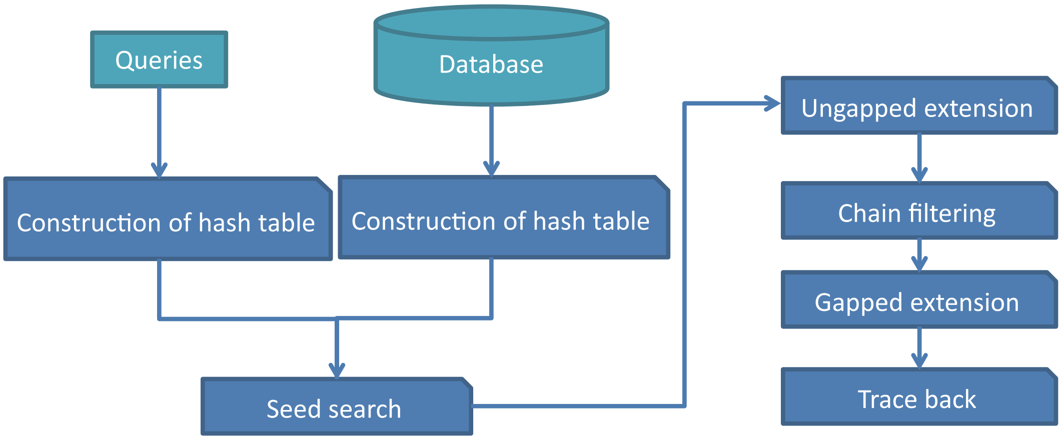


Figure 4.8: Flow of the proposed protein sequence homology search method without database subsequence clustering for the purpose of comparison.

4.3 Results

4.3.1 Datasets and Computing Environment

We evaluated the performance of the protein sequence homology searches with and without subsequence clustering using protein sequences in the KEGG GENES database [20, 19] (May 2013). This database contains approximately 10,000,000 protein sequences comprising a total of approximately 3,600,000,000 residues. For query sequences, we used the datasets of microbiome metagenomic sequences derived from soil (accession number SRR407548, read length = 150 bp), ocean (accession number ERR315856, read length = 104 bp), and humans (accession number SRS011098, read length = 101 bp). SRR407548 and ERR315856 were obtained from the DDBJ Sequence Read Archive. SRS011098 was obtained from the HMP-DACC web site. We used the whole metagenomic shotgun sequencing data from SRS011098. For all datasets, 10,000 randomly selected short DNA reads were used. Evaluation tests were performed on a workstation with two Intel Xeon 5670 processors (2.93 GHz, 6 cores) and 54 GB memory.

For the protein sequence homology search with and without subsequence clustering, we used the seed score threshold of $T_{seed} = 39$. T_{seed} was determined to be similar in sensitivity to RAPSearch. The parameters used for gapped and ungapped extensions were the same as BLASTX default parameters. To efficiently perform database subsequence clustering, similar sequences were arranged close to each other in the database file based on the results of CD-HIT.

Table 4.1: Computation times for protein sequence homology searches using different subsequence length for SRR407548 reads against the KEGG GENES database. L is the subsequence length. The increase in processing speed is presented as the ratio of the time used for search with subsequence clustering to that used for search without subsequence clustering.

	Computation time (sec.)	Acceleration ratio
Without clustering	954.1	1.0
$L = 6$	332.6	2.9
$L = 8$	393.7	2.4
$L = 10$	464.3	2.1
$L = 12$	456.2	2.1
$L = 14$	508.3	1.9

4.3.2 Relationship Between Subsequence Length and Acceleration Ratio and Search Sensitivity

The subsequence clustering method has three parameters: subsequence length L, distance threshold for the representative of a cluster $T_{cluster}$, and distance threshold for similarity check $T_{distance}$. Subsequence length L particularly affects the performance of the search method because $T_{cluster}$ and $T_{distance}$ depend on L; therefore, we first determined the optimal subsequence length using $L = 6, 8, 10, 12$, and 14 and fixed distance thresholds of $T_{cluster} = 0.1L$ and $T_{distance} = 0.2L$. We used 10,000 randomly selected short DNA reads from soil microbiome metagenomic sequences (SRR407548) and the KEGG GENES database. The speed with different L used for the subsequence clustering search method was compared with the method without subsequence clustering in Table 4.1. As shown, the speed of the search method that included subsequence clustering increased when L decreased.

The search sensitivity of the protein sequence homology search for different query sequences was estimated using search results obtained by the Smith-Waterman local alignment algorithm implemented in SSEARCH [41] as the correct result. Performance was estimated as the portion of the results corresponding to the correct result achieved when the subject sequence with the highest score in SSEARCH was the same as the subject sequence obtained by our search method with subsequence clustering. The search sensitivity of $L = 10$ was better than that of the other lengths (Fig. 4.9). Therefore, we considered $L = 10$ as the optimal subsequence length because it yielded a good balance between sensitivity and computation time. Using database subsequence

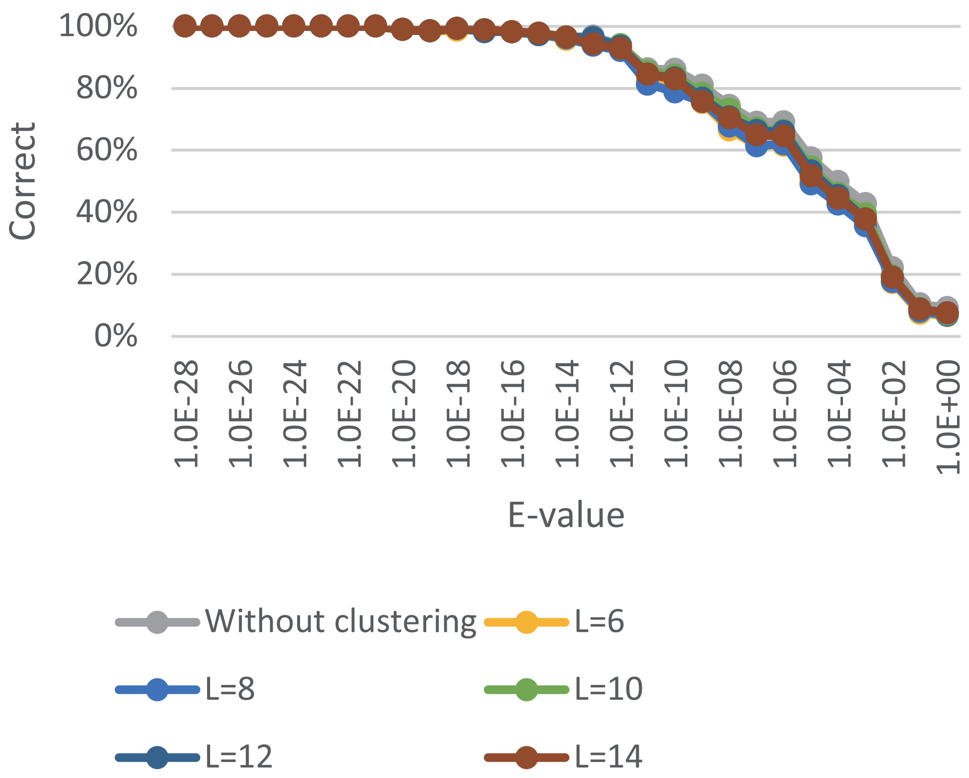


Figure 4.9: The search sensitivity of GHOSTZ for SRR407548 sequence alignments against the KEGG GENES database. The percentage of correct answers is shown on the vertical axis, and the E-values of the alignments are shown on the horizontal axis.

clustering, GHOSTZ provided an approximately two-fold increase in processing speed without a significant decrease in search sensitivity.

4.3.3 Comparison of the Proposed Search Method with Other Methods

To further evaluate GHOSTZ, we compared its search sensitivity and computation time with those of NCBI BLASTX (version 2.2.28+), BLAT (version 34 standalone), and RAPSearch (version 2.12) using metagenomic DNA sequences (SRR407548, SRS011098, and ERR315856) and the KEGG GENES database. Composition-based statistics [6] were not used because it is not employed in SSEARCH. Therefore, BLASTX was executed with the command line options "-outfmt 6 -comp_based_stats 0," used for tabular format. BLAT does not include a function to translate DNA reads to pro-

Table 4.2: Search sensitivity of SRR407548, SRS011098, and ERR315856. The search sensitivity is calculated as the ratio of correctly searched queries with the E-values $< 10^{-5}$.

	SRR407548	SRS011098	ERR315856
GHOSTZ	0.86	0.98	0.96
GHOSTX	0.84	0.98	0.95
RAPSearch	0.87	0.97	0.96
BLAT	0.51	0.93	0.81
BLASTX	0.94	0.97	0.96

tein sequences; therefore, we translated DNA reads based on the standard codon table. BLAT was executed with the command line options "-q = prot -t = prot -out = blast8," which instructed the program to use protein queries and databases, and tabular format for output data. RAPSearch was executed with the default command line options. GHOSTZ was executed with $L = 10$.

Search sensitivity was evaluated the same way as in Section 4.3.2. The results for SRR407548, SRS011098, and ERR315856 are shown in Fig. 4.10, 4.11 and 4.12, respectively. The search sensitivity of GHOSTZ was lower than that of BLASTX, especially for hits with the E-values above $1.0 \times E^{-6}$. However, hits with such high E-values are not used in practice because of their unreliability. For example, Trunbaugh *et al.* used hits with the E-values below $1.0 \times E^{-5}$ [53], and Kurokawa *et al.* used hits with the E-values below $1.0 \times E^{-8}$ [23]. Therefore, we used the single-value search sensitivity calculated as the ratio of correctly searched queries to all queries with the E-values $< 1.0 \times E^{-5}$. Table 4.2 showing search sensitivity for each program indicates that the search sensitivity of GHOSTZ was better than that of BLAT and almost equal to that of RAPSearch and GHOSTX. Therefore, we believe that GHOSTZ has search sensitivity sufficient for most metagenomic applications.

The computation time for each method was also evaluated. The software was run with the same commands used to analyze search sensitivity. Computation times for the tested methods with SRR407548, SRS011098, and ERR315856 are shown in Table 4.3, 4.4 and 4.5, respectively. GHOSTZ demonstrated approximately 213.3–285.3, 3.5–5.0, 2.6–3.0, and 1.0–2.0 times faster processing than BLASTX, BLAT, RAPSearch, and GHOSTX, respectively.

We also measured search sensitivity and computation time for each tool using different parameters. We used 10,000 short DNA reads randomly selected from SRR407548

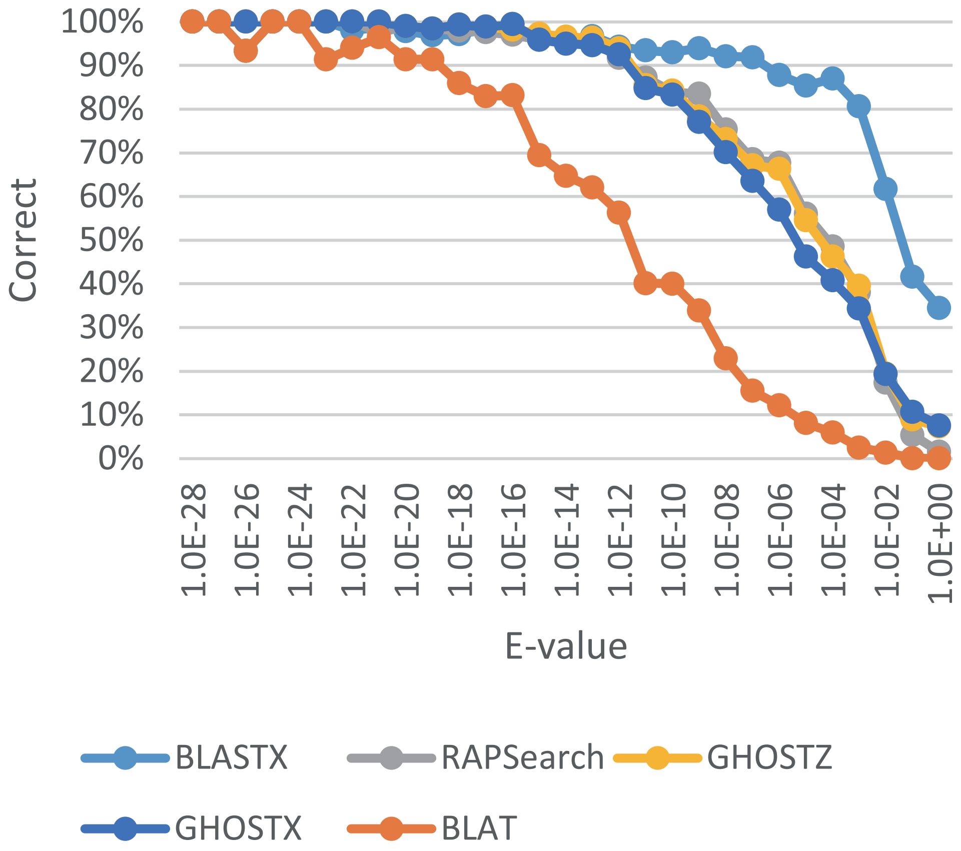


Figure 4.10: Search sensitivity of different search methods for SRR407548 sequence alignments against the KEGG GENES database. The percentage of correct answers is shown on the vertical axis, and the E-values of the alignments are shown on the horizontal axis.

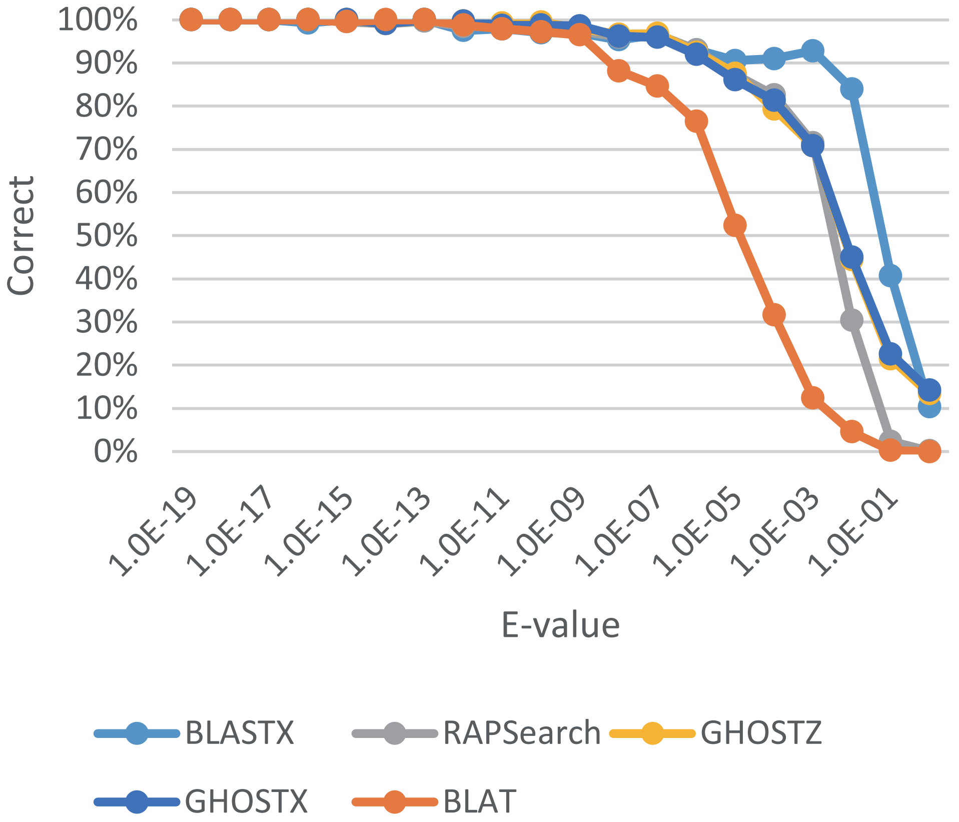


Figure 4.11: Search sensitivity of different search methods for SRS011098 sequence alignments against the KEGG GENES database. The percentage of correct answers is shown on the vertical axis, and the E-values of the alignments are shown on the horizontal axis.

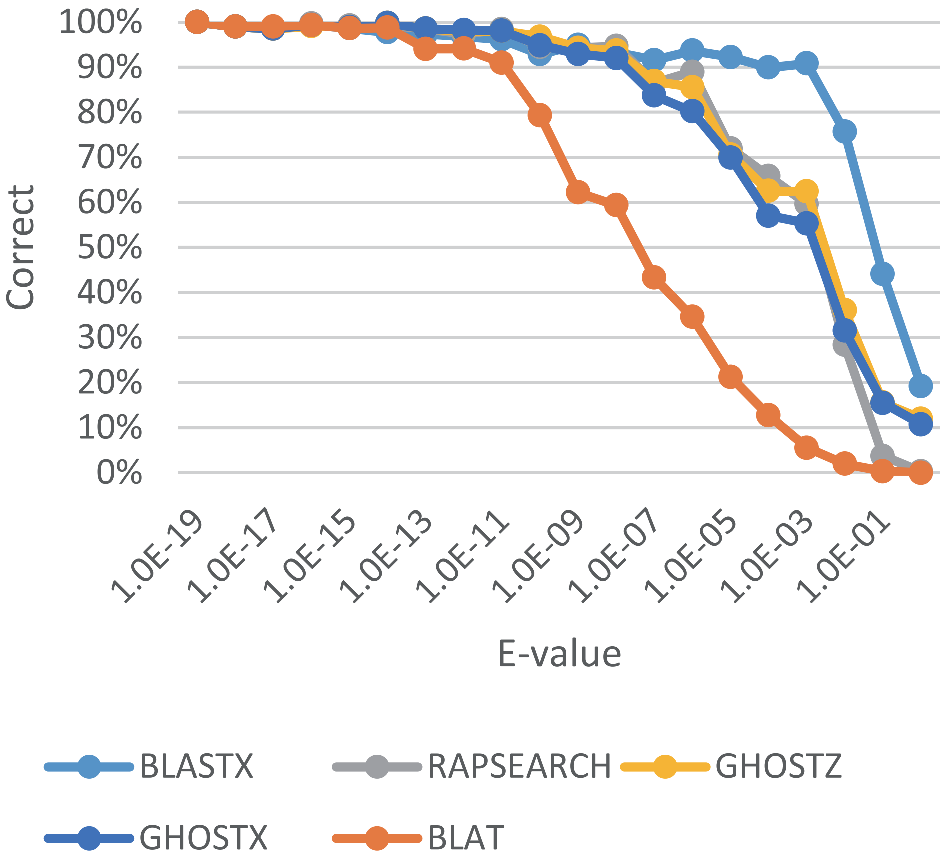


Figure 4.12: Search sensitivity of different search methods for ERR315856 sequence alignments against the KEGG GENES database. The percentage of correct answers is shown on the vertical axis, and the E-values of the alignments are shown on the horizontal axis.

Table 4.3: Computation times for SRR407548 reads against the KEGG GENES database. The increase in processing speed for the search using subsequence clustering compared to BLASTX using one thread.

	Computation time (sec.)	Acceleration ratio
GHOSTZ	464.3	285.3
GHOSTX	906.0	146.2
RAPSearch	1404.2	94.3
BLAT	2333.8	56.8
BLASTX	132450.0	1.0

Table 4.4: Computation times for SRS011098 reads against the KEGG GENES database. The increase in processing speed for the search using subsequence clustering compared to BLASTX using one thread.

	Computation time (sec.)	Acceleration ratio
GHOSTZ	290.7	231.8
GHOSTX	323.9	208.0
RAPSearch	742.4	90.8
BLAT	1145.1	58.9
BLASTX	67391.3	1.0

Table 4.5: Computation times for ERR315856 reads against the KEGG GENES database. The increase in processing speed for the search using subsequence clustering compared to BLASTX using one thread.

	Computation time (sec.)	Acceleration ratio
GHOSTZ	387.7	213.3
GHOSTX	384.2	215.2
RAPSearch	995.3	83.1
BLAT	1357.9	60.9
BLASTX	82672.7	1.0

and the KEGG GENES database. Because it is difficult to compare multiple plots showing the results for various parameters, we used the single-value search sensitivity calculated as the ratio of correctly searched queries to all queries with the E-values $< 1.0 \times E^{-5}$. Using these conditions, GHOSTZ computation time and search sensitivity were 464.3 seconds and 0.86, respectively. Table 4.6, 4.7, 4.8 and 4.9 show search

Table 4.6: The influence of GHOSTX parameters on sensitivity and computation time. We changed T_{seed}, which is the threshold of seed search, to 22, 24, 26, 28, 30, and 32, and D, which is the upper mismatch score, to 1, 4, and 7. The columns show T_{seed}, D , sensitivity, and computation time.

T_{seed}	D	Sensitivity	Computation time (sec.)
22	1	0.85	1542.0
22	4	0.93	4313.4
22	7	0.96	10964.6
24	1	0.84	1248.0
24	4	0.92	3311.4
24	7	0.95	9221.9
26	1	0.81	859.2
26	4	0.90	2124.1
26	7	0.95	5183.0
28	1	0.77	542.9
28	4	0.87	1141.0
28	7	0.93	3877.2
30	1	0.73	357.9
30 (default)	4 (default)	0.84	906.0
30	7	0.92	3637.9
32	1	0.69	246.7
32	4	0.81	650.3
32	7	0.90	2793.5

sensitivity and computation time for GHOSTX, RAPSearch, BLAT, and BLASTX, respectively, using different parameters. GHOSTX demonstrated the sensitivity with $\{T_{seed} = 22, D = 1\}$, $\{T_{seed} = 28, D = 4\}$ and $\{T_{seed} = 30, D = 4\}$ similar to that of GHOSTZ, but required longer computation time. The search sensitivity of RAPSearch showed a dramatic decrease in the fast-mode, and that of BLAT was not significantly improved, even with a smaller tile size. Using the fastest parameter, the sensitivity of BLASTX was similar to that of GHOSTZ; however, the required computation time was much longer.

4.3.4 Evaluation of Memory Size

The amount of memory required for running GHOSTZ depends on the size of a database. The memory sizes of current computing systems are often small compared to those of current databases. Therefore, GHOSTZ divides a database into several

Table 4.7: The relationship between RAPSearch parameters and performance. We changed -a F (default) and T, which instructed the program to perform a fast-mode search. The columns show the parameters, search sensitivity, and computation time.

RAPSearch parameters	Search sensitivity	Computation time (sec.)
default (-a F)	0.87	1404.2
fast mode (-a T)	0.60	156.3

Table 4.8: The relationship between BLAT parameters and performance. We changed -tileSize, which is the subsequence length for seed search, to 4, 5 (default), and 6. The columns show the parameters, search sensitivity, and computation time.

-tileSize	Search sensitivity	Computation time (sec.)
4	0.60	65979.0
5 (default)	0.51	2333.8
6	0.45	492.9

chunks, sequentially searches each chunk, and then merges the results with those of the previous chunk search performed when database division has been done prior to the construction of the database indexes. The default chunk size is 1 GB. Using this approach, GHOSTZ dramatically reduces working memory requirements. However, even using this technique, GHOSTZ needs more memory than RAPSearch. When we used 10,000 randomly selected short DNA reads from soil microbiome metagenomic sequences (SRR407548) and the KEGG GENES database, GHOSTZ required approximately 41 GB of memory for constructing the indexes of the database, and approximately 7 GB for protein sequence homology search (Table 4.10). In contrast, RAPSearch required only approximately 4 GB for protein sequence homology search. However, GHOSTZ can reduce memory requirements by decreasing database chunk size. As shown in Table 4.10, the memory required for GHOSTZ has a nearly linear relationship with the size of database chunks. If a database is divided into smaller chunks, the required memory decreases proportionally. However, it decreases search speed; consequently, the computation time for protein sequence homology search increases with the decrease in database chunk size, because the number of clusters increases and the cache hit ratio in ungapped extension decreases. However, this speed reduction is not dramatic; as shown in Table 4.11, GHOSTZ search with 128 MB chunks is approximately 12% slower than that with 1 GB chunks. Therefore, with smaller database

Table 4.9: The relationship between BLASTX parameters and performance. We changed -threshold, which is the threshold for neighborhood words, to 12 (default), 14, 16, 18, 20, 22, and 24, and -word_size, which is the subsequence length for seed search, to 3 (default), 4, 5, 6, and 7. The columns show -threshold, -word_size, search sensitivity, and computation time. When the parameters were "-threshold 12 -word_size " and "-threshold 12 -word_size 7", BLASTX required > 96 hours, and we were unable to measure the computing time under these conditions.

-threshold	-word_size	Search sensitivity	Computation time (sec.)
12 (default)	3 (default)	0.94	132450.0
12	4	0.94	182071.8
12	5	0.94	298595.0
12	6	N/A	N/A
12	7	N/A	N/A
14	3	0.94	41032.6
14	4	0.94	65636.3
14	5	0.94	126303.4
14	6	0.94	264059.3
14	7	0.94	217667.0
16	3	0.94	26589.0
16	4	0.94	27609.1
16	5	0.94	58431.0
16	6	0.94	109505.2
16	7	0.94	106926.1
18	3	0.93	106926.1
18	4	0.92	12434.9
18	5	0.94	12434.9
18	6	0.94	47378.5
18	7	0.94	42263.1
20	3	0.93	22920.8
20	4	0.88	6610.6
20	5	0.93	13458.9
20	6	0.94	22860.9
20	7	0.93	22241.0

chunks, GHOSTZ is executable even on a regular PC.

Table 4.10: Memory usage for database construction and protein sequence homology search with various database chunk sizes. The columns show the size of database chunks, memory required for constructing the index (GB), and memory required for the protein sequence homology search (GB). We searched the KEGG GENES (3.9 GB) database.

Tool (chunk size)	Memory size for constructing index (GB)	Memory size for protein sequence homology search (GB)
GHOSTZ (128 MB)	5.4	1.4
GHOSTZ (256 MB)	10.1	2.2
GHOSTZ (512 MB)	21.0	3.8
GHOSTZ (1 GB)	41.0	6.7
RAPSearch	6.9	4.1

Table 4.11: Computation time for database construction and protein sequence homology search with various database chunk sizes. The columns show the size of database chunks, computation time, and processing speed relative to GHOSTZ using 1 GB database chunks. We se arched the KEGG GENES (3.9 GB) database.

Tool (chunk size)	Computation time (sec.)	Acceleration ratio
GHOSTZ (128 MB)	545.2	0.88
GHOSTZ (256 MB)	488.2	0.94
GHOSTZ (512 MB)	479.1	0.96
GHOSTZ (1 GB)	460.8	1.00
RAPSearch	1285.5	0.35

4.4 Discussion

In the evaluation experiment, GHOSTZ demonstrated an approximately two-fold increase in speed compared to GHOSTZ without clustering, which can be probably attributed to the reduction in the number of ungapped extensions. To validate this hypothesis, we compared the total number of ungapped extensions required by each method. In the database subsequence clustering, similarity filtering requires computing time comparable to that for ungapped extension; therefore, we added the number of similarity filtering steps to that of ungapped extensions and found that the number of ungapped extensions could be reduced to approximately one-third of the original using database subsequence clustering. To evaluate the performance of similarity

Table 4.12: Computation time of seed search, similarity filtering, ungapped extension, and gapped extension. Computation time for each step as the percentage of total computation time is shown in brackets.

	GHOSTZ	GHOSTZ without clustering	GHOSTX	BLASTX
Seed search	42.2 (8%)	104.2 (11%)	298.6 (34%)	115173.0 (80%)
Similarity filtering	28.8 (10%)	-	-	-
Ungapped extension	236.8 (46%)	590.0 (60%)	104.7 (12%)	19768.5 (14%)
Gapped extension	159.6 (31%)	255.4 (26%)	425.9 (48%)	5873.3 (4%)
Others	47.4 (9%)	36.5 (4%)	53.2 (4%)	2435.2 (2%)
Total computation time	514.8 (100%)	986.1 (100%)	882.5 (100%)	143250.0 (100%)

filtering, we obtained the execution profiles of BLASTX, GHOSTX, GHOSTZ, and GHOSTZ without clustering. Seed search, similarity filtering, ungapped extension, and gapped extension of GHOSTX, GHOSTZ, and GHOSTZ without clustering were not used in inline expansion to obtain execution profile. These execution profiles were obtained from the calculation using 10,000 short DNA reads in SRR407548 as queries and KEGG GENES as a database. Table 4.12 shows execution profiles of GHOSTZ, GHOSTZ, GHOSTZ without clustering, and BLASTX. The seed search of GHOSTZ and GHOSTZ without clustering was faster than that of GHOSTX, because GHOSTZ and GHOSTZ without clustering used hash tables. In addition, ungapped extension of GHOSTZ was faster than that of GHOSTZ without clustering because of similarity filtering. Currently, ungapped extension is one of the primary bottlenecks in fast protein sequence homology searches. Thus, we consider that this effective decrease in ungapped extensions contributes to a significant increase in search speed observed when subsequence clustering for protein sequence homology searches is used.

GHOSTZ allows database indexes to be constructed anew, so that the users who need to consider other parameters can employ this method. The construction of the indexes for a 1 GB database requires approximately three hours of computation time. However, when a large number of DNA reads obtained using NGS are to be processed, the computation time for protein sequence homology searches is generally much longer than that required for database construction. Therefore, we think that in practice, the computational time spent in rebuilding database indexes and clustering is not likely to be a problem.

To further reduce the number of ungapped extensions, other clustering methods may be effective. However, the ungapped extension step takes about 46% of total time and, therefore, acceleration of only this step is insufficient to significantly increase search speed.

4.5 Summary

We developed a new protein sequence homology search algorithm with subsequence clustering, where we reduced the number of ungapped alignment extensions by clustering subsequences in a database and achieved a two-fold increase in processing speed without reduction in search sensitivity. The algorithm was designed for functional and taxonomic annotations in metagenomic analysis. The proposed database subsequence clustering method could also be useful in proteomics requiring a large number of sequence protein sequence homology searches.

Chapter 5

A GPU-Accelerated Protein Sequence Homology Search

5.1 Introduction

To improve the performance of protein sequence homology search, we developed two algorithms. However, it has limits to increase the acceleration. Therefore, we used GPU to improve protein sequence homology search.

In this study, we developed a protein sequence homology search algorithm suitable for GPU calculation and implemented it on GPUs, called GHOSTM. It accepts a large number of short reads produced by a NGS as the input and, like the BLASTX program, performs DNA sequence homology searches against a protein sequence database. We used NVIDIA CUDA to implement the GPU computing. The search system demonstrated a calculation speed that was 130 times faster with one GPU than BLASTX on a CPU. This system should enable researchers to analyze large amounts of metagenomic data from NGSs, even with a small-scale workstation.

5.2 General-Purpose Computing on Graphics Processing Units (GPGPU)

GPUs are architectures that were originally designed for graphics applications. However, new-generation GPUs have been transformed into powerful co-processors for general purpose computing, and their computational power supersedes that of CPUs. GPUs have already been used for several bioinformatics applications, such as CUD-

ASW++ [31, 32, 33] and GPU-HMMER [55]. These applications have successfully achieved more than a 5-fold increase in acceleration compared to their CPU-based counterparts. Using GPUs, the BLASTP program was also accelerated to create new applications, known as GPU-BLAST [54] and CUDA-BLASTP [30]. BLASTP performs protein versus protein sequence searches, whereas BLASTX conducts a translated DNA sequence search against a protein database with automatic translation of the query sequence into all six of the possible reading frames. However, the calculation speed of CUDA-BLASTP was only approximately 10 times faster than BLASTP on the CPU platform, and GPU-BLAST was only approximately 3 times faster. The small increase in speed was likely related to the BLAST search algorithm being complicated and inefficient when implemented on GPUs. Therefore, a new and efficient search algorithm optimized for GPU calculations is required.

5.3 Methods

Our protein sequence homology search tool was mainly composed of three components, as shown in Fig. 5.1. The first component searched the candidate alignment positions for a sequence from the database using the indexes. The second component calculated local alignments around the candidate positions using the Gotoh algorithm [17] for calculating the alignment scores. Finally, the third component sorted the alignment scores and output the search results

5.3.1 Construction of Database Indexes

Before searching a database, the indexes for all of the database sequences were constructed. All of the sequences in the database were connected to inserting delimiters to transform them into several long sequences. Index keys were generated for every offset of a k-mer, which is a k length sequence, in a database sequence. The position at which each key appeared was stored in the order in which it appeared in the database. For large database, the sequences in the databases were divided into several chunks because of the limitation of memory space. In a search process, the system searches for homologues for each database chunk by switching them and then merges the search results. GHOSTM automatically divides a database into chunks according to the upper limit of the database chunk size specified by the user.

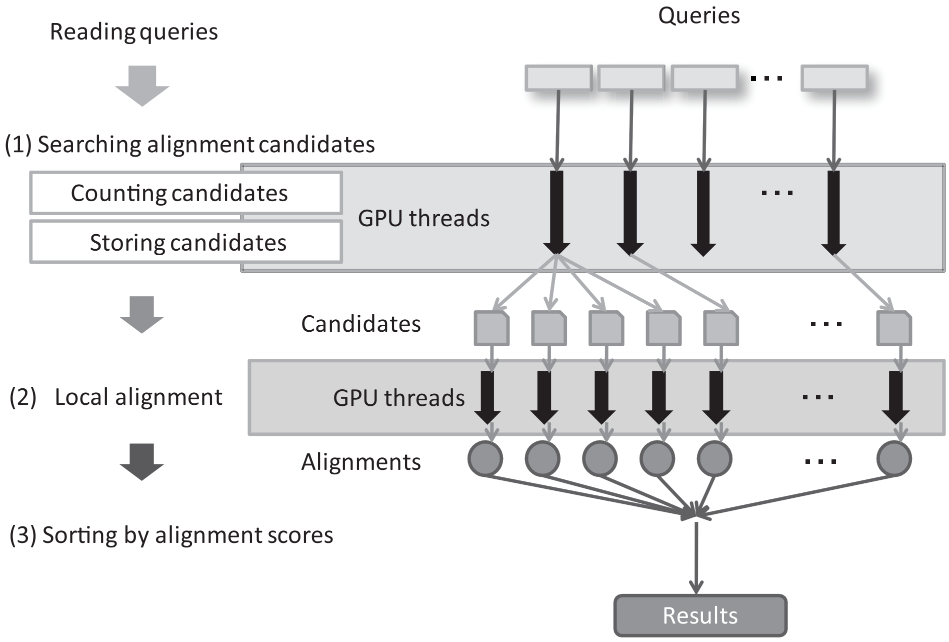


Figure 5.1: Data flow and processing within GHOSTM.

5.3.2 Search for Candidate Alignment Positions

The DNA query sequences were initially translated into protein sequences in all of the six open reading frames. GHOSTM uses the same method to translate DNA sequence as BLASTX.

The index keys of protein sequences were generated in the same way as the database indexes but with s character skips. These skips reduce the calculation cost at the expense of search sensitivity in the candidate search component. For confirming matches, a database sequence was first divided into regions of size r, and the key of each query was compared with the keys of the database sequences. If more than a threshold number t of keys matched in a region and the right adjacent region, the position was stored as a candidate alignment. Fig. 5.2 shows an example of a search result in which three candidate positions were reported with a threshold of $t = 2$.

5.3.3 Local Alignment

After searching for sequence alignment positions, optimal sequence local alignment was performed for the region around each candidate position using the Gotoh algorithm, and the alignment score for each candidate position was calculated. When calculating

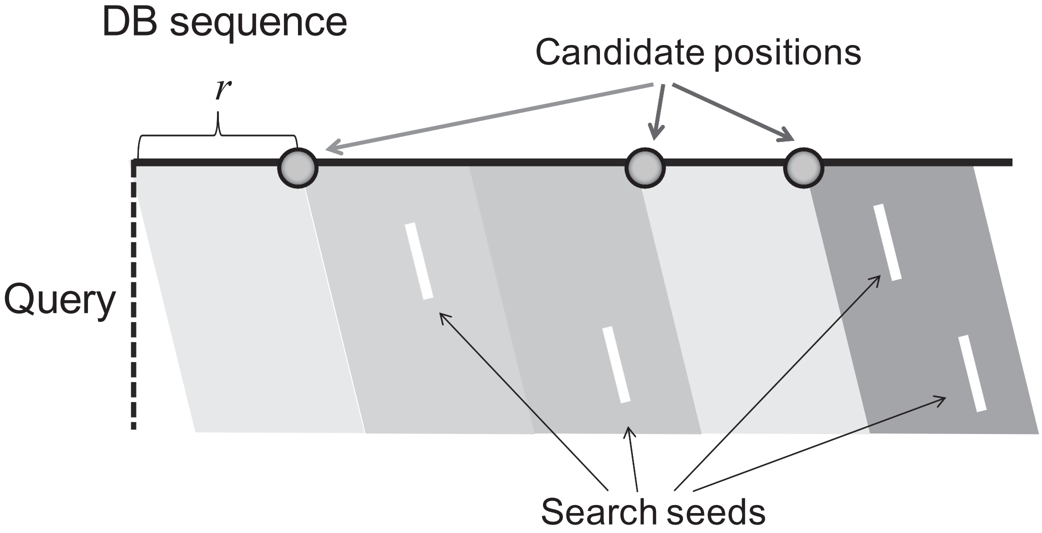


Figure 5.2: Search for candidate alignments.

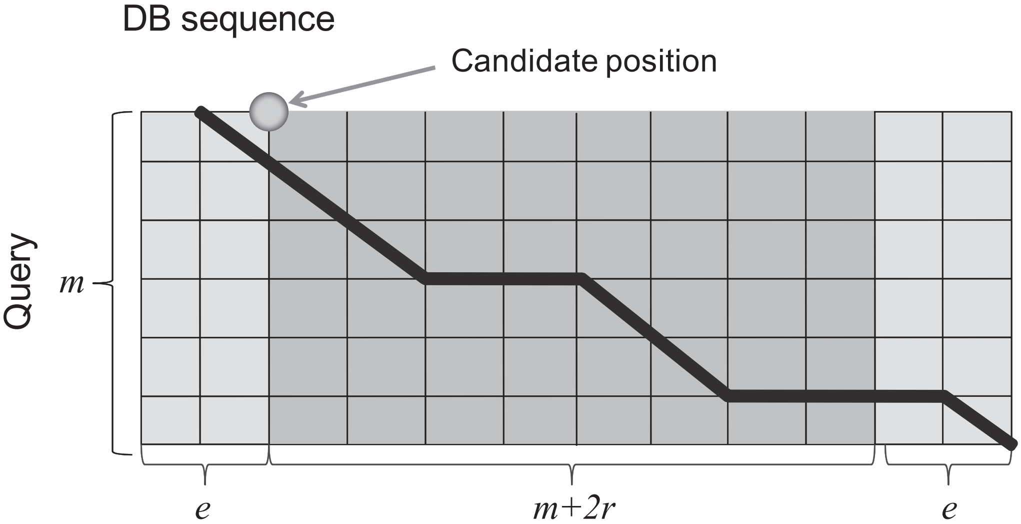


Figure 5.3: Calculation of an alignment in the region around a candidate position.

the local alignment, we restricted the alignment target of a database sequence to a small region of size $m + 2r + 2e$, where m was the length of the query and e was the extension width of an alignment region, as illustrated in Fig. 5.3.

5.3.4 Mapping to GPUs

Both the candidate search and local alignment components required a large amount of computing time. Therefore, we processed queries on both components in parallel and mapped them onto GPUs. Thus, multiple queries were simultaneously processed on different GPU cores. We used NVIDIA CUDA 2.2 to implement the GPU computing and mapped the two different calculation components as the two kernel functions.

The GPU computing program has several limitations, even with the use of current GPUs and CUDA. Thus, we introduced some techniques to our implementation. First, because it was impossible to access the host memory during GPU execution, the calculation results had to be stored to memory on a GPU. However, the size of the memory on a GPU is limited, and the global memory, which is the largest on a GPU, is also used for storing query sequences, database sequences and indexes. Furthermore, we could not know, a priori, the number of candidates and the size of the results to be stored when we generated a candidate for a large number of queries. Consequently, the storage of the results often failed because of the shortage of GPU memory. To overcome this problem, we first counted the number of candidates at the alignment position and then divided the queries into subqueries, whose results could be stored in the global memory of the GPU.

For the implementation of local alignment, a GPU-accelerated Gotoh algorithm has already been proposed [31, 32, 33]. However, this implementation was designed for alignments between long sequences and required the synchronization of multiple threads. Shorter sequences require more frequent synchronizations, which slows the calculation. Thus, in our proposed system, a thread was assigned to each candidate alignment position, and the synchronization among threads was removed. In the alignment process, all of the threads randomly and frequently accessed the scoring matrix. Thus, the matrix data were stored on the texture memory of a GPU because the access speed was much faster than the global memory of a GPU.

To utilize GPUs with CUDA, we must decide the number of grids, blocks, and threads. We fixed the number of grids, blocks, and threads to 1, 128, and 256, respectively. We optimized these parameters for the Tesla S1070, which we used. These parameters do not affect the performance significantly, but they should be optimized for other types of GPUs to achieve maximum performances.

5.4 Results

5.4.1 Datasets and Conditions

To evaluate the performance of GHOSTM, we compared its search sensitivity and computation time with the NCBI BLAST (version 2.2.25+) and BLAT (standalone package, version 34). We used protein sequences obtained from KEGG GENES as of November 2010 as the search target database. The number of sequences in the database was approximately 4,200,000, and the total length of these sequences was approximately 2,000,000,000 amino acids. We used DNA sequence reads obtained from a polluted soil metagenome study with Illumina/Solexa sequencing as the DNA query sequences. We used approximately 6,800,000 high-quality reads selected from approximately 20,000,000 reads that were obtained from the Illumina/Solexa sequence run. We selected reads that had a quality score greater than 15 (Q15 or over) over a continuous region of more than 60 bp. Thus, the lengths of the reads ranged from 60 to 75 bp. For all of the evaluations, we used the BLOSUM62 as the score matrix and performed all of the tests on a workstation with two dual core CPUs (3.2 GHz Dual-Core AMD Opteron 2224 SE) and a GPU server (1.44 GHz Tesla S1070), which included 4 GPUs.

5.4.2 Evaluation of Computation Time

We ran GHOSTM, BLASTX, and BLAT to measure their computation times. For comparing BLAT with GHOSTM, we used all of the 6,800,000 reads as query sequences. However, we used only 100,000 randomly selected reads as query sequences for comparing GHOSTM with BLASTX because the calculation cost of BLASTX is too excessive to perform millions of reads. As previously described, the queries were DNA reads, and the database was composed of protein sequences; thus, we executed the BLASTX program with the command line options “-outfmt 6 -seg no”, which instructed the program to output in tabular format. We did not use the SEG filter [57] because BLASTX sometimes fails to find significant hits with this filtering for short queries. We tested BLASTX with 1 thread and 4 threads. The BLAT program does not include a function to translate DNA reads to protein sequences. Therefore, we translated the DNA reads into protein sequences based on the standard codon table. We executed the BLAT program with the command line option “-q=prot -t=prot -out=blast8”, which instructed the program to use protein queries as well as a protein database and to output data in the BLASTX tabular format. The BLAT program does

Table 5.1: Computation time for 100,000 reads. The first, second, third, and fourth columns show the name of each program, the number of GPUs used for the calculation, the computation time, and the acceleration in processing speed relative to BLASTX using 1 thread, respectively.

	Computation time (sec.)	Acceleration ratio
GHOSTM (1 GPU)	2,855	129.5
GHOSTM (4 GPUs)	909	406.7
BLAT	9,898	37.3
BLASTX (1 thread)	369,678	1.0
BLASTX (4 threads)	102,255	3.6

not support a multi-core processor. Thus, we executed the BLAT with only 1 thread. For GHOSTM, we used the command line options “db -k 4 -l 128” for constructing database indexes: the length of the search seeds was $k = 4$, and the size of a database chunk was 128 Mbp. Using these parameters, GHOSTM generated 16 database chunks for the KEGG GENES database. The command line options “aln -l 128 -s 2 -r 4 -e 2 -t 2” were used for the search process, with character skips at $s = 2$, search region size at $r = 4$, extension size at $e = 2$, and the number of required matches at $t = 2$. We determined these parameters based on the balance between the prediction sensitivity and computational time. The performance of GHOSTM with other parameters is discussed in the following section.

Table 5.1 shows the computational times for BLASTX, BLAT, and GHOSTM for 100,000 reads. The GHOSTM program achieved a calculation speed 129.5 and 35.8 times faster than the BLASTX using 1 thread and 4 threads, respectively. Moreover, GHOSTM was approximately 3.4 times faster than BLAT. In addition, GHOSTM implemented on a system with 4 GPUs showed a processing acceleration that was 406.7 and 112.5 times faster than the computational speed of BLASTX using 1 thread and 4 threads, respectively. Thus, GHOSTM implemented on a system with 4 GPUs showed an acceleration that was approximately 3.1 times greater than the speed achieved using a single GPU.

Table 5.2 shows the computational times required for BLAT and GHOSTM to analyze the 6,800,000 reads. The GHOSTM program was 4.2 times faster than the BLAT program. Moreover, GHOSTM implemented on a system with 4 GPUs showed a processing acceleration that was 14.6 times faster than BLAT. GHOSTM on a4 GPUs system was 3.5 times faster than the 1 GPU system for the 6,800,000 reads, while the

Table 5.2: Computation time for approximately 6,800,000 reads. The first, second, third, and fourth columns show the name of each program, the number of GPUs used for the calculation, the computation time, and the fold increase in the acceleration in the processing speed relative to BLAT, respectively.

	Computation time (sec.)	Acceleration ratio
GHOSTM (1 GPU)	166,740	4.2
GHOSTM (4 GPUs)	47,995	14.6
BLAT	699,300	1.0

increase in speed with 4 GPUs was approximately 3.1 for the 100,000 reads.

5.4.3 Evaluation of Search Sensitivity

To evaluate the search sensitivity, we used the search results obtained with SSEARCH, and these results were assumed to be the correct answers. We analyzed the performance of a particular method in terms of the fraction of its results that corresponded to the correct answers obtained by SSEARCH.

For this analysis, we used only 10,000 randomly selected reads because the calculation cost of the local alignment by SSEARCH was excessive. We translated the DNA reads into protein sequences in the same manner used for the evaluation of the computation time with BLAT because SSEARCH does not have a translation function. For these protein sequences, we executed the BLASTP program with the command line options "-outfmt 6 -seg no -comp_based_stats 0". We did not use composition-based statistics [6] because this method was not employed in the default configuration of BLASTX. We also did not use the SEG filter. For GHOSTM and BLAT, we used the same command line options that were used for the evaluation of the computation time.

Fig. 5.4 shows the evaluation of the results of the search sensitivity. The search sensitivity of GHOSTM was clearly higher than BLAT. However, the sensitivity of GHOSTM was lower than BLASTP, especially for those hits whose scores were below 40. However, low-scoring hits (e.g., < 50) are generally not used in practice because such hits can occur by chance. With the exception of the low-score hits, GHOSTM successfully identified more than 90% of the hits identified by SSEARCH. This result suggests that GHOSTM is sufficiently accurate for general usage.

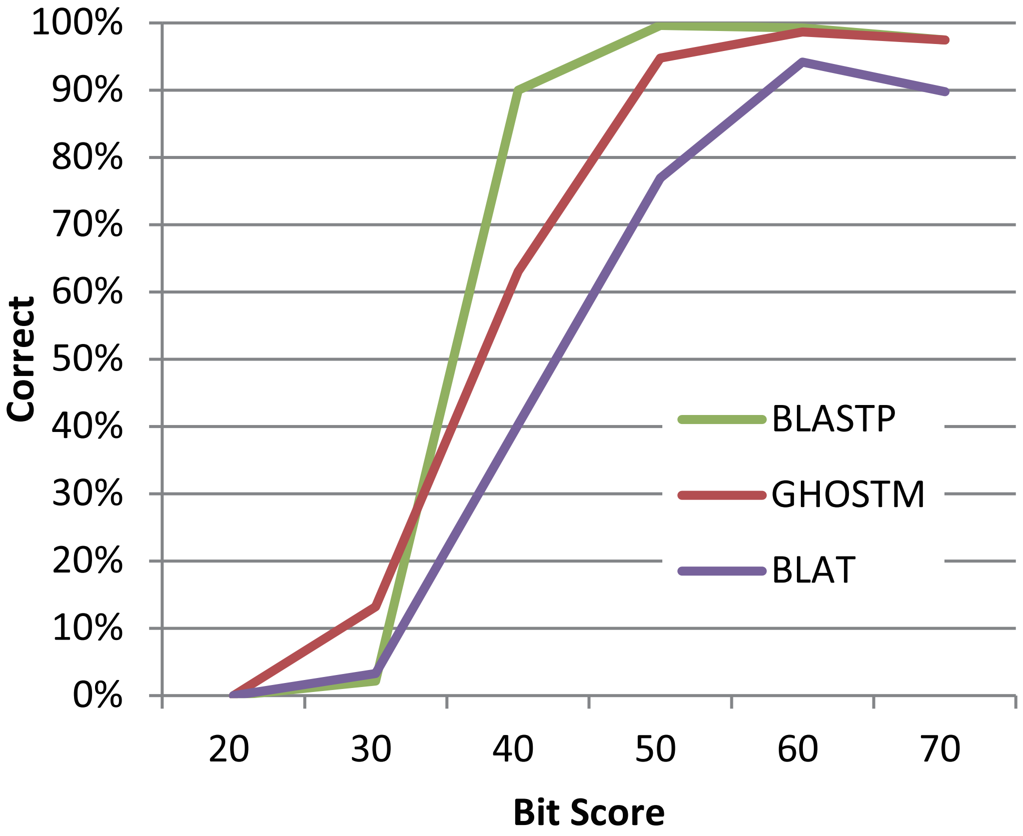


Figure 5.4: Search sensitivity. The vertical axis shows the percentage of results for each method that corresponds to the correct answers. The horizontal axis shows the bit scores of the alignments.

5.4.4 Relationships between Search Parameters and Their Sensitivity and Computation Time

To determine the relationships between search parameters and their sensitivity and computation time, we executed GHOSTM by changing one of its parameters from default to different values and measured the computation time and search sensitivity. To evaluate the search sensitivity and computation time, we used the same method used for comparing BLASTX, BLAT and GHOSTM. We tested the following parameters and compared their computation times: $k = 3, 4$, and 5; $s = 2, 3$, and 4; $r = 2, 4$, and 8; $e = 0, 2$, and 4; and $t = 1, 2$, and 3.

Fig. 5.5 and 5.6 show the acceleration in processing speed relative to GHOSTM with default parameters for different search regions size r and extension size e. As shown in the figure, search region size and extension size do not significantly change the search sensitivity and computation time. However, other parameters, including the length of the search seed k, character skips s, and the number of required matches t, significantly change the performance. Using $k = 5, s = 4$, or $t = 3$, the acceleration of BLASTX increases to 931.2, 329.5, and 239.6, respectively (Figs. 5.7A, 5.8A and 5.9A). However, the search accuracies decrease to levels similar to BLAT (Figs. 5.7B, 5.8B and 5.9B). With these parameters, GHOSTM often fails to find search seeds, including significant hits, which causes this low search sensitivity. We believe that these search accuracies are insufficient for metagenomic analysis; thus, we did not use these settings as default settings. GHOSTM with $k = 3, s = 1$, or $t = 1$ shows good search sensitivity that is comparable with BLASTX (Figs. 5.7B, 5.8B and 5.9B.). However, the calculation speed is slower, and the acceleration of BLASTX with 1 thread is 5.2, 22.2, and 5.2, respectively (Figs. 5.7A, 5.8A and 5.9A). These accelerations are smaller than BLAT; thus, we did not use these parameters as default settings.

5.5 Discussion

GHOSTM clearly outperformed BLASTX in reducing the computation time for conducting protein sequence homology searches. The reason for the acceleration in computation time was that the system simultaneously processed multiple queries on different GPU cores (the Tesla S1070 has 240 cores per GPU). Importantly, the GPU system requires a sufficient number of queries, and in fact, when using only one query sequence, the calculation of GHOSTM becomes much slower than BLASTX. Table 5.3 shows the relationship between the number of query sequences and the acceleration

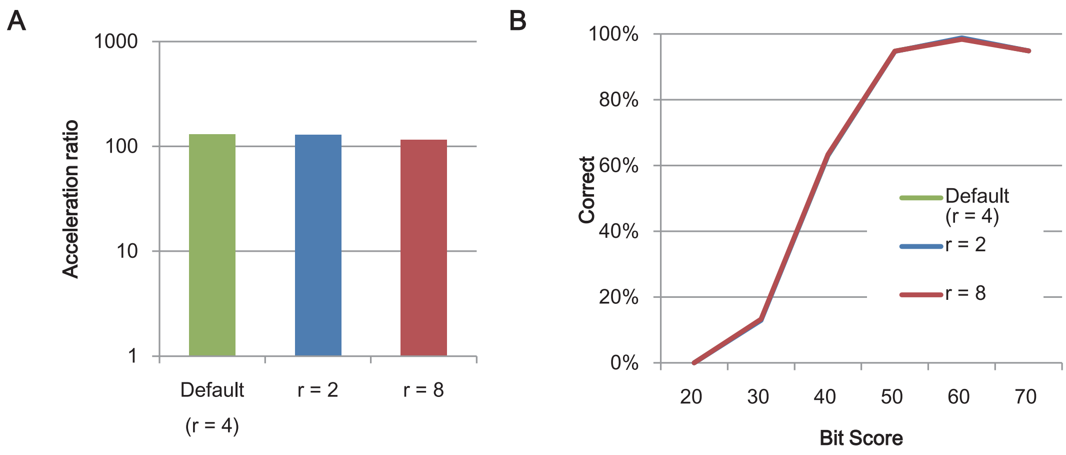


Figure 5.5: The relationships between search speed and sensitivity and the search region size r. (A) The acceleration in processing speed relative to BLASTX using 1 thread and (B) search sensitivity.

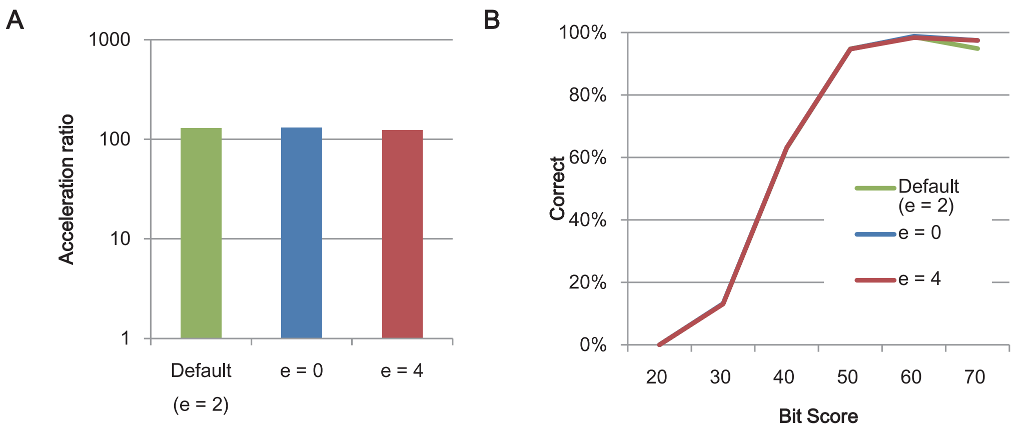


Figure 5.6: The relationships between search speed and sensitivity and the extension size e. (A) The acceleration of processing speed relative to BLASTX using 1 thread and (B) search sensitivity.

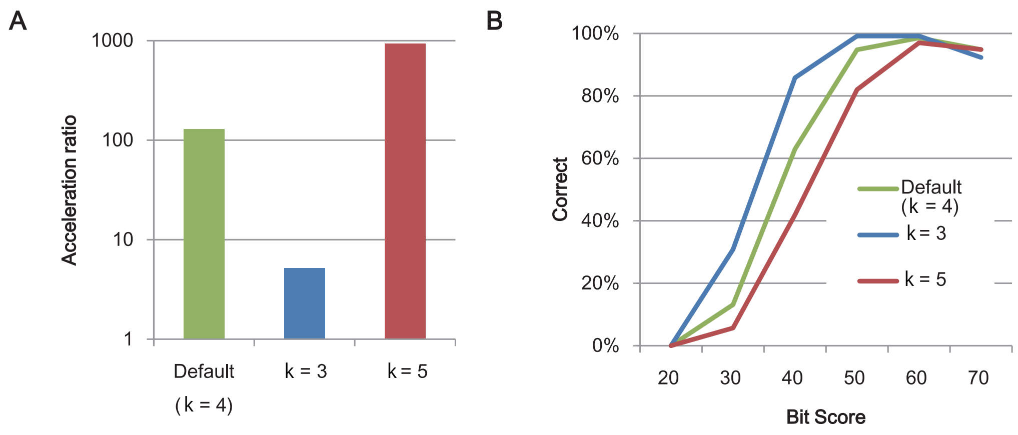


Figure 5.7: The relationships between search speed and sensitivity and the length of search seeds k. (A) The acceleration of processing speed relative to BLASTX using 1 thread and (B) search sensitivity.

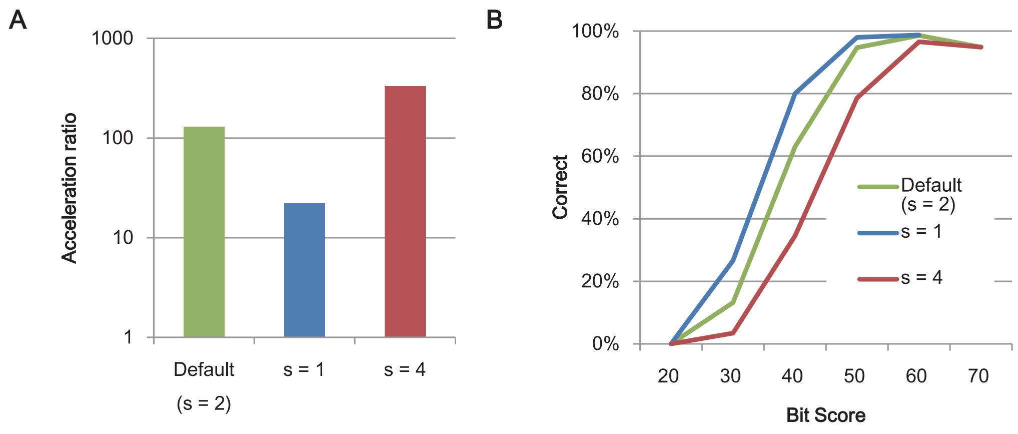


Figure 5.8: The relationships between search speed and sensitivity and the character skips s. (A) The acceleration of processing speed relative to BLASTX using 1 thread and (B) search sensitivity.

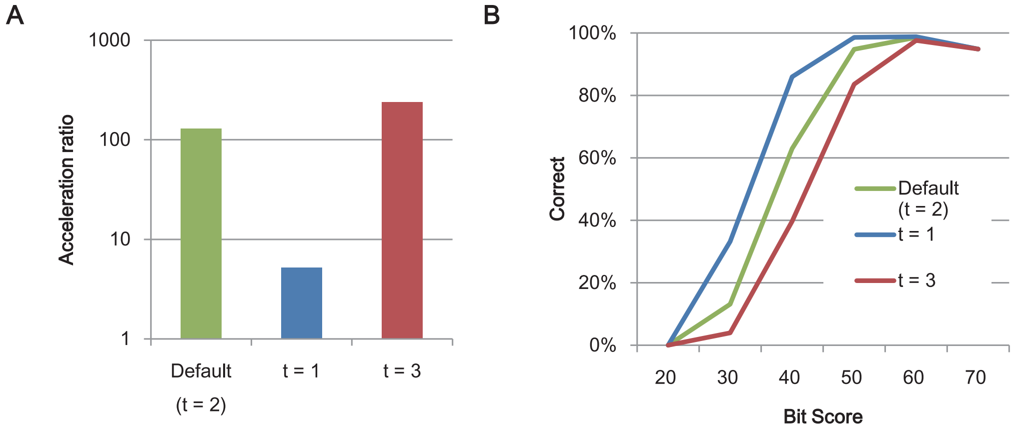


Figure 5.9: The relationships between search speed and sensitivity and the number of required matches t. (A) The acceleration of processing speed relative to BLASTX using 1 thread and (B) search sensitivity.

Table 5.3: Computation time and acceleration of GHOSTM on a 1 GPU system relative to BLASTX for different query numbers.

#queries	GHOSTM (sec.)	BLASTX (sec.)	Acceleration ratio
1,000	213	4,180	19.6
10,000	422	37,167	88.0
100,000	2,855	369,678	129.5

in computation time. This result explains why GHOSTM on a system with 4 GPUs achieved a calculation speed that was only 3.1 times faster than GHOSTM on a system with 1 GPU for the small query set. However, the calculation speed of GHOSTM on a 4 GPUs system was approximately 3.5 times faster than the speed obtained on the 1 GPU system when the number of queries was sufficient, as shown for 6,800,000 reads. Thus, we suggest that the acceleration of GHOSTM will increase almost linearly as a function of the number of GPUs in practical situations in metagenomic analysis projects comprising hundreds of millions of reads.

In addition to the number of queries, GHOSTM had another restriction because it assumed that the length of all of the queries was approximately the same. For calculating the local alignment of each query, GHOSTM takes a GPU memory allocation plan according to the length of the longest query. Once GPU memory is allocated according to the maximum memory consumption case at first, GHOSTM can reuse the allocated space until the end of calculation, with avoiding overhead of GPU memory re-allocation. Thus, if the lengths of the queries were markedly different, GHOSTM required too much memory, which decreased the number of queries that GHOSTM could process concurrently. However, the number of reads from NGSs is large, and the lengths of the reads are approximately the same. Therefore, these two restrictions are generally satisfied, and we predict that they will have little impact on the calculation speed of GHOSTM.

5.6 Summary

We developed a GPU-optimized algorithm to perform sensitive protein sequence homology searches and implemented the system as GHOSTM. Currently, sequencing technology continues to improve, and sequencers are increasingly producing larger and larger quantities of data. This explosion of sequence data makes computational analysis with contemporary tools more difficult. We developed GHOSTM, which is a cost-efficient tool, and offer this tool as a potential solution to this problem.

Chapter 6

A Protein Sequence Homology Searches with Clustering Subsequences Technique on GPUs

6.1 Introduction

GHOSTM enabled us to perform fast protein sequence homology search using GPUs. However, the analysis of large metagenomic data, a faster protein sequence homology search tool is still required. To accelerate protein sequence homology search, we developed a GHOSTZ-based protein sequence homology search algorithm with GPUs and designated it as GHOSTZ-GPU.

GHOSTZ has several bottlenecks. Table 6.1 shows the proportion of calculation time required for each step of GHOSTZ. The ungapped extension step takes most (46.0%) of total calculation time. However, mapping of ungapped extension to GPUs is insufficient to significantly improve search speed, because the other steps such as distance calculation and gapped extension also consume considerable time. To improve search speed with the GPUs, the mapping of these steps, including distance calculation, ungapped extension, and gapped extension, onto the GPUs is obviously critical for achieving significant process acceleration.

For GPU implementation, we used NVIDIA CUDA 6.0. A current computing system often has multiple CPU cores and multiple GPU cards in a computing node; thus, it is important to fully exploit such a computing environment. Therefore, we targeted a computing node with multiple CPU cores and multiple GPU cards.

Table 6.1: The execution profile of GHOSTZ calculation on 1 thread. This profile was obtained by the calculation using 10,000 randomly selected short DNA reads in soil microbiome metagenomic sequences (accession number, SRR407548, read length = 150 bp) as queries and the KEGG GENES (released May 2013) as a database. The profile was obtained on a workstation with a 2.93 GHz Intel Xeon 5670 processor and 54 GB memory. GHOSTZ is compiled by GCC (version 4.3.4) with the -O3 optimization option. To obtain the profile, the functions of distance calculation, ungapped extension, and gapped extension were not inlined.

	CPU time (sec.)	Ratio (%)
Distance calculation	28.9	5.6
Ungapped Extension	236.8	46.0
Gapped Extension	159.6	31.0
Others	89.4	17.4
Total	514.8	100.0

6.2 Methods

The flow of GHOSTZ-GPU is shown in Fig. 6.1. We mapped the following steps of GHOSTZ algorithm to the GPUs: distance calculation, ungapped extension, and gapped extension. CUDA programs contain functions performed on the GPU called a kernel. Kernels represent operations launched by a single CPU thread and are invoked as a set of concurrently executing threads on the GPU. These threads are organized in a hierarchy consisting of thread blocks and grids. A thread block is a set of concurrent threads and a grid is a set of independent thread blocks. CUDA uses several memories such as global memory, local memory, shared memory, and register. Global memory is used for communication between the CPU and GPU. Local memory stores local variables of a thread, if registers are not used. Although global and local memories are larger than the other GPU memories, the accesses to these memories are slow. Therefore, it is important for GPU calculations to reduce the number of accesses to these memories. It is often achieved by using shared memory, which is smaller than global and local memories and the access is faster; this memory is also used for communication among threads in a block. Distance calculation, ungapped extension, and gapped extension in GHOSTZ require access to sequence data in global memory. Therefore, GHOSTZ-GPU is optimized for the efficient access to the data stored in global memory on the GPU. Moreover, we reduced the computation time required for synchronization

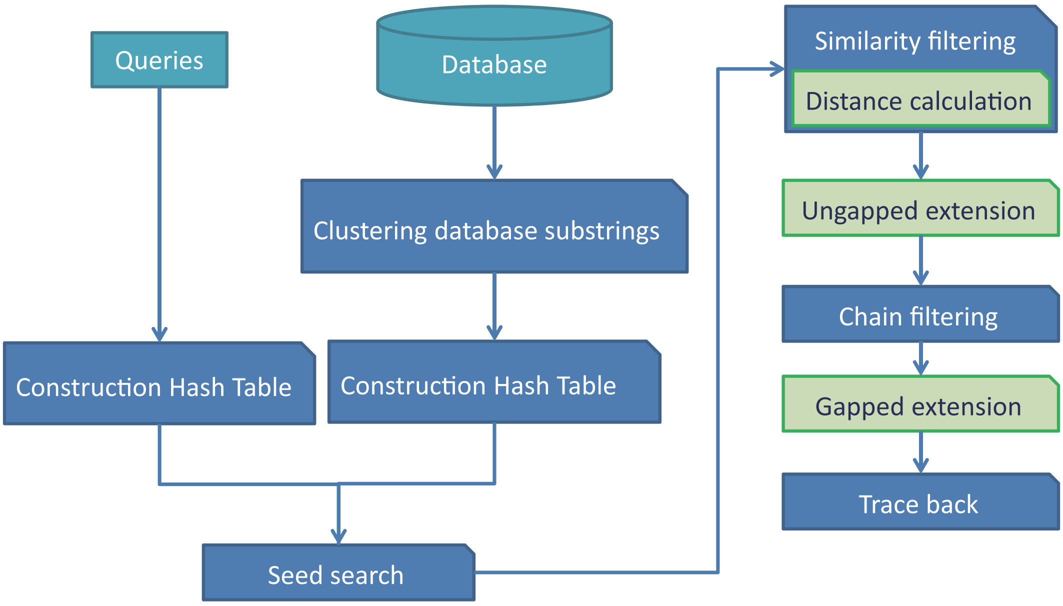


Figure 6.1: The flow of GHOSTZ-GPU. The green boxes are steps to be mapped into the GPU.

to provide full use of such a computing environment. This is achieved via reduction of inactive threads in gapped extension and usage of asynchronous executions on the CPU and GPU implemented in GHOSTZ-GPU.

6.2.1 Distance Calculation

Distance calculation is a part of similarity filtering. The inputs of distance calculation are query sequences, database sequences, a reduced amino acid alphabet, and start positions of subsequences. Each distance calculation is performed independently by different threads on the GPU. However, when each thread calculates different distances in a block, the access to query or database memory occurs randomly in each step. To fully utilize GPU efficiency, it is important to limit random accesses to global memory. To achieve this, we used two approaches: vectorized memory access and group memory access.

In GHOSTZ-GPU, character size in protein sequence is 5 bits, because the size of protein alphabet is 20. Therefore, 8 bits of memory is enough for one character in protein sequence. However, if 8 bits are used, multiple global memory accesses are required for protein sequence. To reduce the number of global memory accesses, larger-size memories (32 or 64 bit) are often used. In CUDA programming, such an

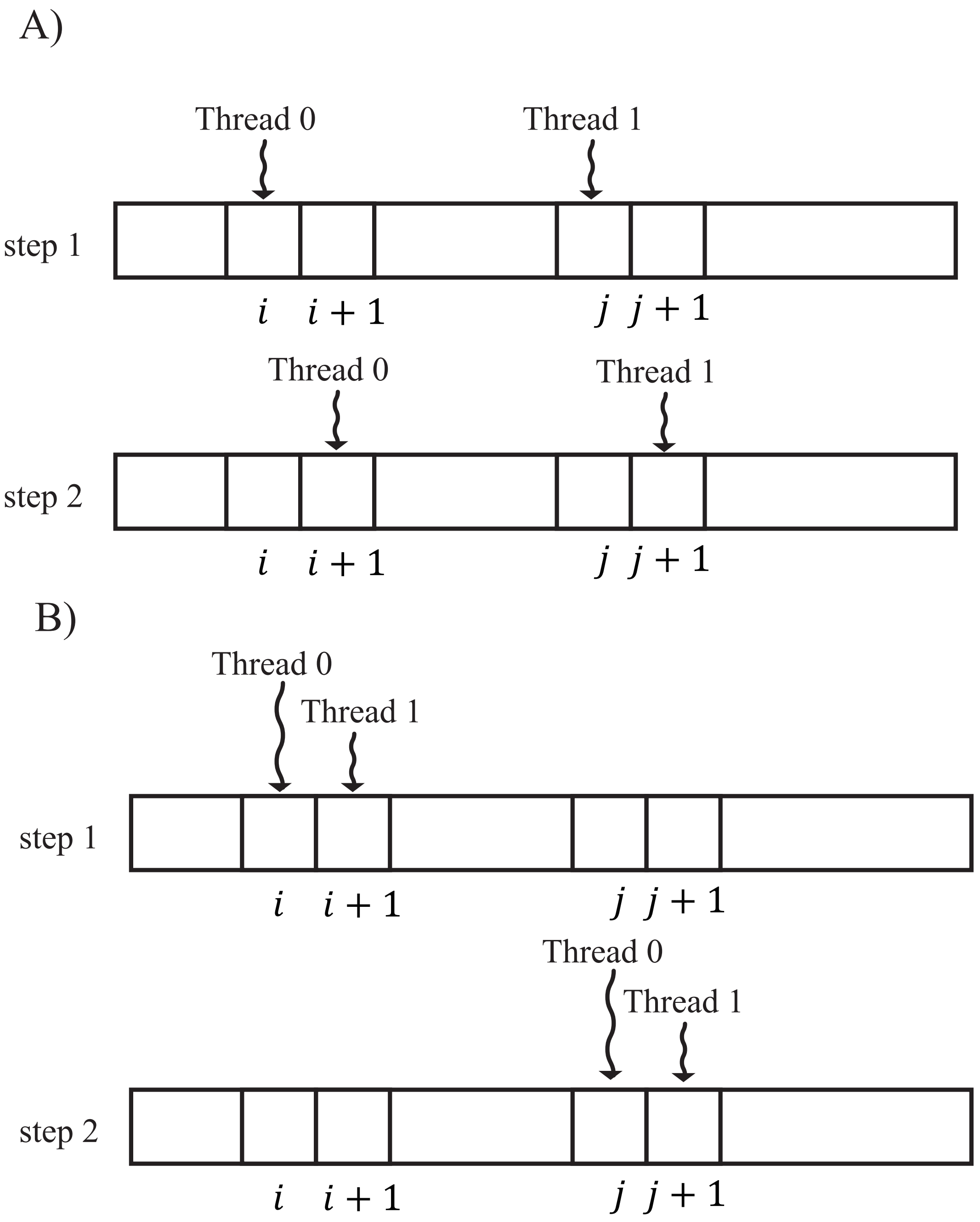


Figure 6.2: The examples of accessing sequence data. A) An example of accessing sequence data without group memory access. B) An example of accessing sequence data with group memory access.

approach is generally called vectorized memory access. When we use this method, the accessed data have to be assigned to a consecutive region in global memory. Let w be the number of characters for a single access and L be the length of sequence to compute distance and also the length of subsequence for clustering. By this memory access, the maximum number of global memory accesses is $\lceil (L + w - 1)/w \rceil$. In GHOSTZ-GPU, sequence data are allocated to consecutive global memory regions; therefore, we can use vectorized memory for accessing sequence data. GHOSTZ-GPU uses $L = 10$ and 64 bit access for protein sequence; in this case, w is 12. Therefore, the maximum number of global memory accesses is reduced by 1/5.

Moreover, to accelerate global memory access, we propose group memory access. As described above, sequence data were assigned to a consecutive region. We required up to $\lceil (L+w-1)/w \rceil$ global memory accesses for each sequence data for different distance calculation by vectorized memory access. To reduce the number of accesses, we divided threads in a block into smaller groups to use coalesced memory access for sequence data. When the threads in a block access the same region in global memory, these memory accesses are often coalesced into a single transaction. It is called coalescing memory access. Therefore, when the threads in a group access consecutive region in global memory, coalescing memory access is used for these accesses and decrease the number of global memory accesses. Examples of memory access without and with coalescing are shown in Fig. 6.2. For group memory access, we used shared memory for temporarily storage of sequence data and for communication among threads in a group. GHOSTZ uses $L = 10$. The size of a character in the GPU is 5 bits; therefore, when GHSOTZ-GPU uses 64 bit memory access, w is 12. In this case, the number of memory accesses required without and with group memory access is two and one, respectively.

6.2.2 Ungapped Extension

Each ungapped extension is performed independently by different threads on the GPU. The calculation of ungapped extension requires the following six parameters: seed, query ID, query sequence, database sequence, X-dropoff value, and score scheme. Among them, query sequence, database sequences, X-dropoff value, and score scheme parameters are used in any ungapped extension calculation, and are, therefore, stored constantly in global memory. The other data, i.e., seed and query ID, are sent to the GPU before performing each ungapped extension. After each ungapped extension, the extension score is also sent to the CPU. However, the number of ungapped extensions in GHOSTZ is large and requires long computation time for data transfer. To reduce the amount of data transferred between the CPU and GPU, we used two approaches, setting query data on the GPU and sending only the next-step flag instead of ungapped extension score.

The data sent to the GPU for ungapped extension are seeds consisting of start positions for query and database subsequences, and query IDs. The data on queries are larger than those on the database; therefore, GHOSTZ-GPU reduces query data to be transferred to the GPU. In GHOSTZ seed search, the database is represented by an outer loop and query is represented by an inner loop to optimize memory access in ungapped extension in Fig. 4.7 in Section 4.2.2. These loops are exchanged in GHOSTZ-GPU because the same query data in seeds are repeated in the list of seeds. Therefore, only query positions and their repeat counts in hash tables are required to be sent to the GPU. Using these data, start positions of query subsequences and query IDs are set on the GPU. As the size of query hash tables is smaller than that of query subsequence start positions and query IDs, the total data transferred to the GPU are decreased.

Moreover, to reduce the size of transferred data, a flag indicating that the seed is taken to the next step is sent instead of the ungapped extension score. The score of ungapped extension is used to determine whether the process goes to the next step; therefore, this selection is performed on the GPU and only the result is sent to the CPU. To make this decision, thresholds are required and they are sent to GPU before ungapped extensions. The total size of these thresholds is smaller than that of ungapped extension scores, thus reducing the data transferred to the GPU.

Vectorized memory access and group memory access are also effective in ungapped extension. However, because X-dropoff is used in this calculation, the length of alignment in ungapped extension is not preliminarily determined. Therefore, in GHOSTZ-GPU, the number of group members is four. If the length used in ungapped extension is more than four, GHOSTZ-GPU uses only vectorized memory access for the remaining sequence data.

6.2.3 Gapped Extension

Ungapped extension excludes useless seeds produced by seed search and similarity filtering. Then, similar nearby seeds are brought together by a chain filter. Finally, these seeds are extended with gaps in gapped extension. In gapped extension, GHOSTZ employs X-dropoff the same way as does BLAST [5]. By using an appropriate X-dropoff value, we can save time to compute gapped extension. Each gapped extension is performed independently; therefore, these calculations are done by different threads on the GPU. However, the computation time of gapped extension for each seed is different. Several threads in a block must execute the same instruction at any given time, which results in branch divergence. For example, when some threads in a block run at "if" instruction, they split in two for the branch, and all paths are executed sequentially, even though each thread executes only one paths. When threads run at "while" instruction, they wait for other thread executions to end. Branch divergence causes an increase of inactive threads on the GPU and extends computation time. Therefore, branch divergence should be reduced. The primary cause of the problem is size difference among S, I, and D in gapped extensions. The order of calculating gapped extension in GHOSTZ is shown in Fig. 6.3A. In gapped extension, query sequence length is represented by the inner loop. Query length affects computing time more than database sequence length. For better load balancing, GHOSTZ-GPU sorts seeds by query length and then assigns a seed to a thread on the GPU in order. Therefore, in GHOSTZ-GPU, the computation times of gapped extensions are sorted.

In gapped extension, global memory access consumes a large portion of total computation time. Therefore, vectorized and group memory accesses are used in gapped extension. However, accessing alignment matrices in gap extensions, S, I and D, also takes significant computation time. When only the gapped alignment score is calculated, we do not need to store all data in these matrices, because not all data are required to compute $S[i, j], I[i, j]$ and $D[i, j]$. Thus, GHOSTZ-GPU only stores previous columns of S and I in local memory. The length of S and I columns depends on query length and is generally short in current metagenomic analysis. However, accesses to local memory in the GPU are slower than those to register or shared memory. For accelerating gapped extension process, a reduction in the number of accesses to local memory is required. Therefore, to compute gapped extension we used shared memory and added another loop for calculating gapped extension. The calculation flow is shown in Fig. 6.3. The additional loop is short and requires small memory, which can be assigned to shared memory. The loop length is four in GHOSTZ-GPU. The shared memory in gapped extension is reused in group memory access. Therefore, an additional shared memory allocation for this loop is not needed.

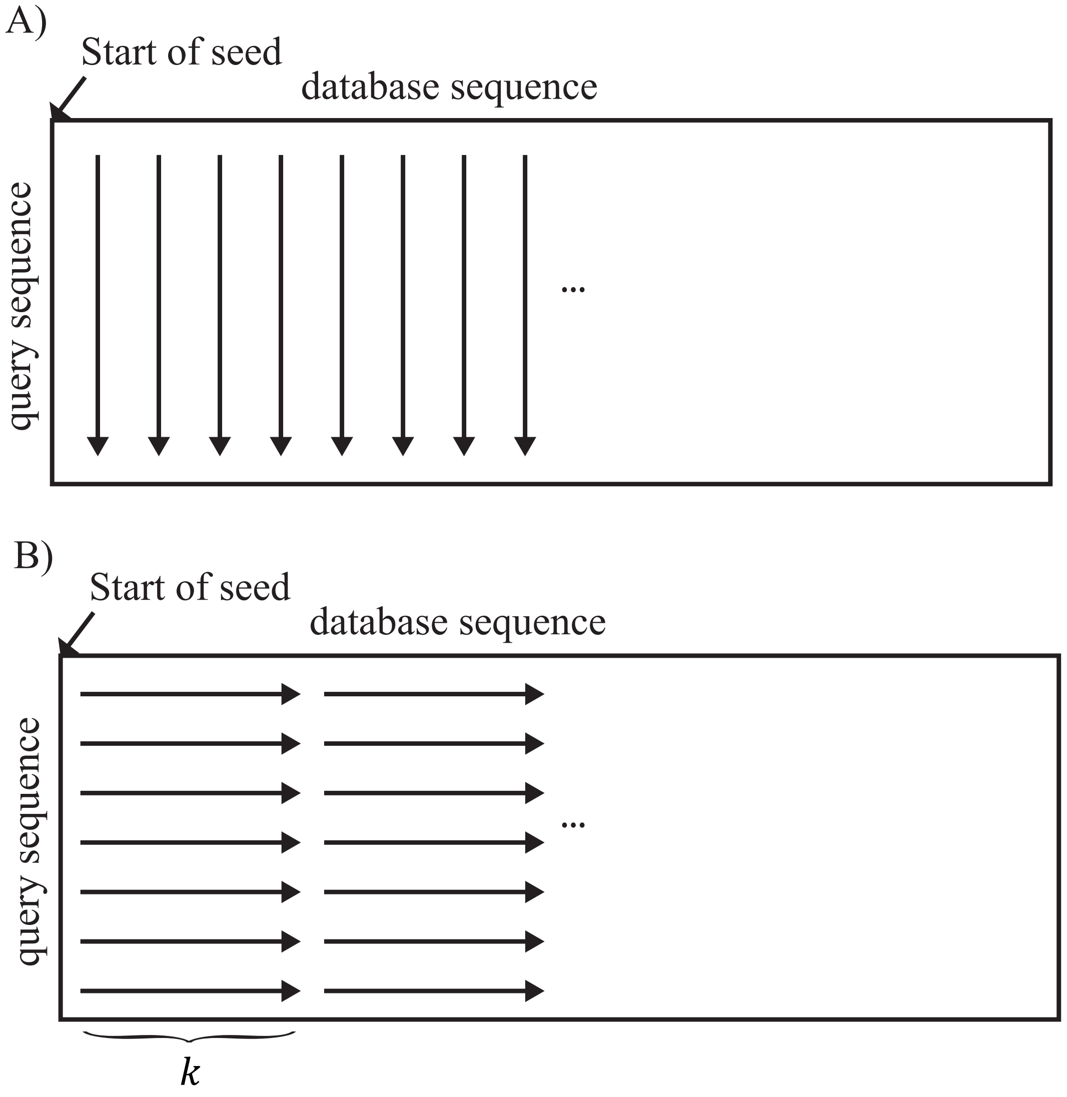


Figure 6.3: The examples of gapped extension on the GPU. A) An example of accessing sequence data without the short loop. B) An example of gapped extension with the short loop.

6.2.4 Asynchronous Execution on CPU and GPU

To make full use of a computing environment with the GPUs, overlapping between CPU and GPU calculations is essential. GHOSTZ-GPU divides the process with the CPU and GPU into two main phases. The first phase consists of seed search, similarity filtering, and ungapped extension and the second phase includes chain filtering and gapped extension. Threads on the CPU are calculated independently in each phase. Each thread on the CPU has different global memory on the GPU. If multiple GPUs are used, each GPU is assigned to almost the same number of threads on the CPU. To achieve overlapping between CPU and GPU calculations, each thread on the CPU applies double buffering technique to CPU and GPU memories, which are used in all steps on the GPU.

The first phase is shown in Fig. 6.4. Seed search for distance calculation is performed. Then, distances for similarity filtering are calculated on the GPU. Seed search against hash table B_e is performed on the CPU simultaneously with this GPU calculation, because this seed search is independent of similarity filtering. If distance calculation is finished on the GPU, ungapped extension calculation is started on the GPU. If seed search against B_e is finished on the CPU, seed search and similarity filtering for hash tables B_r and B_m are performed on the CPU. Then, the seeds from tables B_r and B_m are built and ungapped extensions for these seeds are performed. This phase is continued until the process for all query subsequences are complete.

The second phase is shown in Fig. 6.5. Chain filtering is performed on the CPU. If the memory is filled up with seeds, they are sorted by query length for alignment and then used to perform gapped extension. This phase is continued until the process for all seeds is complete.

6.2.5 Optimization of Loading Database

As protein sequence homology search becomes faster by GPU calculation, loading a database, including indexes, takes a large portion of computation time. Therefore, GHOSTZ-GPU uses a thread to load database. While the other threads perform protein sequence homology search against a database chunk, this thread loads the next database chunk. By this approach, the computation time spent on database loading is hidden in protein sequence homology search.

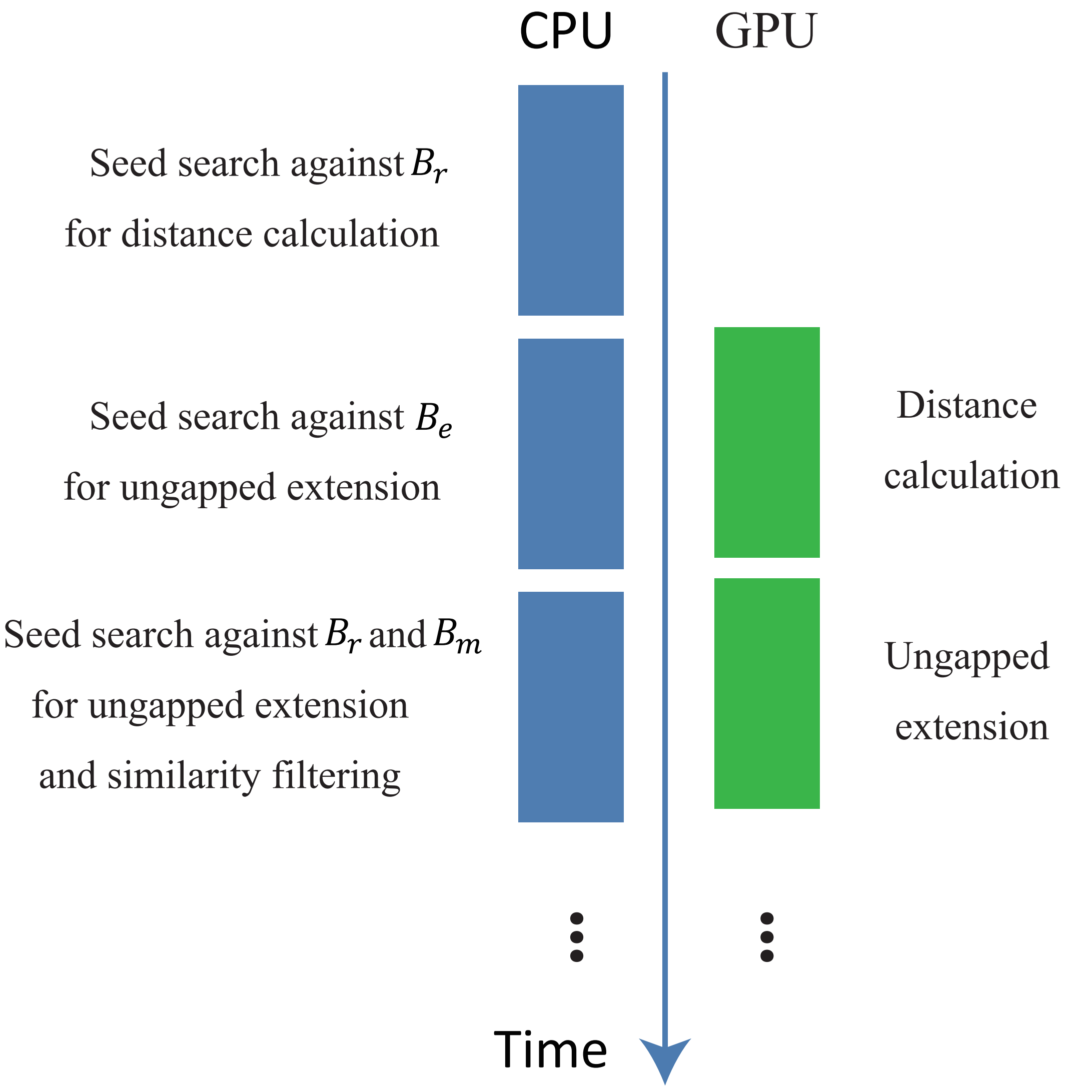


Figure 6.4: The flow of the first phase in GHOSTZ-GPU

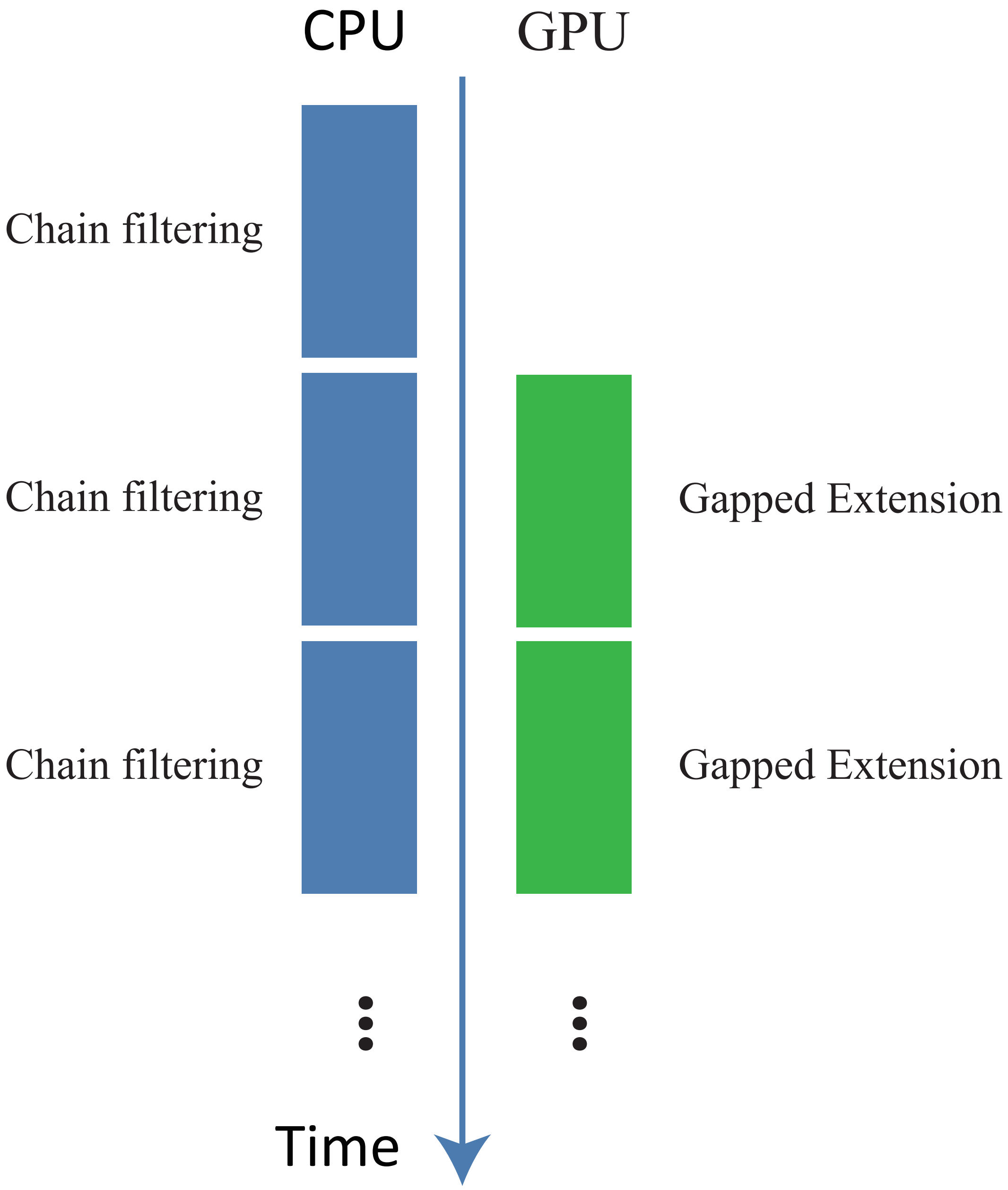


Figure 6.5: The flow of the second phase in GHOSTZ-GPU

6.3 Results

6.3.1 Datasets and Computing Environment

We evaluated the performance of GHOSTZ-GPU using the same dataset as in Section 4.3 expect the number of queries, because 10,000 randomly selected short DNA reads represented too small a sample to measure correct computation time. To evaluate computation time, 1,000,000 randomly selected short DNA reads were used for all datasets. However, only 10,000 randomly selected short DNA reads from SRR407548 were used to evaluate search sensitivity of GHOSTZ-GPU because of computation cost. The evaluation tests were performed on the same workstation mentioned in Section 4.3, which had three NVIDIA Tesla K20X with 6 GB memory.

The parameters of GHOSTZ and GHOSTZ-GPU were the same as in Section 4.3. To perform GHOSTZ and GHOSTZ-GPU, similar sequences were arranged close to each other in the database file, based on the results of CD-HIT. Since the optimization of database loading is also effective for GHOSTZ, we applied this approach to GHOSTZ.

6.3.2 Evaluation of the Acceleration by GPUs

To evaluate acceleration by the GPUs, we ran GHOSTZ-GPU and GHOSTZ using their default parameters, except for the multithreading option. In this evaluation, we used only short DNA reads from soil microbiome metagenomic sequences (SRR407548) as queries. Fig. 6.6 shows computation time for each program with 1, 2, 4, 8, and 12 threads and 1, 2, and 3 GPUs. GHOSTZ and GHOSTZ-GPU with 12 threads demonstrated the best performance. In addition, with 12 threads, GHOSTZ-GPU showed an acceleration of approximately 3.9, 6.3, and 7.1 times with 1, 2, and 3 GPUs, respectively, compared to GHOSTZ.

6.3.3 Evaluation of Search Sensitivity

The search results of GHOSTZ-GPU may be different from those of GHOSTZ because of the difference in calculation order for cells in S, I, and D in gapped extension. To evaluate the search sensitivity of GHOSTZ-GPU, we ran GHOSTZ-GPU and GHOSTZ using their default parameters. In this evaluation, we used only 10,000 short DNA reads from soil microbiome metagenomic sequences (SRR407548) as queries to compare with the results of SSEARCH. Search sensitivity was evaluated the same way as in Section 4.3. We used the single-value search sensitivity calculated as the ra-

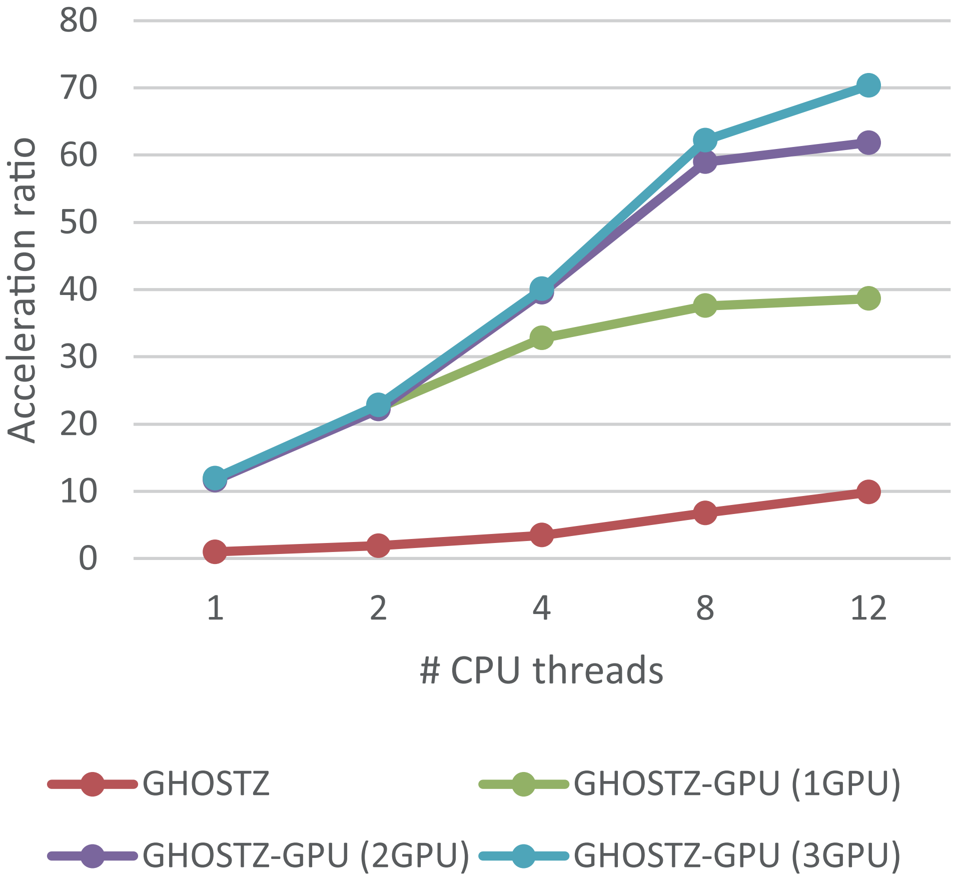


Figure 6.6: Computation times with multithreading of CPU and multi GPUs.

Table 6.2: Search sensitivity of different search methods for SRR407548 sequence alignments against the KEGG GENES database.

	Search sensitivity
GHOSTZ	0.86
GHOSTZ-GPU	0.86

tio of correctly searched queries to all queries with the E-values $< 1.0 \times E^{-5}$. The results for SRR407548 are shown in Table 6.2. The search sensitivity for GHOSTZ-GPU was approximately equal to that of GHOSTZ. In addition, we compared the results of GHOSTZ-GPU with those of GHOSTZ. We used 10,000 short reads from soil microbiome metagenomic sequences (SRR407548) as queries and compared subject sequences with the highest score in GHOSTZ-GPU with those in GHOSTZ. As a result, the difference between the results of GHOSTZ-GPU and GHOSTZ is only one query. Therefore, we believe that GHOSTZ-GPU has search sensitivity sufficient for most metagenomic applications.

6.3.4 Comparison of the Proposed Search Method with Other Methods

To further evaluate GHOSTZ-GPU, we compared its computation time with that of RAPSearch (version 2.12) and GHOSTZ using metagenomic DNA sequences (SRR407548, SRS011098, and ERR315856) and the KEGG GENES database. The RAPSearch program was executed with the default command line options.

The computation times of the tested methods for SRR407548, SRS011098, and ERR315856 are shown in Table 6.3. Among the software tested with 12 threads, GHOSTZ-GPU showed the fastest search speed: with 3 GPUs, it demonstrated approximately 5.1–7.1 and 22.3–39.0 times faster processing compared to GHOSTZ and RAPSearch, respectively.

6.3.5 Evaluation of Optimizations

To further evaluate GHOSTZ-GPU, we assessed optimization, asynchronous execution on the CPU and GPU, addition of threads for database loading, group memory access, and load balancing of gapped extension using 1,000,000 randomly selected short DNA reads from soil microbiome metagenomic sequences (SRR407548) and the KEGG

Table 6.3: Computation times for all datasets. We used GHOSTZ-GPU with 3 GPUs; each tool was used with 12 threads.

	SRR407548 (sec.)	SRS011098 (sec.)	ERR315856 (sec.)
GHOSTZ-GPU (3 GPUs)	561.6	423.8	584.6
GHOSTZ	3995.1	2140.3	3409.5
RAPSearch	21903.6	9719.3	13076.4

Table 6.4: Computation times for SRR40754 reads. We performed GHOSTZ-GPU without and with optimizations. The increase in processing speed is shown as the ratio of the time used for GHOSTZ-GPU with optimization to the time used for GHOSTZ-GPU with previous optimization and GHOSTZ with the thread for database loading.

	Computation time (sec.)	Acceleration ratio of each optimization	Cumulative acceleration ratio
GHOSTZ	3995.08	1.0	1.0
+ GPU	1006.0	4.0	4.0
+ Asynchronous execution	838.8	1.2	4.8
+ Loading database thread	617.8	1.4	6.5
+ Group memory access	571.7	1.1	7.0
+ Load balancing	561.6	1.0	7.1

GENES database. GHOSTZ was run with 12 threads and GHOSTZ-GPU with 12 threads and 3 GPUs. GHOSTZ used the thread for database loading. GHOSTZ-GPU was run without and with each optimization. Acceleration expressed as the ratio of the time used by GHOSTZ-GPU with optimization to the time used by GHOSTZ with the thread for database loading is shown in Table 6.4. Each optimization accelerated GHOSTZ-GPU. The asynchronous execution on the CPU and GPU and additional thread for database loading provided the most significant increase in speed, indicating that these optimizations are important for accelerating protein sequence homology search with the GPU.

6.4 Discussion

In this work, we mapped distance calculation, ungapped extension, and gapped extension to the GPU. However, performing these steps on the GPUs resulted in the remaining steps becoming new bottlenecks. For CPU calculation in GHOSTZ-GPU, the most time-consuming step was seed search. However, this step overlapped with distance calculation and ungapped extension on the GPU; therefore, the computation time for seed search was hidden in that for distance calculation and ungapped extension on the GPUs. However, in GHOSTZ-GPU, file I/O accounts for a significant portion of the computing time, suggesting that for the best performance, GHOSTZ-GPU should process large query data simultaneously. For concurrent processing of large query data, GHOSTZ-GPU needed larger memory. Thus, when we used 1,000,000 randomly selected short DNA reads from soil microbiome metagenomic sequences (SRR407548) and the KEGG GENES database, GHOSTZ-GPU and GHOSTZ required approximately 43 GB and 41 GB, respectively, for protein sequence homology search. However, current computing systems with multi-GPUs usually have relatively large memory. For example, the node in TSUBAME 2.5 has at least 54 GB. Therefore, GHOSTZ-GPU can be run in common multi-GPU environments.

6.5 Summary

We developed GPU-version of GHOSTZ. Distance calculation, ungapped extension, and gapped extension are the bottlenecks in GHOSTZ. We mapped these processes to the GPU, and optimized memory access in GPU calculation. GHOSTZ-GPU keeps sufficient search sensitivity for practical analyses and is 5.1–7.1 times faster than GHOSTZ. Given that sequencing technology continues to improve and sequence data for metagenomic analysis are increasing, GHOSTZ-GPU could be useful for circumventing these bottlenecks.

Chapter 7

A Large-scale Protein Sequence Homology Search on Massively Parallel Computing System

7.1 Introduction

To perform protein sequence homology searches with large amounts of data such as outputs from large-scale metagenomic projects, an efficient execution in supercomputing environments is critical. In recent years, the field of high performance computing has been rapidly evolving and we can use powerful supercomputers such as the K computer at the RIKEN Advanced Institute for Computational Science and TSUBAME 2.5 at Tokyo Institute of Technology. Thus, full utilization of large-scale computing resources makes it possible to comprehensively analyze large metagenomic datasets.

In metagenomics, query sequence data consist of many DNA reads independently processed for protein sequence homology search, which can be done in parallel. Ideally, parallel search reduces computation time, which is inversely proportional to the number of computational units. Darling *et al.* have developed mpiBLAST [8], which is a parallel implementation of NCBI BLAST with Message Passing Interface (MPI). mpiBLAST performs parallel query searches using multiple processes in distributed memory system with multiple CPU cores to reduce search time. However, mpiBLAST performance is insufficient to execute large-scale analyses, because its protein sequence homology search algorithm is much slower than most modern algorithms. Therefore, a faster protein sequence homology search tool operating on massive parallel computing systems is required. Here, we developed systems to effectively perform GHOSTM and

GHOSTX on supercomputers.

7.2 A Large-scale Protein Sequence Homology Search by Using GHOSTM

We developed a large-scale system based on GHOSTM, which allows the analysis of large metagenomic datasets obtained by NGS in real time by utilizing computational resources of TSUBAME 2.0. Using this system, we could process metagenomic information obtained from a single NGS run in a few hours.

For parallel performing of GHOSTM, the system uses Parallel Distributed Shell (PDSH) [2], an efficient multithreaded remote shell client that simultaneously executes commands on multiple remote nodes. However, file I/O processes, including database copying and search result writing, caused contention in TSUBAME 2.0, when many computation nodes were used. Thus, we employed a sophisticated file transfer process, when the data were simultaneously copied from a local disk of one node to the others in a binary-tree manner.

7.2.1 Results

We performed large-scale metagenomic analysis using our system on TSUBAME 2.0. Strong scaling was measured to evaluate scalability limitation by parallelization. In a strong scaling setup, the number of total query sequences was fixed; therefore, it evaluated how fast different methods could process the same data amount. The overview of the computing environment is as follows. TSUBAME 2.0 consists of 1,408 thin compute nodes, each of which has two Intel Xeon Processors X5670 (2.93 GHz, 6 cores) and 54 GB of main memory and is connected with full bisection bandwidth fat-tree network. Each compute node has three NVIDIA Tesla C2050 GPUs. We used data sampled from polluted soils and sequenced using NGS. Original metagenomic data had 224,000,000 75-bp DNA reads. After excluding low-quality data, the dataset comprised 71,000,000 DNA reads, which were used to perform protein sequence homology searches with NCBI nr program (4.2 GB; released April 2011). We evaluated the effectiveness of system performance with GHOSTM on the GPUs. The results show that with GHOSTM as a protein sequence homology search program, the system could process about 60,000,000 reads per hour with 2,520 GPUs (840 computing nodes) (Fig. 7.1). However, the speed with 2,520 GPUs increased only by 20% compared to that with

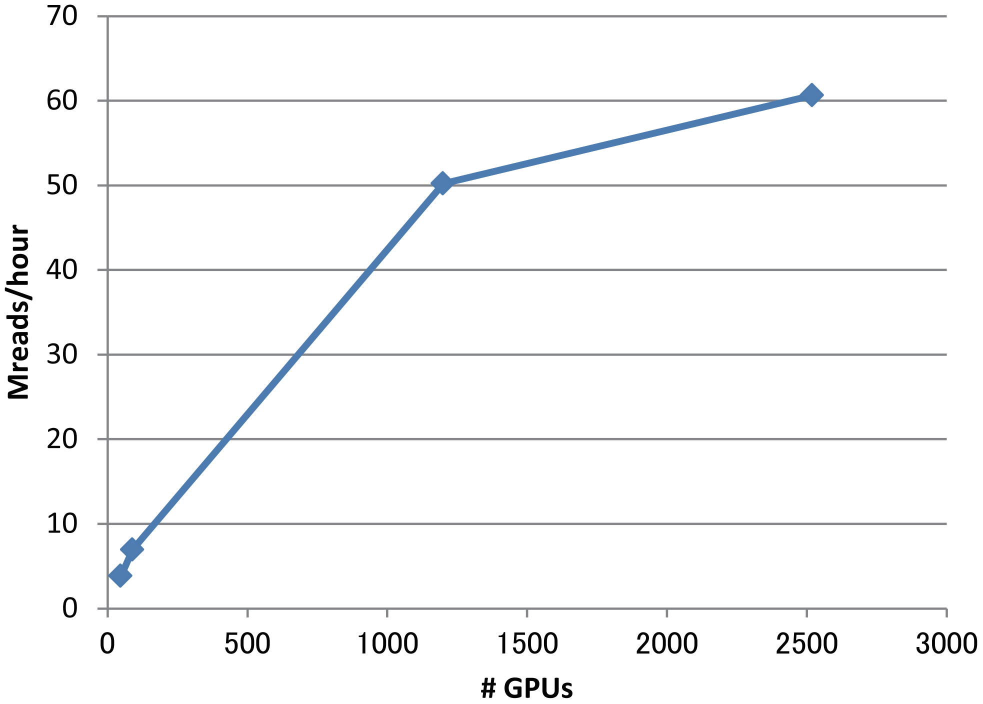


Figure 7.1: Speedup of the GHOSTM-based system for the number of GPUs.

1,260 GPUs, because the dataset was too small for 2,560 GPUs and might have caused load balancing failure. Therefore, linear acceleration may be achieved even with more than 2,520 GPUs by processing considerably larger amounts of metagenomic data.

7.3 A Large-scale Protein Sequence Homology Search by Using GHOSTX

We also developed a GHOSTX-based protein sequence homology search tool for massively parallel computing system, GHOST-MP, which adopted hierarchical parallelization with master-worker model and allowed performing efficient parallel search. To test the applicability of GHOST-MP to large-scale metagenomic analyses, we measured search speed and scalability of GHOST-MP on two massively parallel computing systems, TSUBAME 2.5 and the K computer. GHOST-MP demonstrated faster protein sequence homology search than the state-of-the-art method, and enabled us to perform large-scale metagenomic analysis in a short period of time.

7.3.1 Hierarchical Parallelization of Protein Sequence Homology Search with Data Parallelism

GHOST-MP adopts two-level hierarchical parallelization. Protein sequence homology search is parallelized by MPI on an inter-node level and by OpenMP on an intra-node level. Compared with parallelization by only MPI, hierarchical parallelization has two advantages. First, it significantly reduces memory usage by worker processes, because it allows sharing common data such as database sequences and indexes on an intra-node level. The size of database sequences and indexes (an index points to the corresponding position in concatenated database sequence, and database index size is the product of concatenated database sequence length) is often too large for memory size in massively parallel environments. For example, when KEGG GENES (3.5 GB; released July 2012) is used as a protein sequence database, the total size of database sequences and its suffix array is approximately 20 GB. If we use MPI both in inter- and intra-node parallelization, each process has to store the same database individually even within the same computing node. The nodes of current massively parallel computing systems rarely contain enough memory to store multiple copies of a database and its indexes. To reduce memory usage, it is possible to perform database partitioning and spread each partition over multiple nodes as an alternative approach, which is, however, inefficient in terms of search speed for two reasons. First, splitting database requires an additional merge step to integrate search results of the same query sequence obtained by different nodes to select the most similar hits. Second, searching for alignment candidates with split database takes longer CPU time than with undivided database, since the computation time of searching for alignment candidates with a suffix array is proportional to the logarithm of database size. Hierarchical parallelization can also lead to scalable parallel search. Since master-worker communication is implemented by MPI point-to-point communication, parallel search with smaller MPI reduces the number of master-worker communications compared to non-hierarchical parallelization (parallelization only with MPI). The details of two-level hierarchical parallelization with GHOST-MP are as follows. On the inter-node level, GHOST-MP adopts a master-worker model with communications implemented by MPI. The master process assigns a task to each worker process and then receives workers' reports; an idle worker process would be assigned a new task if available. Each task consists of a file containing query sequences and is accessed by worker processes through distributed file system. Each worker process executes protein sequence homology search, reports to the master process, and then receives the next task. It is required to split query

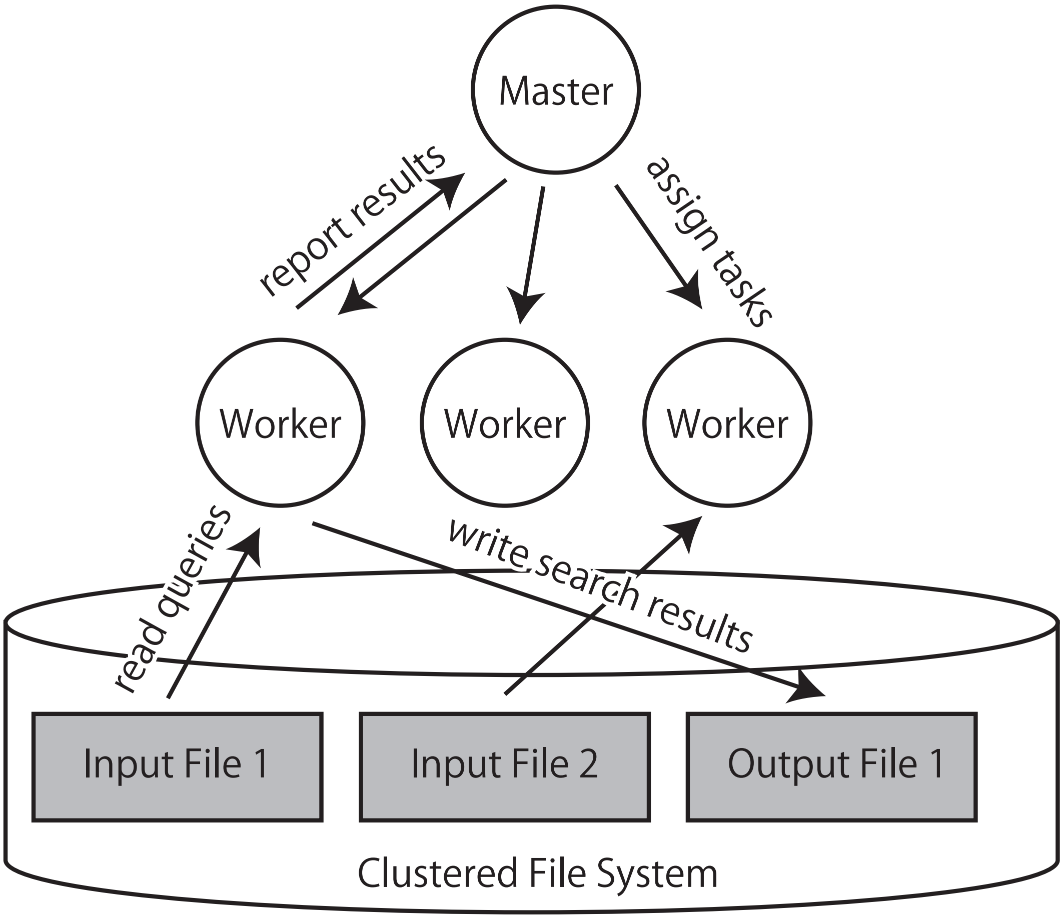


Figure 7.2: Schematic view of task distribution and file I/O of GHOST-MP.

sequence files in preprocessing, if the number of files does not exceed that of worker processes, because a sufficient number of tasks are needed for good parallel efficiency. Fig. 7.2 shows a schematic view of such task distribution and file I/O. On the intranode level, protein sequence homology searches are parallelized with OpenMP. Query sequences in a file are divided into more fine-grained tasks, which are arranged into a queue, and each OpenMP thread sequentially dequeues each task by releasing the lock.

7.3.2 Results

Evaluation of search speed and scalability of GHOST-MP on massively parallel computing systems

The search speed of GHOST-MP was measured on two systems, TSUBAME 2.5 and the K computer. Weak scaling and strong scaling were also measured to evaluate scalability limitations, since speedups achieved by parallelization are limited by sequential steps. In a weak scaling setup, the number of query sequences per core was fixed as the number of cores was increased; therefore, it evaluated how a big problem could be solved efficiently. In a strong scaling setup, the number of total query sequences was fixed; therefore, it evaluated how fast different methods could process the same data amount. We compared GHOST-MP with mpiBLAST (version 1.6.0) on TSUBAME 2.5, and also assessed GHOST-MP performance on the K computer as a larger system. We did not evaluate mpiBLAST performance on the K computer because of bus error, which could be caused by unaligned memory access, since the K computer processor does not allow such accesses. The overview of the two computing environments is as follows. TSUBAME 2.5 at Tokyo Institute of Technology is a supercomputer consisting of 1,408 thin compute nodes, each of which has two Intel Xeon Processors X5670 (2.93 GHz, 6 cores) and 54 GB of memory, and is connected with full bisection bandwidth fat-tree network. Each compute node has three NVIDIA Tesla K20X GPUs, which we did not use. The K computer at the RIKEN Advanced Institute for Computational Science is a supercomputer consisting of 82,944 compute nodes, each of which has a SPARC64 VIIIfx processor (2.0 GHz, 8 cores) and 16 GB of memory, and is connected to 6-dimensional mesh/torus network. We used up to 1,536 CPU cores (128 nodes) and 49,152 CPU cores (6,144 nodes) to measure the scalability of GHOST-MP on TSUBAME 2.5 and the K computer, respectively. To evaluate search speed and scalability, we used tongue dorsum metagenomic sequencing data (accession number, SRS078182) downloaded from HMP-DACC as queries. This metagenomic database consisted of approximately 147,000,000 reads and its file was approximately 36 GB; the longest read was 95 constituting 71.4% of the whole reads. For evaluation on TSUBAME 2.5, we used only 1,280,000 and 80,000 query sequences for GHOST-MP and mpiBLAST, respectively, because of limitation of computational resources. The KEGG GENES database (3.5 GB; released July 2012) was used as reference sequences. Both GHOST-MP and mpiBLAST performed protein sequence homology search with PAM30 score matrix, gap opening penalty of -9, and gap extension penalty of -1. Fig. 7.3 shows the search speed and scalability of GHOST-MP and mpiBLAST on TSUBAME 2.5. GHOST-MP was approximately 89 times faster than mpiBLAST with 1,536 CPU cores, because of an efficient alignment candidate search algorithm. In addition, GHOST-MP demonstrated almost linear scalability, as was the case with mpiBLAST regardless of its search speed, indicating that serial sections of GHOST-MP such as I/O and scheduling take only a small fraction of computation time compared to a parallelizable protein sequence homology search section. We further evaluated the scalability of GHOST-MP on the K computer to analyze its performance on a larger parallel computing system. It is generally more difficult to scale well on larger systems, because the master process has to communicate with more workers. However, GHOST-MP scaled well up to over 10,000 CPU cores in both systems (Fig. 7.4). GHOST-MP took 1.73 hours to process the whole data with 24,576 cores. This search speed suggests that 100 samples can be processed within 8 days providing large-scale metagenomic analysis with sensitive protein sequence homology search. However, the search speed decreased with 24,576 and 49,152 cores in weak and strong scaling, respectively, compared to the ideal. The decrease of search speed in weak scaling indicates that larger data cannot be processed efficiently, and such a decrease in strong scaling indicates that no further acceleration with the increase in computational resources is available. The reason for this performance drop seems to be a large number of point-to-point communications from the workers to the master. To make parallel search more scalable for large-scale analysis, the reduction in these communications is required. Introducing multiple masters or sub-masters on the MPI level or collective communication instead of point-to-point communication may address this problem.

7.4 Summary

For the analysis of metagenomic data obtained by NGS in real time, we developed a large-scale computing system with GHOSTM, which enabled us to utilize large computational resources provided by TSUBAME 2.0. We used GHOSTM to analyze metagenomic data in this system, and show that the system could process about 60 million reads per hour with 2,520 GPUs (840 computing nodes). We also developed GHOST-MP, a GHOSTX-based massively parallel protein sequence homology search tool, which performed parallel protein sequence homology search using a two-level hierarchical model. Combination of sophisticated database indexes and massively parallel processing allowed fast protein sequence homology search and large-scale metagenomic analysis. We confirmed the applicability of GHOST-MP to the current large-scale metagenomic analysis of NGS data by evaluating its search speed using an actual large metagenomic database. GHOST-MP performed an approximately 89 times faster search than mpiBLAST and achieved an almost linear speedup with the increase of CPU cores on TSUBAME 2.5. GHOST-MP also scaled well up to over 10,000 CPU cores on the K computer. These systems enable us to perform large-scale metagenomic analysis on a massively parallel computing system.

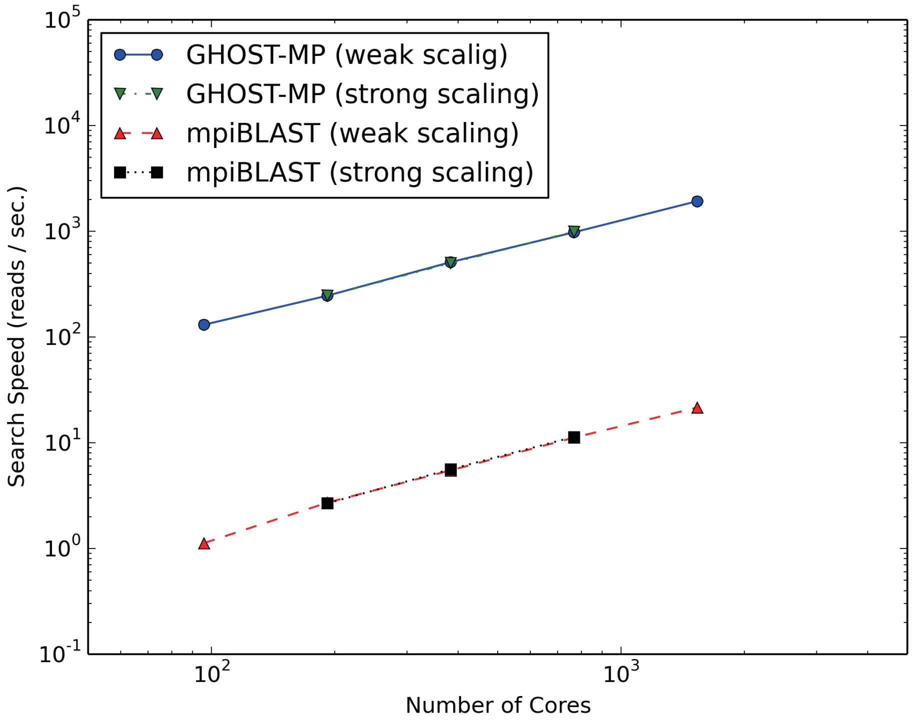


Figure 7.3: Scalability of GHOST-MP and mpiBLAST on TSUBAME 2.5.

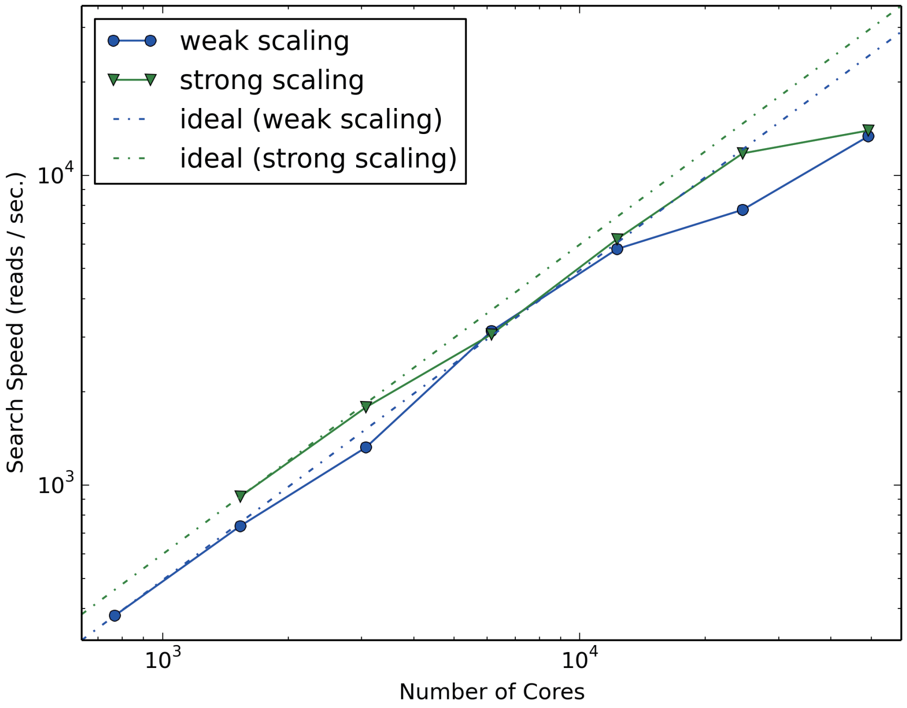


Figure 7.4: Scalability of GHOST-MP on K computer.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

In this thesis, we developed protein sequence homology search algorithms that can be used for metagenomic analysis of constantly accumulating sequence data. In addition, we increased the speed of our protein sequence homology search algorithms by using GPU calculation and adapted the tools to modern supercomputing environments. Based on the results of Chapter 4 and 6, GHOSTZ-GPU on 12 CPU cores and 3 GPUs is estimated to achieve a 20,000-fold increase in processing speed of BLASTX on 1 CPU core. The results of GHOST-MP on 8 and 128 nodes in TSUBAME 2.5 described in Chapter 7 show that week scaling of GHOST-MP is 0.92. Therefore, if we use 60 nodes on TSUBAME 2.5, GHSOTZ-GPU is estimated to achieve approximately 1,000,000-fold increase in processing speed compared to BLASTX on 1 CPU core. Moreover, if we use all nodes on TSUBAME 2.5 (1,408 nodes), GHSOTZ-GPU is estimated to achieve approximately 26,000,000-fold increase in processing speed compared to BLAST on 1 CPU core. Based on these estimations, we could perform metagenomic analysis of all data produced by the latest NGS in real time. Below, we describe practical significance of this work.

8.1.1 Contributions

1. In Chapter 3, we proposed a protein sequence homology search algorithm that identified similarities between query and database sequences based on the suffix arrays of these sequences and designated it as GHOSTX. This method uses a seed search method relying on a similarity-based optimal length. We showed that GHOSTX could provide approximately 131–165-fold protein sequence homology

search acceleration compared to BLASTX at similar levels of sensitivity.

2. In Chapter 4, we proposed a protein sequence homology search method based on database subsequence clustering, and designated it as GHOSTZ. This method clusters similar subsequences in a database to reduce the number of alignment candidates based on triangle inequality. This database subsequence clustering approach provided an approximately two-fold increase in speed without a significant decrease in search sensitivity. When we measured the performance with metagenomic data, GHOSTZ was approximately 213–285 times faster than BLASTX.

3. In Chapter 5, we developed a protein sequence homology search algorithm suitable for GPU calculations. We implemented it as a GPU system, and designated as GHOSTM. GHOSTM first searches for positions of sequence alignment candidates in a database using a hash table, and then calculates the scores of local alignments around the candidate positions before calculating similarity. The system with 1 GPU and 4 GPUs performed calculations approximately 130 and 407 times, respectively, faster than BLASTX with 1 CPU core. The system with 1 GPU and 4 GPUs also showed higher search sensitivity and calculation speed, which was approximately 4 and 15 times higher than that of BLAT with 1 CPU core.

4. In Chapter 6, we developed a GPU-version of GHOSTZ. Several calculation steps such as distance calculation, ungapped extension, and gapped extension are the bottlenecks in GHOSTZ. We mapped these processes to GPU and designated the algorithm as GHOSTZ-GPU. We optimized memory access in GPU calculation of GHOSTZ-GPU. In addition, in GHOSTZ-GPU, CPU and GPU calculations were overlapped to improve utilization efficiency of the CPU and GPU. When we used the approach with metagenomic data, GHOSTZ-GPU with 12 CPU cores and 3 GPUs was approximately 5.1–7.1 times faster than GHOSTZ with 12 CPU cores.

5. In Chapter 7, we developed a large-scale system for analyzing large amounts of metagenomic sequence data obtained by NGS in real time. We used GHOSTM to perform such an analysis by utilizing computational resources on TSUBAME 2.0. The system could process about 60 million reads per hour with 2,520 GPUs (840 computing nodes). In addition, we developed a parallel protein sequence homology search method to be used on massively parallel computing systems. This method provided fast protein sequence homology search with database indexes

and hierarchical parallel search as well as large-scale metagenomic analysis; its high parallel efficiency and search speed were confirmed on two massively parallel computing systems, TSUBAME 2.5 and the K computer. The method scaled well up to over 10,000 CPU cores and was approximately 89 times faster than mpiBLAST.

8.2 Future Work

DNA sequence technology is constantly improving and sequencing costs are gradually reducing; therefore, many organizations can benefit from this situation. Thus, hospitals can make use of DNA sequencing technology for diagnostic and prognostic purposes, and some companies can apply it to monitor environmental changes. To make DNA sequence data available to these organizations, a smaller size DNA sequencer is being developed. Wide application of such instrumentation would popularize the use of DNA sequence information related to the environment such as soil and ocean, and human body, for constant monitoring and control. However, a comprehensive picture of the interaction between microbial communities, humans, and environment can be obtained by analyzing large metagenomic data. Therefore, technologies to store and evaluate large amounts of sequence data will be required in the future.

Appendix A

Comparison of Search Sensitivity with Score Based on E-value

A.1 Comparison of Search Sensitivity with Score Based on E-value

To compare search sensitivity, we used the single-value search sensitivity calculated as the ratio of correctly searched queries to all queries with the E-values $< 1.0 \times E^{-5}$ in Section 4.3.3. We consider the importance of each hit with the E-values $< 1.0 \times E^{-5}$ as equal in sensitivity. However, the hits with low E-values are more important than others; therefore, we also compared search sensitivities with the scores based on the E-value.

Let e be the E-value. The score based on the E-value s is defined as follows.

$$s(e) = \begin{cases} -\log_{10} e & (e < 1.0 \times E^{-5}) \\ 0 & (otherwise) \end{cases} \tag{A.1}$$

We used the sum of s in the correct results to evaluate search sensitivity. A search result was considered correct when a subject sequence with the highest score in SSEARCH was the same as a subject sequence obtained by each tool the same way as in Section 4.3.3. We compared the sum of s for GHOSTZ, GHOSTX, RAPSearch, BLAT, BLASTX, and SSEARCH using the same dataset as in Section 4.3.3. Table A.1 shows the sum of s for each program.

Table A.1: Total score based on the E-value of SRR407548, SRS011098 and ERR315856. The ratio of total score of each tool to SSEARCH is in brackets.

	SRR407548	SRS011098	ERR315856
GHOSTZ	42070.2 (0.91)	35385.8 (0.99)	43877.9 (0.97)
GHOSTX	41486.7 (0.89)	35307.2 (0.98)	42354.3 (0.97)
RAPSearch	42085.6 (0.91)	35250.2 (0.98)	42613.6 (0.97)
BLAT	28574.0 (0.62)	34194.5 (0.95)	38193.4 (0.87)
BLASTX	44297.9 (0.95)	35065.1 (0.98)	42460.1 (0.97)
SSEARCH	46389.2 (1.00)	35913.0 (1.00)	43877.9 (1.00)

When we compared the sums of s, sensitivity rank was almost equal to Table 4.2. Therefore, we considered it sufficient for the comparison of search sensitivity using only the sensitivity calculated as the ratio of correctly searched queries.

References

[1] DNA Sequencing Costs. http://www.genome.gov/sequencingcosts/. Accessed: December 11, 2014.

[2] pdsh - Parallel Distributed Shell. https://code.google.com/p/pdsh/. Accessed: December 11, 2014.

[3] A. V. Aho and M. J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Communications of the ACM*, 18(6):333–340, 1975.

[4] S. F. Altschul *et al.* Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[5] S. F. Altschul *et al.* Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Research*, 25(17):3389–3402, 1997.

[6] S. F. Altschul *et al.* Protein database searches using compositionally adjusted substitution matrices. *FEBS Journal*, 272(20):5101–5109, 2005.

[7] NCBI Resource Coordinators. Database resources of the National Center for Biotechnology Information. *Nucleic Acids Research*, 42(D1):D7–D17, 2014.

[8] A. E. Darling *et al.* The Design, Implementation, and Evaluation of mpiBLAST. In *Proceedings of 4th International Conference on Linux Clusters: The HPC Revolution 2003 in conjunction with ClusterWorld Conference & Expo*, 2003.

[9] G. Das *et al.* Episode Matching. In *Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching*, pages 12–27, 1997.

[10] M. O. Dayhoff *et al.* A model of evolutionary change in proteins. *Atlas of Protein Sequence and Structure*, 5(suppl 3):345–351, 1978.

[11] R. Delourme *et al.* High-density SNP-based genetic map development and linkage disequilibrium assessment in Brassica napus L. *BMC Genomics*, 14(1):120–138, 2013.

[12] R. D. Finn *et al.* The Pfam protein families database. *Nucleic Acids Research*, 38(D1):D211–D222, 2010.

[13] L. Fontanesi *et al.* A genome wide association study for backfat thickness in Italian Large White pigs highlights new regions affecting fat deposition including neuronal genes. *BMC Genomics*, 13(1):583–592, 2012.

[14] L. Fu *et al.* CD-HIT: accelerated for clustering the next-generation sequencing data. *Bioinformatics*, 28(23):3150–3152, 2012.

[15] M. Ghodsi and M. Pop. Inexact Local Alignment Search over Suffix Arrays. In *Proceedings of the 2009 IEEE International Conference on Bioinformatics and Biomedicine*, pages 83–87, 2009.

[16] J. A. Gilbert *et al.* Meeting report: the terabase metagenomics workshop and the vision of an Earth microbiome project. *Standards in Genomic Sciences*, 3(3):243–248, 2010.

[17] O. Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162(3):705–708, 1982.

[18] S. Henikoff and J. G. Henikoff. Amino acid substitution matrices from protein blocks. *Proceedings of the National Academy of Sciences*, 89(22):10915–10919, 1992.

[19] M. Kanehisa *et al.* Data, information, knowledge and principle: back to metabolism in KEGG. *Nucleic Acids Research*, 42(D1):D199–D205, 2014.

[20] M. Kanehisa and S. Goto. KEGG: Kyoto Encyclopedia of Genes and Genomes. *Nucleic Acids Research*, 28(1):27–30, 2000.

[21] R. M. Karp and M. O. Rabin. Efficient Randomized Pattern-matching Algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[22] W. J. Kent. BLAT–The BLAST-Like Alignment Tool. *Genome Research*, 12(4):656–664, 2002.

[23] K. Kurokawa *et al.* Comparative Metagenomics Revealed Commonly Enriched Gene Sets in Human Gut Microbiomes. *DNA Research*, 14(4):169–181, 2007.

[24] B. Langmead *et al.* Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(3):R25, 2009.

[25] B. Langmead and S. L. Salzberg. Fast gapped-read alignment with Bowtie 2. *Nature Methods*, 9(4):357–359, 2012.

[26] V. Levenshtein. Binary codes capable of correcting spurious insertions and deletions of ones. *Problems of Information Transmission*, 1:8–17, 1965.

[27] H. Li and R. Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[28] H. Li and R. Durbin. Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinformatics*, 26(5):589–595, 2010.

[29] W. Li and A. Godzik. Cd-hit: a fast program for clustering and comparing large sets of protein or nucleotide sequences. *Bioinformatics*, 22(13):1658–1659, 2006.

[30] W. Liu *et al.* CUDA-BLASTP: Accelerating BLASTP on CUDA-Enabled Graphics Hardware. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 8(6):1678–1684, 2011.

[31] Y. Liu *et al.* CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units. *BMC Research Notes*, 2(1):73–83, 2009.

[32] Y. Liu *et al.* CUDASW++2.0: enhanced Smith-Waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions. *BMC Research Notes*, 3(1):93–105, 2010.

[33] Y. Liu *et al.* CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions. *BMC Bioinformatics*, 14(1):117–127, 2013.

[34] T. Luczak and W. Szpankowski. A Suboptimal Lossy Data Compression Based On Approximate Pattern Matching. *IEEE Transactions on Information Theory*, 43:1439–1451, 1996.

[35] B. Ma *et al.* PatternHunter: faster and more sensitive homology search. *Bioinformatics*, 18(3):440–445, 2002.

[36] U. Manber and G. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[37] F. Melo and M. A. Marti-Renom. Accuracy of sequence alignment and fold assessment using reduced amino acid alphabets. *Proteins*, 63(4):986–995, 2006.

[38] L. R. Murphy *et al.* Simplified amino acid alphabets for protein fold recognition and implications for folding. *Protein Engineering*, 13(3):149–152, 2000.

[39] G. Navarro *et al.* Indexing Methods for Approximate String Matching. *IEEE Data Engineering Bulletin*, 24(4):19–27, 2001.

[40] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.

[41] W. R. Pearson. Searching protein sequence libraries: Comparison of the sensitivity and selectivity of the Smith-Waterman and FASTA algorithms. *Genomics*, 11(3):635–650, 1991.

[42] Z. Peng *et al.* Comprehensive analysis of RNA-Seq data reveals extensive RNA editing in a human transcriptome. *Nature Biotechnology*, 30(3):253–260, 2012.

[43] J. Qin *et al.* A human gut microbial gene catalogue established by metagenomic sequencing. *Nature*, 464(7285):59–65, 2010.

[44] P. H. Sellers. The theory and computation of evolutionary distances: Pattern recognition. *Journal of Algorithms*, 1(4):359–373, 1980.

[45] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[46] S. Suzuki *et al.* An Ultra-Fast Computing Pipeline for Metagenome Analysis with Next-Generation DNA Sequencers. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 1549–1550, 2012.

[47] S. Suzuki *et al.* GHOSTM: A GPU-Accelerated Homology Search Tool for Metagenomics. *PLoS ONE*, 7(5):e36060, 2012.

[48] S. Suzuki *et al.* GHOSTX: An Improved Sequence Homology Search Algorithm Using a Query Suffix Array and a Database Suffix Array. *PLoS ONE*, 9(8):e103833, 2014.

[49] S. Suzuki *et al.* Faster sequence homology searches by clustering subsequences. *Bioinformatics*, in press.

[50] R. L. Tatusov *et al.* A Genomic Perspective on Protein Families. *Science*, 278(5338):631–637, 1997.

[51] R. L. Tatusov *et al.* The COG database: an updated version includes eukaryotes. *BMC Bioinformatics*, 4(1):41–55, 2003.

[52] The Human Microbiome Project Consortium. Structure, function and diversity of the healthy human microbiome. *Nature*, 486(7402):207–14, 2012.

[53] P. J. Turnbaugh *et al.* An obesity-associated gut microbiome with increased capacity for energy harvest. *Nature*, 444(7122):1027–1031, 2006.

[54] P. D. Vouzis and Nikolaos V. Sahinidis. GPU-BLAST: using graphics processors to accelerate protein sequence alignment. *Bioinformatics*, 27(2):182–188, 2011.

[55] J. P. Walters *et al.* Evaluating the use of GPUs in liver image segmentation and HMMER database searches. In *IPDPS'09: Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12, 2009.

[56] C. Whitelaw *et al.* Using the Web for Language Independent Spellchecking and Autocorrection. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing*, pages 890–899, 2009.

[57] J. C. Wootton and S. Federhen. Statistics of local complexity in amino acid sequences and sequence databases. *Computers & Chemistry*, 17(2):149–163, 1993.

[58] Y. Ye *et al.* RAPSearch: a fast protein similarity search tool for short reads. *BMC Bioinformatics*, 12(1):159–169, 2011.

[59] Y. Zhao *et al.* RAPSearch2: a fast and memory-efficient protein similarity search tool for next-generation sequencing data. *Bioinformatics*, 28(1):125–126, 2012.

List of Publications

Reviewed Journal Papers

1. **Shuji Suzuki**, Takashi Ishida, Ken Kurokawa and Yutaka Akiyama. GHOSTM: A GPU-Accelerated Homology Search Tool for Metagenomics. *PLOS ONE*, 7(5):e36060, 2012. (Chapter 5)

2. **Shuji Suzuki**, Masanori Kakuta, Takashi Ishida and Yutaka Akiyama. GHOSTX: An Improved Sequence Homology Search Algorithm Using a Query Suffix Array and a Database Suffix Array. *PLOS ONE*, 9(8):e103833, 2014. (Chapter 3)

3. **Shuji Suzuki**, Masanori Kakuta, Takashi Ishida and Yutaka Akiyama. Faster sequence homology searches by clustering subsequences. *Bioinformatics.* (in press) (Chapter 4)

Reviewed International Conference Proceedings

1. **Shuji Suzuki**, Takashi Ishida and Yutaka Akiyama. An Ultra-Fast Computing Pipeline for Metagenome Analysis with Next-Generation DNA Sequencers. *In Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, 1549–1550, Salt Lake City, 2012. (Chapter 7)

Honors and Awards

1. **Shuji Suzuki**, JSPS Research Fellow (DC1) (2012–2015).

2. **Shuji Suzuki**, Grant-in-Aid for JSPS Fellows (24·8766) (2011–2014).

3. **Shuji Suzuki**, 2010 IPSJ SIGBIO Best Student Presentation Award (2011).

4. **Shuji Suzuki**, 1st prize for GPU Challenge 2010.

5. Hiroya Nagao and **Shuji Suzuki**, IBM Ground Prize (1st prize) for "Cloud Computing Competition" in Interop Tokyo 2010.