<서울대학교 자연과학대학 생명과학부 학사졸업논문>

Petasearch: Fast, approximate comparison of huge sequence datasets

(페타탐색: 방대한 서열 데이터셋에 대해 빠른 유사성 검색)

이 논문을 이학사 학위논문으로 제출함

2022년 8월

구분: 실험

논문심사 대상자 소속: 자연과학대학 생명과학부

학번: 2017-17232

성명: Minghang Li (인)

논문지도교수: Martin Steinegger (인)

연구윤리 준수 서약서

본인 (Minghang Li)은 서울대학교 연구자로 연구를 진행함에 있어 다음 사항을 준수할 것을 서약합니다.

1. 서울대학교 연구윤리 관련 규정 및 지침과 국가 법령 및 정부 지침 그리고 일반적으로 학계에서 인정되는 연구윤리 기준을 준수하여 서울대학교 내에서 연구를 수행할 때 위조 변조 표절 등 학문적 진실성을 훼손하는 연구부정행위 또는 연구부적절행위를 하지 않겠습니다.

2. 인간, 동물 등 연구대상에 대한 국내외 윤리 기준을 준수하도록 하겠습니다.

3. 연구 진행 중 이해상충이 발생할 경우 이를 공개하도록 하겠습니다.

4. 정부 및 본교 지침에 따라 연구노트를 작성하며, 연구 데이터에 대한 관리를 철저히 하도록 하겠습니다.

2022년 6월 4일

서약자 소속(학과명): 자연과학대학 생명과학부

이름: Minghang Li (서명)

Abstract

The Sequence Read Archive currently holds over 60 petabases, representing a treasure trove for medicine and biotechnology. Bloom-filter and sketching based approaches were proposed to accelerate searches, however they offer only limited sensitivity. We developed Petasearch to enable fast and sensitive searching through huge protein databases. Its algorithm contains three stages: (1). We pre-process the database sequences to extract k-mers, sort and store them in a highly compressed k-mer index. (2). We extract query k-mers, add similar k-mers and find matches between query and database k-mers. To maximize throughput, we exploit the caching and prefetch infrastructure of modern CPUs, advanced Linux IO techniques, and the enormous read bandwidth of NVMe-SSDs. (3). We compute SIMD-accelerated banded Smith-Waterman alignments between sequences of high-scoring k-mer matches. With such design, Petasearch is proved to have great efficiency: it is up to 190 times faster than state-of-the-art algorithms on a 9.3TB benchmark. At much accelerated speeds, Petasearch matches state-of-the-art algorithms on sensitivity down to sequence identities of 60%. On a SCOP25 benchmark we showed that Petasearch' s profile search detects sequence homology down to 40% sequence identity. We also showed that Petasearch can be applied in finding novel Cas family proteins and discovering new RNA-dependent RNA polymerase (RdRP) homologs. In conclusion, Petasearch is a tool with huge potential. It will enable fast querying of current and upcoming databases and bring bioinformatic researches to a larger scale.

Keywords: Sequence analysis, Sequence search, Protein databases, Proteins, Protein profiles, Large-scale annotation

Student ID: 2017-17232

Contents

1. Introduction

1.1 Sequence Databases

Next generation sequencing (NGS) technologies have revolutionized the way we collect and analyze biological data. Thanks to NGS, the cost of sequencing has dropped drastically and continued to decrease with more new technologies developed. Accompanying this change is the explosive growth of the amount of sequencing data and the size of sequence databases. The Sequence Read Archive (SRA) is one of the most popular and widely used sequence databases that store both private and public sequence reads and provide access in various foramts including the commonly used `FASTQ` file format. Its size has grown exponentially since 2008 and currently reached more than 60 petabytes large. The growth in size of Sequence Read Archive is visualized in Figure 1.1. Among all 16,700,872 SRA samples shown in SRA sequence browser, merely searching with they keyword "metagenome" already gives 2,176,719 results. Such a huge amount of metagenomic data is a valuable resource for many research projects including drug discovery, environment conservation, and finding candidates for bioengineering.

1.2 State-of-the-art Algorithms for Sequence Searches

Due to the extensive size of the Sequence Read Archive and other sequence databases, it is a challenge to provide much needed algorithms to search against the vast amount of data. The classical `BLAST` and `BLASTX` are sensitive but slow [1], thus not feasible for an all-encompassing search through petabyte-scale databases or other high-throughput scenarios. Although there were attempts to use cloud computing to speed up the search process [2], the base performance of the algorithms were not improved. Algorithms such as `MMseqs2` [3], `DIAMOND` [4] and `BIGSI` [5] are some of the most promising ones that strike a good balance between sensitivity and speed. However, they are still not capable of handling terabyte-scale data.

1.3 Motivation and Contribution of the Thesis

In conclusion, the search of homologs in large sequence databases requires a fast yet sensitive enough algorithm specially designed for petabyte-scale analysis. The state-of-the-art searching algorithms failed to satisfy this need. To tackle these problems and, we developed the `Petasearch`. Jonas H*ü*gel first proposed and impelmented the prototype of `Petasearch` in his Master's thesis in collaboration with Milot Mirdita. The thesis author

revised the design of the core data structures and database format of `Petasearch`, integrated new modules, and added the profile-search functionality, pushing the algorithm to a finishing state. The thesis author also impelmented new benchmarks, prepared metagnomic assembly for `Petasearch` to search on, and conducted two explorative analyses using `Petasearch`. The main contribution of this thesis is the major improvement of the `Petasearch` algorithm in speed, space consumption and sensitivity.

In Chapter 2, we will start with introducing the current overview of the `Petasearch` algorithm, and continue with describing the its further development and optimization. We will also describe the design of the benchmarks in Chapter 2. In Chapter 3, we will first show the improvements in efficiency and effectiveness of the forementioned optimizations. Afterwards, we will show a thorough comparison of the performance of the `Petasearch` algorithm with the state-of-the-art algorithms. In Chapter 4, we will discuss the potential application of `Petasearch` and show two examples of its usage.

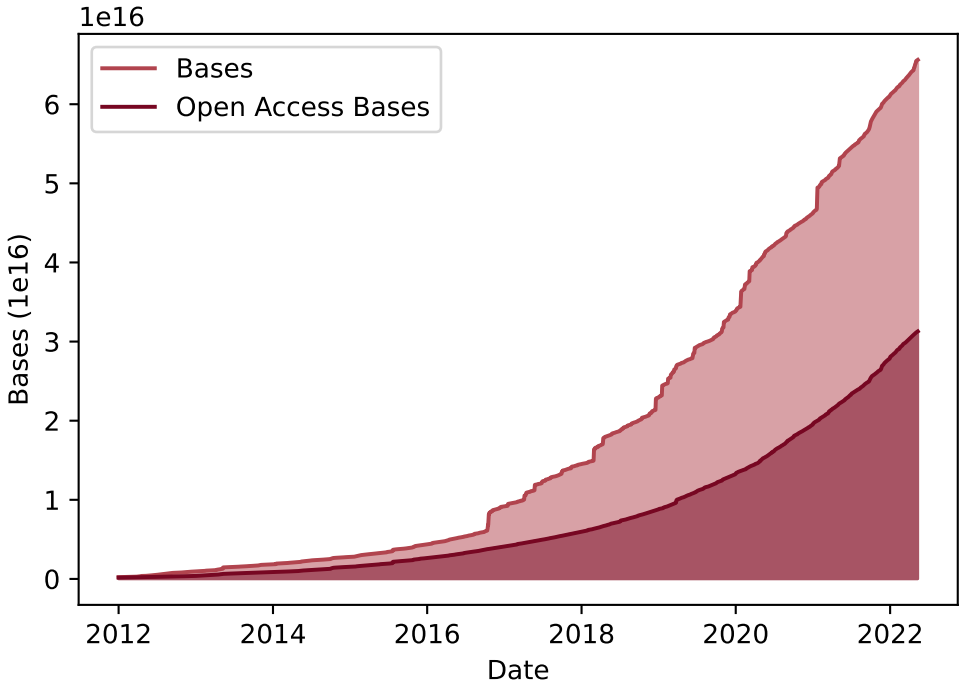


Figure 1.1: The exponential growth of the Sequence Read Archive over the decade (from 2012 to 2022). The total amount of sequence data (unit in bases) and publicly available data are visualized in pink and dark red respectively.

2. Materials & Methods

2.1 Overview of Current `Petasearch` Algorithm

An overview of the current `Petasearch` algorithm is visualized in Figure 2.1. The algorithm contains the following three stages:

1. **Pre-processing phase**: In this phase, we first convert the input `FASTA/Q` file into a compressed `Petasearch` sequence database format (described in Section 2.2.2), and then we use the k-mer extraction tool to create the target tables. For each database, two tables were created. The first one contains the unique k-mers, the second one contains the corresponding sequence ID in the database. Instead of storing the k-mers as strings, we first converted the k-mers their numeric representations, and then sorted them. To further decrease the required amount of storage, we implemented a diff-index to store only the difference between k-mers, which is compressed using a creative bit-squeezing technique (described in Section 2.2.1), resulting in mostly 2 bytes of memory instead of 8 bytes per k-mer.

2. **Prefiltering phase**: In the second step, we first create a query table for the input query sequence or profile database, extract all the k-mers inside and generate similar k-mers using either BLOSUM [6] matrices or the profile substitution matrix. We then perform a linear read and comparison between the query and target k-mers to find all the double k-mer matches. Such linear reading of the database is friendly to the caching and prefetch infrastructure of modern CPUs and modern NVMe SSDs. To maximize the huge NVMe linear read throughput, we also exploit advanced Linux IO techniques (described in Section 2.3.1). After the comparison finishes, we sort the result and remove all entries where we have fewer than two hits for a target sequence from one query sequence to reduce random hits.

3. **Alignment phase**: In the alignment phase, we read in the prefiltered results and retrieve the corresponding target sequence for the hits. In order to reduce unnecessary computation, we calculate the diagonal where the k-mer match occurred, which is the position of the k-mer in the target-sequence minus that in the query sequence. We skip the calculation if we do not find two diagonals within a distance of 4. Afterwards, we calculate the gapless alignment score for the all the diagonals and use the highest-scoring diagonal to compute SIMD-accelerated banded Smith-Waterman alignments.

The contributions of the author were detailed in Section 2.2, Section 2.3 and Section 2.4.

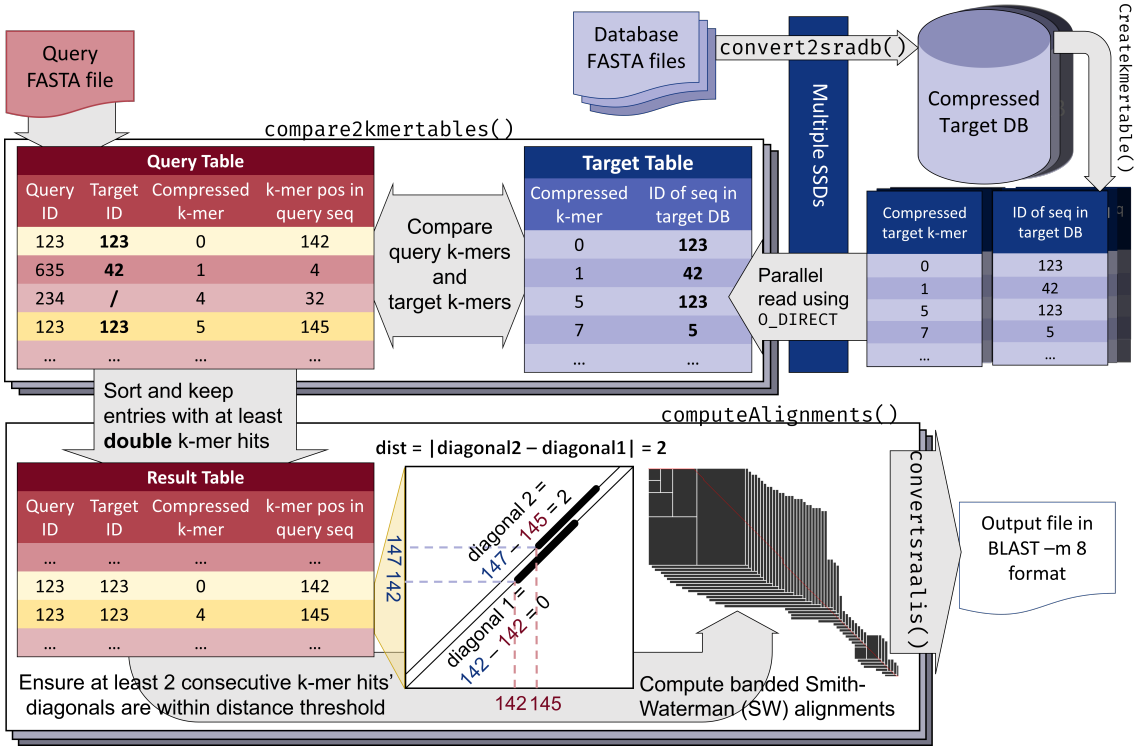


Figure 2.1: General overview of the Petasearch steps. The preparation step, which will utilize the `convert2sradb()` and `createkmertable()` functions, creates the target tables that store all unique k-mers for each database. The pre-filtering step, which is implemented in `compare2kmertables()` consists of a highly parallelized double index search that streams the target tables directly from different solid state devices (SSDs) to saturate their maximum possible read rates and avoid random memory accesses. Requiring duplicate k-mer matches for each query-target sequence pair reduces the possibility of random hits and increases the specificity of the hits. An alignment for a query-target sequence pair is computed if the distance between diagonals for at least one k-mer pair is less than four. This constraint removes sequence pairs where the positions of the matching k-mers are not close and therefore a good alignment is unlikely. The alignment phase is realized in function `computeAlignment()` and the alignment output is possible to be converted into popular output formats such as the `BLAST -m 8` format.

2.2 Space Optimization

The core idea of `Petasearch` is actually sacrificing space for fast computation. However, the resources are not limitless. Thus, we would like to keep the cost of space as low as possible while keeping the searching speed high. In this section, we will discuss the space optimization techniques utilized to improve `Petasearch` prototype.

2.2.1 Petasearch Data Structures Compression

As is described in Section 2.1, the values in diff-index created in the k-mer extraction step are usually able to fit inside a `short` integer. However, when the difference is larger than `USHRT_MAX`, in the prototypical implementation, we will store multiple `USHRT_MAX` as long as the difference is larger than `USHRT_MAX`. This will make any k-mer difference larger than $4 \times$ `USHRT_MAX` $= 262140$ require a larger space to store than the original `unsigned long` representation. This situation is not uncommon especially when k is large.

Also, in the prototypical implementation, the ID of the source sequence will also be stored multiple times in the ID table. This redundancy is both unnecessary and troublesome. It will increase the size of `Petasearch` data structures even more than the repeated `USHRT_MAX` since the IDs are stored as `unsigned int` (32-bit integers).

Figure 2.2 showed the space consumption of the diff-index created in the k-mer extraction step when $k = 11$. Without optimization, the diff-index (k-mer table) and its corresponding ID table will take up 17 GB of space for a merely 1GB-sized database.

To optimize the size of the diff-index, we devised the bit-squeezing technique to compress the difference between two adjacent k-mers: For any 64-bit k-mer difference, we continuously fetch 15 bits into a write buffer starting from the least significant bits. We stop the retrieval until we encounter a zero chunk (15 bits of zeros).

To enable the correct decoding of the diff-index during the next phase, the sign bit of the last element in th write is set to `1` to indicate the end of encoding. Afterwards, we write all the elements in the write buffer to the diff-index. An example encoding process for difference of value 2039432531946 is shown in Figure 2.3. For ID table, the optimization is simple: we store the ID of the source sequence only once instead of repeatedly.

Using the bit-squeezing method, it is possible to obtain a maximum of five chunks, making the final space consumption larger than the size of a `unsigned long` integer. However, such situation only happens when the difference is larger than `1UL << 59` =

576460752303423488, which is extremely rare.

While decoding the compressed diff-index in the process of double-index search, we will reverse the bit-squeezing process through repeatedly retrieving 15 bits from the diff-index table until we encounter the chunk with the sign bit set to `1`. The decoded difference value will be add to the current k-mer. Moreover, since we do not store redundant IDs, the ID pointer will not be incremented until the end of k-mer decoding. Algorithm 1 showed the simplified pseudocode for k-mer decoding.

Algorithm 1 Pseudocode for the k-mer decoding process

```
procedure DecodeKmer(currKmer, currTargetKmerPtr, currTargetIDPtr)
    currDiffIndex ← 0
    while *currTargetKmerPtr > 0 do          ▷ This means the sign bit is not 1.
        currDiffIndex ← Get15Bits(*currTargetKmerPtr)
        currDiffIndex ← currDiffIndex << 15
        Next(currTargetKmerPtr)
    end while
    currDiffIndex ← Get15Bits(*currTargetKmerPtr)
    currKmer ← currKmer + currDiffIndex
    Next(currTargetKmerPtr)
    Next(currTargetIDPtr)
     return currKmer
end procedure
```

2.2.2 Protein Sequence Compression

For terabyte-size databases, the sequences themselves are also space consuming. To further reduce the size of the databases, we developed the `ASCII`-sqeezing technique.

Protein sequences are represented by a limited subset of `ASCII` characters, which are encoded by a single byte. However, as is shown in Figure 2.4, we only need 5 bits to represent all the amino acids. Therefore, we can squeeze every three amino acids into one 16-bit `short`. Similar to the bit-squeezing technique described in Section 2.2.1, we also use the sign bit to indicated the end of the compressed protein sequence. Figure 2.5 showed an example compression process for glutathione (GSH). The `ASCII`-squeezing technique is expected to produce a sqeuence database about 85% of the original size.

2.3 Speed Optimization

Speed is the first and foremost concern of `Petasearch`. The speed of `Petasearch` prototype is already fast, but did not make it stand out too much from its competitors. In this section, we will introduce several techniques to further boost the speed of `Petasearch`.

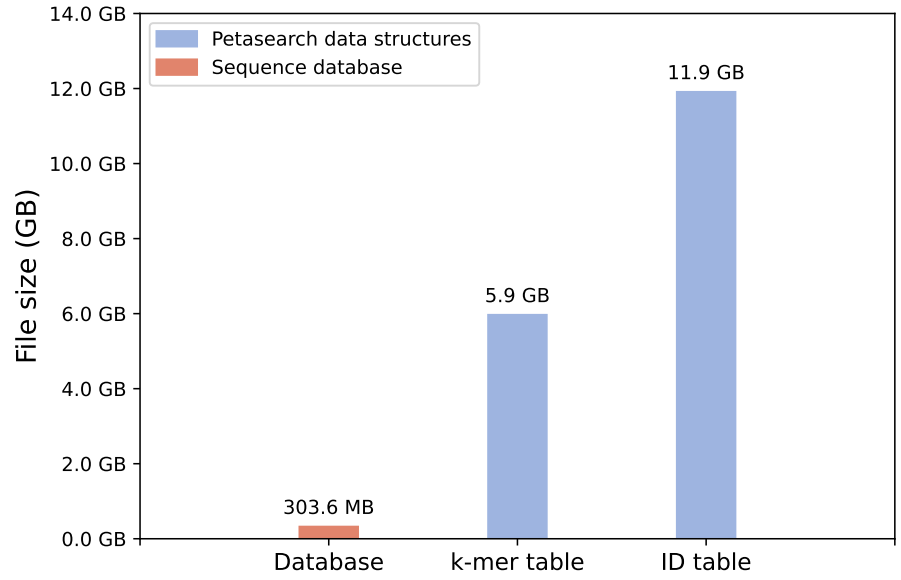


Figure 2.2: Visualizaiton of k-mer table and ID table sizes when k = 11. The database is the *UniProtKB/Swiss-Prot* database obtained through `mmseqs databases UniProtKB/Swiss-Prot swissprot tmp` command. Without optimization, the sizes of `petasearch` data structures are 6.46 times and 12.92 times larger than the sequence database.

2.3.1 IO Performance Optimization

In the prototypical `Petasearch` implementation, `mmap` was selected for reading `Petasearch` data structures stored on NVMe SSDs. However, `mmap` does not scale well with the increase in threads [7] and thus cannot saturate the full throughput of NVMe SSDs. To find the IO tool with the best performance, we conducted a benchmarking study on the performance of various IO tools using FIO benchmark software [8]. For synced IO tools, we benchmarked `pread` using different flags and `mmap`. For async IO tools, we benchmarked `libaio` and `posix_aio`.

The benchmarking results are visualized in Figure 2.6. It is clear that `libaio` performs the best. I is able to saturate the full 3.5 GB/s linear read bandwidth of NVMe SSDs. The other two tools, `posix_aio` and `pread` with `O_DIRECT` (`ioengine = psync` in FIO) have roughly the same performance, with bandwidth around 3.3 GB/s. Unfortunately, `mmap` has the worst performance, with only about 1.5 GB/s at maximum. The performance even fell to 0.25 GB/s when it scaled to 20 threads. Since `mmap` is a synced IO module, adopting another synced IO module will require almost no change in control logic. Considering both the performance and difficulty of refactoring, we chose to use `pread` with `O_DIRECT` in place of `mmap`.

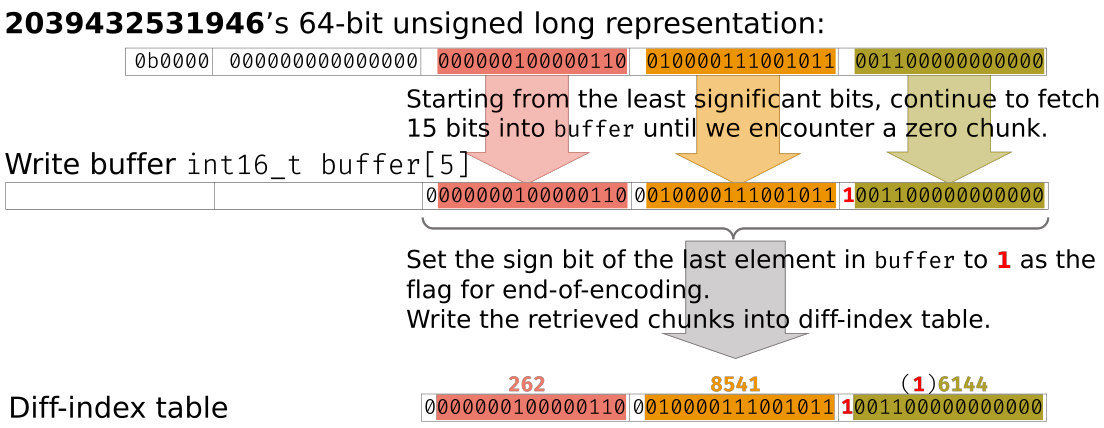


Figure 2.3: The example decoding process of difference index 2039432531946. We first retrieve 15-bit chunks starting from the least significant bits and store them into a write buffer in the reverse order until we encounter a zero chunk. For 2039432531946, its highest non-zero bit is 39, which means that we need three 16-bit `short` to store it.

The implementation of `pread` with `O_DIRECT` is rather simple: we simply create a read buffer according to the currently available memory size, open the k-mer diff-index file with `O_DIRECT` flag, and then read in parallel using `pread` continuously untill `EOF` (end of file).

2.3.2 Simplified Database Index

In `Petasearch` prototype, the sequence database format is the same as that of `MMseqs2`, which uses a complexed index structure for it having many functions. Such complexity is unnecessary for `Petasearch`, especially when the database is huge in size, making the overhead for a complicated index become non-trivial. Hence, we simplified the index, only preservingthe offset of the corresponding entry in the sequence database. The simple yet efficient chagne not only reduce the IO overhead, but also derease the space consumption.

2.3.3 Fast Third-Party Libraries

We integrated several fast thrid-party libraries to replace the slow implementaions in `Petaserch` prototype. The fast parallel in-place sorting algorithm IPS^4o [9] was integrated to replace the slow `std::sort`. The banded Smith-Waterman-Gotoh aligner `block-aligner` [10] was used to allow fast pairwise alignment in the third phase.

2.4 Sensitivity Improvement

The k-mer matching mechanism limits `Petasearch`'s ability of finding homologs for low sequence identity. To improve `Petasearch`'s performance at lower sequence identity, we made `Petasearch` able to perform profile search by allowing profile databases as inputs.

Profile search is using a "sequence profile" generated from multiple sequence alignment (MSA) results as the querying input [11]. The profile Hidden Markov Model (HMM) provides position-specific aminoaid insertion, deleletion and substitution penalties [11], which significantly increase the searching sensitivity. The most sensitive searching tools such as `HMMER` [12], [13], `HHblits` [14] and `HH-suite3` [11] all use the profile search mechanism. Thus, enabling `Petasearch` to perform profile search is expected to improve its sensitivity at low sequence identity.

2.5 Benchmarks

Since `Petasearch` has been much improved and new features were also added, we also devised updated benchmarks to evaluate both the its speed and sensitivity. We performed all the benchmarks on a server with AMD EPYC 7702P (128) @ 2.000GHz CPUs, 995 GB RAM, Debian GNU/Linux 11, twenty two Samsung 970 EVO Plus NVMe SSDs (2 TB) connected via PCI-Express.

2.5.1 Speed Benchmark

In the time benchmark we measured the time that `Petasearch`, fast `MMseqs2` search with default parameters and `DIAMOND` search with both query-index mode (fast) and double-index mode (default) to search a small query set of 2.9 MB against a 9.3 TB large target set. `Petasearch` was only conducted using 9-mers due to space limitation. We used the original 456 GB data of Soil Reference Catalog and Marine Eukaryotic Reference Catalog and duplicated several copies of them until we filled twenty NVMe disks fully with the sequence databases and `Petasearch` tables. As a result, each NVMe roughly contains seven to eight 70GB databases. We ran `Petasearch` algorithm ten times and report the average run-time to avoid deviations. The other two algorithms, due to their slowliness, were only run once to show the rough runtime magnitude. We only varied the `--exact-k-mer-matching` parameter to control the creation of similar k-mers for `Petasearch`, and the `--algo` parameter for `DIAMOND` to choose between only indexing query and double-indexing. All other parameters were set to default.

A	01000001	J	01001010	S	01010011
B	01000010	K	01001011	T	01010100
C	01000011	L	01001100	U	01010101
D	01000100	M	01001101	V	01010110
E	01000101	N	01001110	W	01010111
F	01000110	O	01001111	X	01011000
G	01000111	P	01010000	Y	01011001
H	01001000	Q	01010001	Z	01011010
I	01001001	R	01010010		

Figure 2.4: Part of the ASCII table, showing the bit representation of `A` to `Z` with the last 5 bits highlighted. It can be clearly seen that for `A` to `Z` in the English alphabet, we can represent them using only 5 bits instead of a whole byte (8 bits).

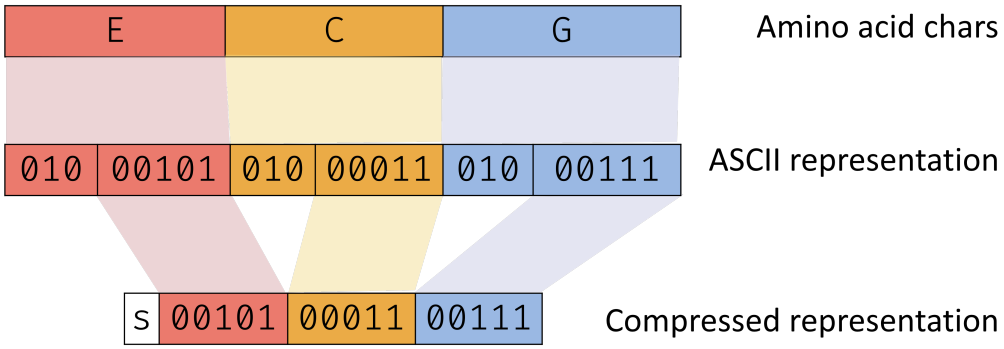


Figure 2.5: The example compression of short peptide glutathione (GSH). GSH consists of only three amino acids: glutamate (E), cysteine (C), and glycine (G). We simply fetch the least significant 5 bits of each amino acid `char` and store them into a single 16-bit `short`.

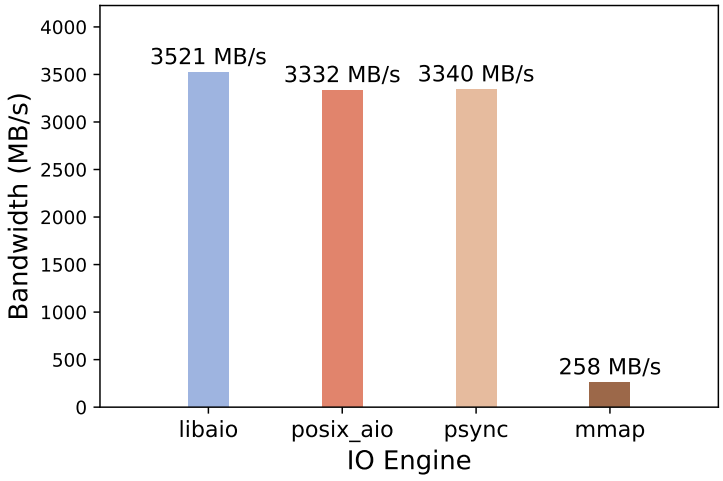


Figure 2.6: Benchmarking results of various IO tools using FIO benchmark software. The benchmark setting is imitating a parallel read from 20 NVMe SSDs. A total of 20 threads were created, each one responsible of reading a 50 GB file stored on one NVMe SSD. The bandwidth is the average reading bandwidth per SSD. The IO engine `psync` is equivalent to opening a file handle with `O_DIRECT` and reading from the handle using `pread`.

2.5.2 Sensitivity Benchmark

For the sensitivity benchmark we measured and compared the results of the `Petasearch` algorithm using both sequence search mode and profile search mode against `MMseqs2` sequence search (fast preset, `-s 5.7`), `MMseqs2` profile search (fast preset) and `DIAMOND` (fast and sensitive presets). The main goal of this benchmark is to show that `Petasearch` is able to compete with the other, more sensitive algorithms in the upper sequence identity buckets, and the newly added profile search is able to have better sensitivity in the lower sequence identity buckets.

We downloaded the *UniProt* database and randomly extracted one million entries and clustered them to 0.9, 0.8, 0.7, 0.6, 0.5 and 0.4 percent sequence identity using `MMseqs2`'s clustering workflow. For each cluster, we randomly selected 5% of the sequences as query database, and the remaining 95% as target database. For each sequence in every databases, we tagged the sequence with its domain information by using `MMseqs2` to search the "scop25 dbset" described by Hauser et. al. [15] against both query database and target database. The sequence without any SCOP domain was filtered out. For all other sequences in the query databases, we reversed the region that is not the SCOP25 domain; for target sequences, we shuffled those regions randomly. In this way, we can use the domain information to identify false positive hits. If the hits happen between two sequences with SCOP domain similarity lower than family level, we will consider it as false positive (FP). Otherwise, it will be considered as true positive (TP). For each algorithm, we only considered the best hit found for each query. Since all algorithms sort their hits by the E-value, we selected the hit with the highest E-value.

3. Results

3.1 Effect of Space Optimization

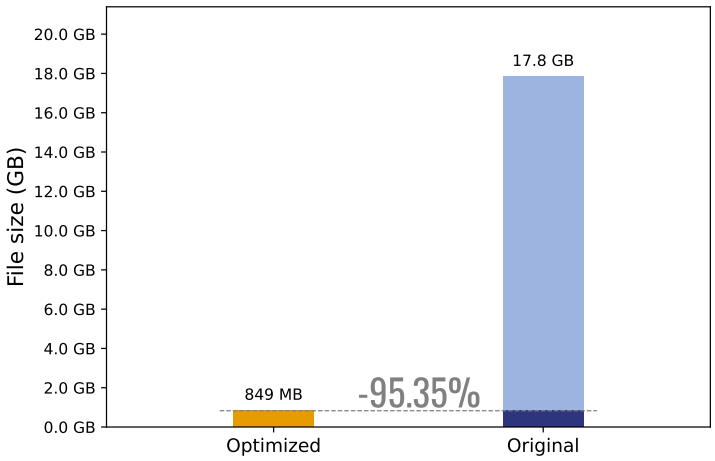
The effect of space optimization for k-mer table and ID table is visualized in the Figure 3.1. For the forementioned 900MB database in Section 2.2, Figure 2.2, when we generate `Petasearch` data structures for 11-mers, the total space usage is reduced from 17.8GB to merely 849MB (0.85GB), decreasing the space usage by 95.35% compared to the original `Petasearch` data structures. Viewing separately, the space usage of the k-mer table is reduced from 5.9GB to 405MB (−93.34%), and the space usage of the ID table is reduced from 11.9GB to merely 444MB (96.36%). The effect of ASCII-squeezing method and simplified database index combined is exhibited in Figure 3.2. The size of the sequence database was reduced by 28.27% for a 300MB-scale database. For gigabyte-sized database, the compression efficiency is even higher – a 82.1GB database got compressed to only 46.3GB thanks to ASCII-squeezing method and simplified index.

3.2 Speed Benchmark

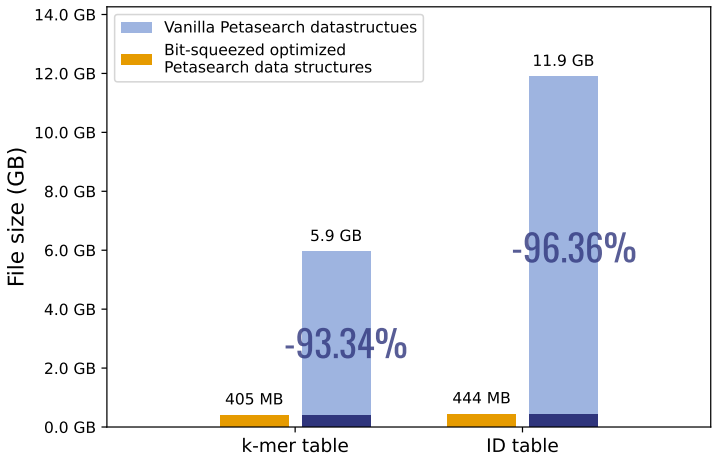
Because of space limitation, the detailed effect of each optimization was omitted. Instead we present the results of speed benchmark in Table 3.1 with all optimizations applied. The results were also visualized in visualized in Figure 3.3 to give the readers an intuitive impression of how fast the algorithm is. It is noteworthy that in the benchmark for `Petasearch` prototype, the searching time for a 450GB target set was already 15m49s without generating similar k-mers.

3.3 Sensitivity Benchmark

We visualized the sensitivity benchmark in **??**. It is confirmed that `Petasearch` worked as expected as it found a similar number of high sequence identity hits to other methods. The performance is comparable up to 60% sequence identity.

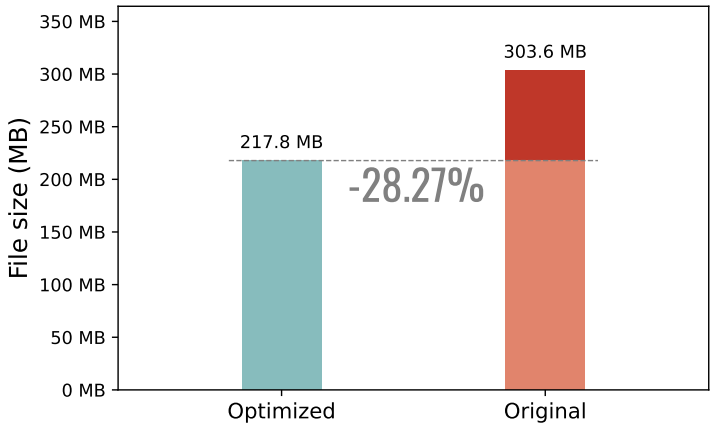


(a) Visualization of he total amount of reduction in space usage for the `Petasearch` data structures. The great reduction in space usage makes the previous implausible search for 11-mer index possible.

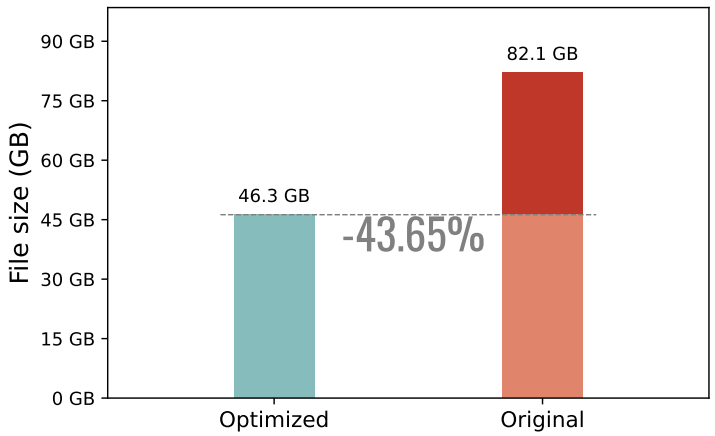


(b) Visualization of the space reduction for k-mer table and ID table separately. The carefully designed bit-squeezing technique and redundancy reducion for ID table are both highly effective.

Figure 3.1: Effect of space optimization for k-mer diff-index table and ID table using bit-squeezing technique at k = 11. Fig. 3.1a showed the total effect of bit-squeezing on both `Petasearch` data structures. Fig. 3.1b showed the effect of bit-squeezing on the k-mer diff-index table and the ID table separately.



(a) The amount of space reduction for a 303.6 MB database. The space saving is about 28.27% or one third of the size of the original database.



(b) The amount of space reduction for a 82.1 GB database. The space saving is even higher than smaller databases, reaching 43.65%.

Figure 3.2: Effect of `ASCII`-squeezing method and simplified database index. We also briefly test the effect of the compression efficiency with respect to the size of the database.

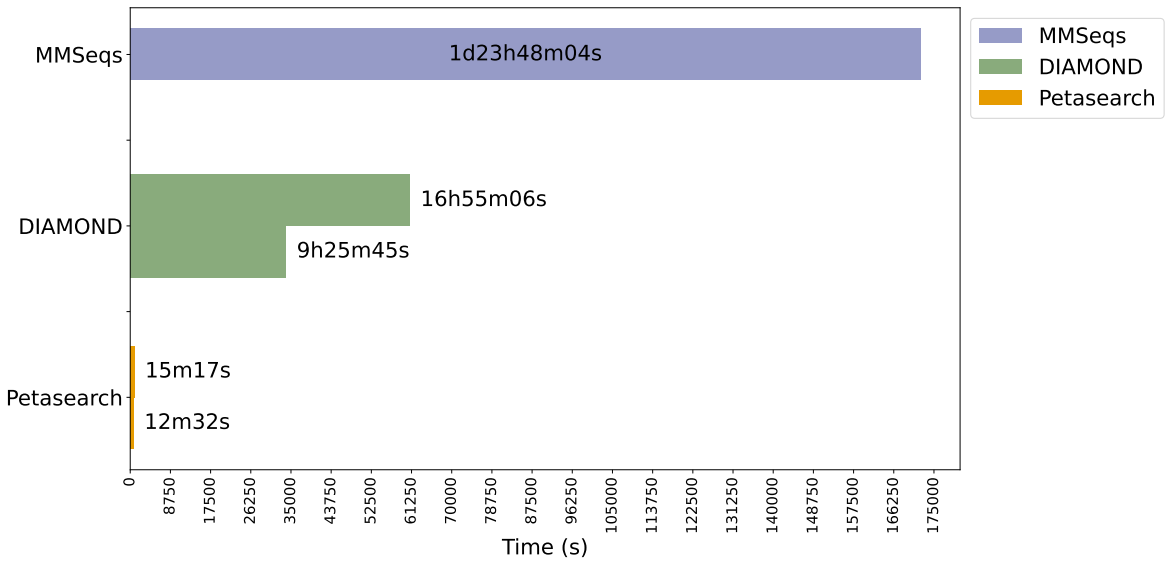


Figure 3.3: Visualization of the speed benchmark results. It is clearly shown that the optimization is very effective – the runtime of `Petasearch` is almost negligible compared to `MMseqs2` and `DIAMOND`.

Name	Time	Speed up	Special parameters	Notes
petasearch	12m 32s	-	`--exact-kmer-matching 1`	Do not generate similar k-mers for query database.
	15m 17s	-	`--exact-kmer-matching 0`	Generate similar k-mers for query database.
DIAMOND	9h 25m 45s	38x	`--min-score 40 --algo 1`	Apply the fast query-indexed algorithm. Did not include the `makedb` time
	16h 55m 06s	68x	`--min-score 40 --algo 0`	Use the default sensitive double-indexed algorithm. Did not include the `makedb` time
MMseqs2	47h 48m 04s	191x	Default params	Did not include the `createdb` time.

Table 3.1: Results of the time benchmark using `Petasearch`, `MMseqs2` and `DIAMOND`. We searched a 2MB query set against a 9.3TB target set (size of the `FASTA` file including the headers). We prepared a total of 146 target database chunks distributed across 21 NVMe SSDs, 11 of which have 6 chunks and 10 of which have 8 chunks. The size of the chunks are evenly distributed. For `Petasearch`, we repeated the benchmark 10 times to get an average time. For `MMseqs2` and `DIAMOD`, they were only run once since the time for a full round of search is too long. The parameters and the corresponding explanation of the parameters used in the benchmark were given in the fourth and fifth columns of the table.

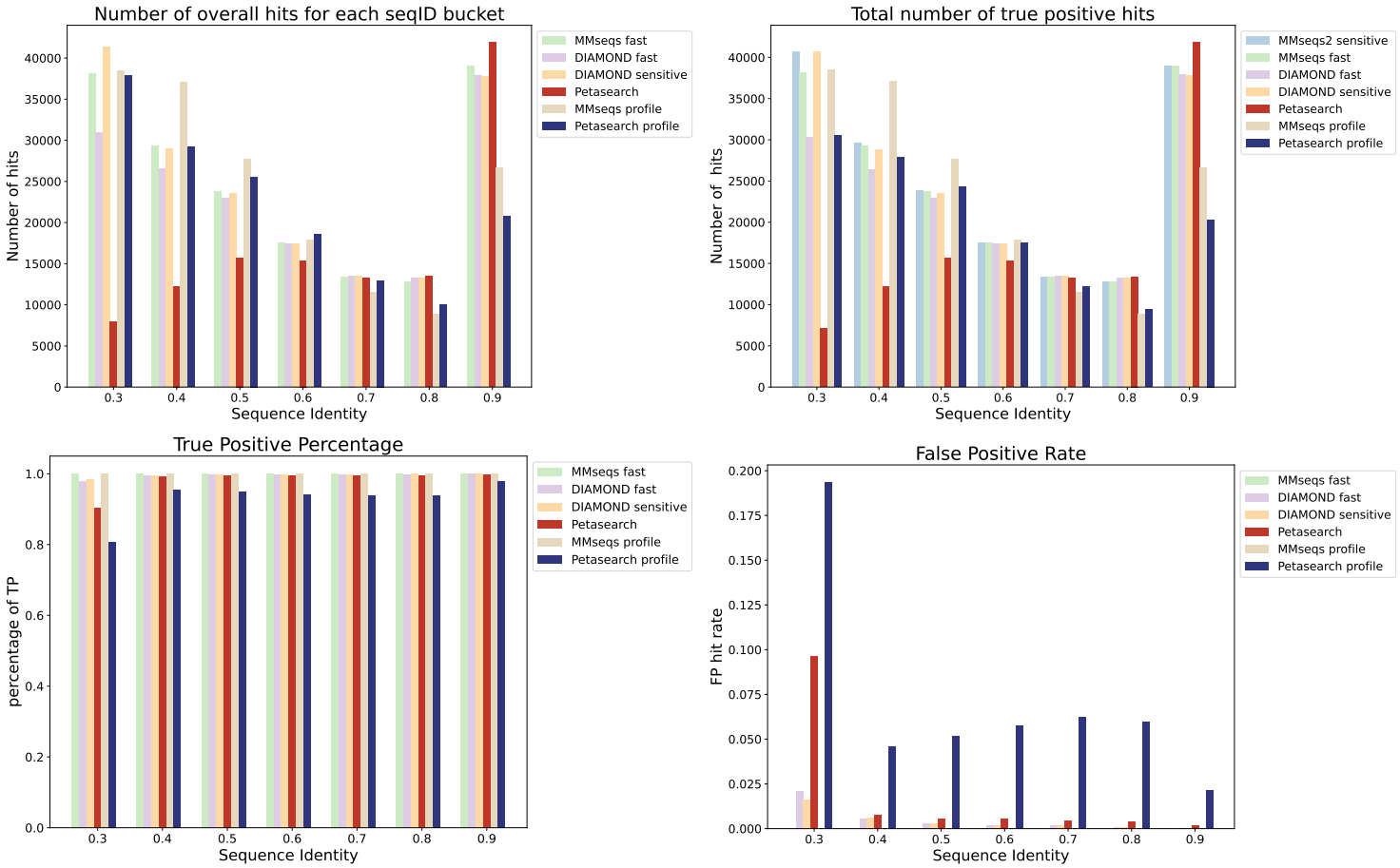


Figure 3.4: Results of the sensitivity benchmark.

4. Discussion

asdfas

asdfsaf

5. References

[1] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of molecular biology*, vol. 215, no. 3, pp. 403–410, 1990.

[2] K. Levi, M. Rynge, E. Abeysinghe, and R. A. Edwards, "Searching the sequence read archive using jetstream and wrangler," in *Proceedings of the practice and experience on advanced research computing*, 2018, pp. 1–7.

[3] M. Steinegger and J. Söding, "Mmseqs2 enables sensitive protein sequence searching for the analysis of massive data sets," *Nature biotechnology*, vol. 35, no. 11, pp. 1026–1028, 2017.

[4] B. Buchfink, C. Xie, and D. H. Huson, "Fast and sensitive protein alignment using diamond," *Nature methods*, vol. 12, no. 1, pp. 59–60, 2015.

[5] P. Bradley, H. C. Den Bakker, E. P. Rocha, G. McVean, and Z. Iqbal, "Ultra-fast search of all deposited bacterial and viral genomic data," *Nature biotechnology*, vol. 37, no. 2, pp. 152–159, 2019.

[6] S. Henikoff and J. G. Henikoff, "Amino acid substitution matrices from protein blocks," *Proceedings of the National Academy of Sciences*, vol. 89, no. 22, pp. 10 915–10 919, 1992.

[7] A. Papagiannis, G. Xanthakis, G. Saloustros, M. Marazakis, and A. Bilas, "Optimizing memory-mapped i/o for fast storage devices," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 813–827.

[8] J. Axboe, *Flexible I/O Tester*, 2022. [Online]. Available: `https://github.com/axboe/fio`.

[9] M. Axtmann, S. Witt, D. Ferizovic, and P. Sanders, "In-place parallel super scalar samplesort (ips 4 o)," *arXiv preprint arXiv:1705.02257*, 2017.

[10] D. Liu and M. Steinegger, "Block aligner: Fast and flexible pairwise sequence alignment with simd-accelerated adaptive blocks," *bioRxiv*, 2021.

[11] M. Steinegger, M. Meier, M. Mirdita, H. Vöhringer, S. J. Haunsberger, and J. Söding, "Hh-suite3 for fast remote homology detection and deep protein annotation," *BMC bioinformatics*, vol. 20, no. 1, pp. 1–15, 2019.

[12] S. R. Eddy, "A new generation of homology search tools based on probabilistic inference," in *Genome Informatics 2009: Genome Informatics Series Vol. 23*, World Scientific, 2009, pp. 205–211.

[13] S. R. Eddy, "Accelerated profile hmm searches," *PLoS computational biology*, vol. 7, no. 10, e1002195, 2011.

[14] M. Remmert, A. Biegert, A. Hauser, and J. Söding, "Hhblits: Lightning-fast iterative protein sequence searching by hmm-hmm alignment," *Nature methods*, vol. 9, no. 2, pp. 173–175, 2012.

[15] M. Hauser, M. Steinegger, and J. Söding, "MMseqs software suite for fast and deep clustering and searching of large protein sequence sets," *Bioinformatics*, vol. 32, no. 9, pp. 1323–1330, Jan. 2016.