BIOINFORMATICS ORIGINAL PAPER Vol. 24 no. 16 2008, pages 1757–1764
doi:10.1093/bioinformatics/btn322

Genome analysis

Database indexing for production MegaBLAST searches

Aleksandr Morgulis, George Coulouris, Yan Raytselis, Thomas L. Madden, Richa Agarwala and Alejandro A. Schäffer*

National Center for Biotechnology Information, National Institutes of Health, Department of Health and Human Services, Bldg. 38A, Room 6S608, 8600 Rockville Pike, Bethesda, MD 20894, USA

Received on March 6, 2008; revised and accepted on June 18, 2008

Advance Access publication June 21, 2008

Associate Editor: John Quackenbush

ABSTRACT

Motivation: The BLAST software package for sequence comparison speeds up homology search by preprocessing a query sequence into a lookup table. Numerous research studies have suggested that preprocessing the database instead would give better performance. However, production usage of sequence comparison methods that preprocess the database has been limited to programs such as BLAT and SSAHA that are designed to find matches when query and database subsequences are highly similar.

Results: We developed a new version of the MegaBLAST module of BLAST that does the initial phase of finding short seeds for matches by searching a database index. We also developed a program `makembindex` that preprocesses the database into a data structure for rapid seed searching. We show that the new 'indexed MegaBLAST' is faster than the 'non-indexed' version for most practical uses. We show that indexed MegaBLAST is faster than miBLAST, another implementation of BLAST nucleotide searching with a preprocessed database, for most of the 200 queries we tested. To deploy indexed MegaBLAST as part of NCBI's Web BLAST service, the storage of databases and the queueing mechanism were modified, so that some machines are now dedicated to serving queries for a specific database. The response time for such Web queries is now faster than it was when each computer handled queries for multiple databases.

Availability: The code for indexed MegaBLAST is part of the `blastn` program in the NCBI C++ toolkit. The preprocessor program `makembindex` is also in the toolkit. Indexed MegaBLAST has been used in production on NCBI's Web BLAST service to search one version of the human and mouse genomes since October 2007. The Linux command-line executables for `blastn` and `makembindex`, documentation, and some query sets used to carry out the tests described below are available in the directory:
ftp://ftp.ncbi.nlm.nih.gov/pub/agarwala/indexed_megablast

Contact: schaffer@helix.nih.gov

Supplementary information: Supplementary data are available at *Bioinformatics* online.

1 INTRODUCTION

BLAST (Altschul *et al.*, 1997) is a package of programs to match a query biological sequence against a database and identify database sequences that have statistically significant local alignments with a part of the query. One of the principal heuristics that makes BLAST fast is preprocessing the query to build a lookup table that is subsequently used to find short high-scoring ungapped alignments, herein called 'seeds'. Seeds can be extended to longer ungapped alignments and then to full gapped alignments in later phases of BLAST programs. The default length for seeds is 3 amino acids for proteins, 11 nucleotides for high-sensitivity nucleotide searching with the BLASTN module and 28 nucleotides for lower sensitivity searching with the MegaBLAST module (Zhang *et al.*, 2000). An important innovation introduced in BLAST version 2.0 was to require the existence of two nearby seeds to reduce the number of candidate matches that pass the seeding stage (Altschul *et al.*, 1997). For proteins, recent work shows that requiring a single contiguous seed of length 6 and more complex preprocessing gives better performance (Shiryev *et al.*, 2007).

Instead of finding seeds by searching a data structure derived from the *query*, one could instead find seeds by searching a data structure derived from the *database*. Numerous research studies, recently reviewed in Jiang *et al.* (2007), suggest that preprocessing the database and searching a database-derived data structure can yield much faster search times.

Two widely used software packages that do preprocess the database for biological sequence comparison are SSAHA (Ning *et al.*, 2001) and BLAT (Kent, 2002). However, both these packages are defined to find only near identical matches [e.g. for a comparison of BLAT and the TBLASTN module of BLAST see Gertz *et al.* (2006)]. The only software package we could find that:

(1) produces results similar to or identical to some module of BLAST,

(2) preprocesses the database, rather than the query, to build a data structure for the seed search phase,

(3) uses comparable amounts of memory for the data structure and the database (to exclude suffix-tree solutions) and

(4) has executables or source code currently and freely available on the web

is miBLAST (Kim *et al.*, 2005). miBLAST was designed for short queries of under 100 bases and we confirm that its relative performance does deteriorate for longer queries. Below, we refer to methods, such as miBLAST, that build a search structure from the database by the collective term 'database indexing'.

Since database indexing seems so promising, but no module of NCBI BLAST uses this paradigm, we set out to engineer

*To whom correspondence should be addressed.

a replacement for (at least) one module of NCBI BLAST and assess whether the claimed performance benefits could be achieved in production usage. The most commonly searched database at NCBI's BLAST Web service is `nr`, which is a comprehensive 'non-redundant' collection of sequences. Indexing of the `nr` database poses specific challenges as compared with other databases because `nr` is so large and is updated daily. Other popular databases for web BLAST searches are the human and mouse genomes, and to a lesser extent other vertebrate genomes.

Due in part to a redesign of NCBI's BLAST web page in 2007, single-genome databases for human and mouse have become increasingly selected by users who submit approximately 10 000 and 3000 queries per weekday for the human and mouse databases, respectively. The current default is that the genome-specific nucleotide databases are searched with the MegaBLAST (Zhang *et al.*, 2000) module, called from within the program `blastn` compiled from the NCBI C++ toolkit.

Therefore, we set out to develop a new version of MegaBLAST that could use a data structure derived from the genome-specific database to search for the initial seeds. We also needed to build a program, herein called `makembindex`, to construct the index structure. The basic goals for 'indexed MegaBLAST' were that it should find the same matching sequences and alignments as the version from which we started, and do so much faster for the majority of queries that arise in practice. Unlike miBLAST, performance of indexed MegaBLAST should degrade gracefully as the queries become longer or have an increasing number of matches.

Early in the project, we realized that a basic impediment to good performance would be queries that align well to DNA sequences present with minor variations many times in the genome. To address this difficulty, we developed the software package WindowMasker (Morgulis *et al.*, 2006a) to quickly mask frequently repeated sequences without using a library of repeats. We realized that most users rarely want to see such repetitive matches output from BLAST, so we modified MegaBLAST to be capable of searching a 'soft-masked' database. Soft-masked means that seeds cannot intersect a masked interval, but alignments can be extended into and through masked intervals. Soft masking a query has long been available in MegaBLAST.

Sections 2 and 3 and Supplementary Material describe how we engineered and tested indexed MegaBLAST to document the potential performance improvement. The acid test is how users of NCBI's web BLAST service would respond. Indexed MegaBLAST was deployed in October 2007 for the BLAST nucleotide search variants of querying the human and mouse genomes (separately) and announced in regular release notes. By default, WindowMasked versions of the genomes are used, but users can turn off masking. There have been zero user complaints and only one general inquiry suggesting that indexed MegaBLAST works well in production.

2 METHODS

In this section and Supplementary Material, we describe the data structures used to organize the database index and the 'seed search algorithm' used to find initial identical substrings in the query and the database. We also describe the testing strategy and how indexed MegaBLAST has been put into production usage. The seed search algorithm has been incorporated into a modified version of NCBI's MegaBLAST, as described subsequently, with few changes to the algorithmic code parts that do subsequent processing of the seeds.

2.1 Index structure

The MegaBLAST database index contains compressed sequence data along with locations of k-mers. Besides the length k, two other important parameters are the minimum seed length $w(\geq k)$ and the stride s used to move through the database. To guarantee that every exact match of length w between a query and a subject is found, we use the relationship $s=w-k+1$. Information about the k-mer ending at every s-th position, satisfying conditions specified below which assure we can find all seeds, is stored in the index. The current implementation defaults to $w=16$, $k=12$ and $s=5$.

A database index is composed of several files called index volumes, each corresponding to a contiguous range of sequences from the underlying database. An index volume file contains three sections: header, sequence data and offset data, as illustrated in Figure 1. The number of nucleotides in the database is denoted by n. In Supplementary Material, we derive an estimate for the number of bytes needed to store the index as:

$$2\times 4^{k+1}+\frac{n}{4}+\frac{4n}{\min\left(s,\frac{k-1}{2}\right)} \tag{1}$$

bytes. For example, for an unmasked database of size 1 GB and our default values $k=12, s=5$, the estimated size of the index would be ~1.175 GB.

2.1.1 Header The header section of an index volume primarily stores the range of sequences from the underlying database that are indexed by that volume. The header section also stores a few auxiliary values that are used for housekeeping purposes, such as index format version. The exact definition of the current version of the index format can be found in the external documentation accompanying the code.

2.1.2 Sequence data Sequence data are stored in compressed format with 2 bits per sequence base using NCBI2NA encoding. In that encoding, bases A, C, G and T are encoded as integers 0, 1, 2 and 3 correspondingly. Positions containing ambiguous characters are encoded as 0. The seed search algorithm, described in the next subsection, never touches ambiguous parts of the sequences. These parts serve merely as placeholders.

The seed search algorithm works with *logical sequences* of more or less uniform size. This makes mapping from k-mer locations to the corresponding sequences more efficient. Logical sequences may be formed by splitting long sequences or by combining several short sequences, as described subsequently. The mapping between the logical sequences and the actual database sequences is stored in the sequence data section of an index volume.

All logical sequences participating in a given index volume are assigned a consecutive integer *sequence id*. Both in indexed MegaBLAST and other variants of NCBI BLAST software, long database sequences are split into overlapping *chunks* to optimize processing. From now on, by *input sequence* we mean either a database sequence or a chunk produced by such splitting, which enables us to avoid cluttering the algorithm description with the details of handling long sequences.

Mapping of input sequences to logical sequences. Let ℓ be the chunk size defined in the BLAST code (current default is 5 Mbases), and let $B=\lceil\log_2(\ell)\rceil$. Because of the chunking, we may assume that all input sequences are of length at most ℓ. Consecutive input sequences are concatenated into logical sequences as long as their combined length does not exceed ℓ. Logical sequences are assigned consecutive integer *logical sequence ids* starting at 0.

The sequence data section of an index volume contains a table that maps each logical sequence id to the input sequences from which it is composed. Each element of the table contains four integers:

(1) sequence id of the first input sequence in the logical sequence;

(2) sequence id of the last input sequence in the logical sequence;

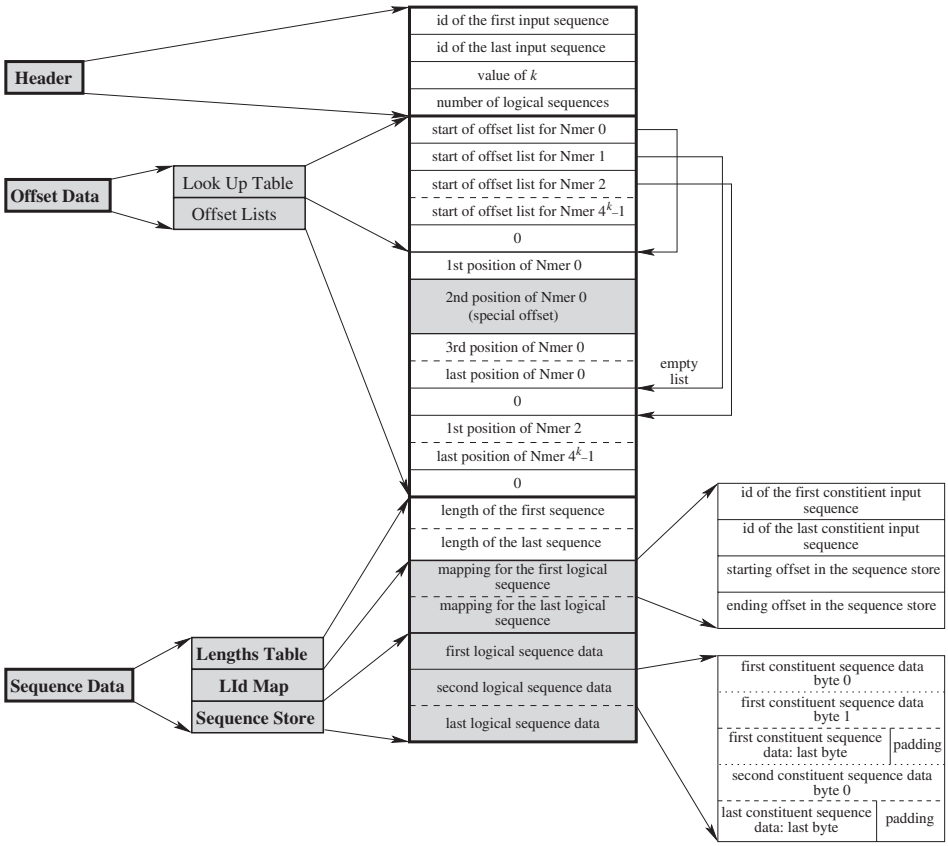

Fig. 1. Schematic of the data structure used for the database index.

(3) offset (in bytes) of the start of this logical sequence from the start of the sequence store and

(4) offset (in bytes) of the end of this logical sequence from the start of the sequence store.

Sequence store. The sequence store is the part of the sequence data section of an index volume that contains compressed sequence data. Logical sequences are stored there in order of increasing logical sequence id.

Logical sequence data contains concatenated encoded constituent input sequences. The encoding of input sequences using 2 bits per base is padded with extra 0 bits at the end, if needed, to ensure that they end on a byte boundary. The index volume stores the true lengths of all input sequences, so the seed search algorithm will not treat the padding 0 bits as part of a nucleotide encoding.

2.1.3 Offset data The offset data section of an index volume consists of two parts: the lookup table and the offset lists. The lookup table contains 4^k entries. The i-th entry contains the pointer to the start of the list of positions in the database called the *offset list*, where the k-mer with value i occurs.

Offset lists come after the lookup table in the data structure. Individual lists are terminated by a sentinel 4-byte word encoding 0. Our method of offset encoding ensures that 0 is not a valid value for a list entry. Positions of k-mers can be added to the offset lists in the order of their appearance in the underlying database, but in practice, the order is permuted for faster searching, as described in Supplementary Material.

To identify which k-mer positions to store, the set of *seed-eligible intervals* is computed for each input sequence. A *seed-eligible interval* is a subsequence of an input sequence that does not contain ambiguities or lower case (masked) letters. A k-mer is stored in the index if the following three conditions hold:

(1) the k-mer is fully contained within a seed-eligible interval;

(2) the seed-eligible interval is at least w nucleotides in length and

(3) the offset of the last letter of the k-mer relative to the start of the corresponding logical sequence is divisible by the stride, s.

Let $s_2 = 2^{\lceil \log_2 s \rceil}$; i.e. the smallest power of 2 greater than or equal to s. For a k-mer the actual value o stored in the offset list is computed in the following way:

$$o = {s_2}^2 + \frac{p}{s} + h \cdot ({s_2}^2 + \left\lceil \frac{2^B}{s} \right\rceil + 1),$$

where h is the logical sequence id, $\lceil 2^B/s \rceil$ is the number of bits used to encode the position of a k-mer within its logical sequence at stride s and p is the offset of the last letter of the k-mer relative to the start of the logical sequence. o is stored as a 32-bit wide unsigned integer.

Special offsets and prefixes. For some k-mer instances, extra information is stored along with its encoded position. Let d_l (d_r) be the distance from the first (last) base of the k-mer to the left (right) end of the corresponding valid interval. We call a k-mer instance *special* if either $d_l < s$, or $d_r < s$, or both. We define u_l (u_r) to be equal to d_l (d_r) if $d_l < s$ ($d_r < s$) and 0 otherwise.

Special prefixes are used in the offset lists by the seed search algorithm to ensure that reported seeds never cross the boundary of a seed-eligible interval. For special k-mer instances, an extra 32-bit prefix value $o_s = s_2 u_l + u_r$ is stored in the offset list immediately prior to their encoded position o. It is always the case that $o_s < {s_2}^2$ and $o \geq {s_2}^2$, so the prefix is easy to distinguish from the actual encoded position.

An example for offset entries. Consider the following subsequence from the first contig NT_019273.18 of the masked human genome starting at offset 8 500 000 (with the first base at offset 0):

gagaggACAACACTTAAAGGTTCAACTAGCAATA

With (default) values of $k=12$, $s=5$, chunk size 5 Mb, and chunk overlap of 100 bp, base at offset 8 500 000 in the input is in the second logical sequence and has offset 3 500 100 in its logical chunk. The 12-mers at offset 3 500 100 (gagaggACAACA) and 3 500 105 (gACAACACTTAA) are partially masked. Therefore, they are not part of any seed-eligible interval and are not added to the lookup table offset lists in the index structure. The next 12-mer considered for addition to the lookup table is at offset 3 500 110 (CACTTAAAGGTT), and that is in a seed-eligible interval. This is also a special offset as the distance to its seed-eligible interval boundary to the left is only four. The 12-mer at offset 3 500 115 (AAAGGTTCAACT) is also in a seed-eligible interval, but it is not a special offset.

Computations for CACTTAAAGGTT in the subsequence above resulting in the entries in the index structure for masked human genome are: (i) lookup table entry index of 4 702 383 using the bit encoding for the four bases, (ii) special offset value of 32 using the formula $2^{\lceil \log_2 5 \rceil} \cdot 4 + 0$, and (iii) offset value of 2 377 873 using the formula $(2^{\lceil \log_2 5 \rceil})^2 + 3500110/5 + 1 \cdot ((2^{\lceil \log_2 5 \rceil})^2 + \lceil 2^{\lceil \log_2 (5000000) \rceil}/5 \rceil + 1)$. The special offset value of 32 is followed by the offset value of 2 377 873 in the list of offsets for lookup table entry index 4 702 383.

2.2 Approach to testing

2.2.1 General setup Database indexing functionality is implemented in the NCBI C++ toolkit via the executables `blastn` and `makembindex`. For testing, we used statically linked versions of `blastn` and `makembindex` built from the toolkit sources of September 5, 2007. The executables were built for the 32-bit Intel x86 architecture under Linux OS kernel version 2.6.5 using GCC v4.0.1 compiler. The executables and some documentation are available in the directory:
ftp://ftp.ncbi.nlm.nih.gov/pub/agarwala/indexed_megablast

All tests were performed on a Dual Intel Xeon 5160 machine running at 3 GHz (two dual-core CPUs) with 8 GB of RAM. All runs were performed in single-thread mode. When measuring the running time of a program, three runs were performed and the median time was recorded, except for the large-scale test. The standard UNIX `time` utility was used to measure total running time. To measure the time used to search for seeds, `blastn` was instrumented with checkpoints that measured the time of seed searching procedures using Linux `gettimeofday` system call.

Instantiations of the following command line, including some variables, were used to run `blastn` for different tests.

```
blastn -db <database> -task megablast \
 -outfmt 6 -use_index <true|false> \
 [-index_name <index>]  -query <query> \
 [-filtering_db <osr_db>] \
 [-lcase_masking] [-dust <true|false>]
```

`-db <database>` specifies the name for BLAST database created using `formatdb` utility. `formatdb` is a part of NCBI BLAST software distribution.

`-task megablast` selects MegaBLAST module.

`-outfmt 6` is used to choose the tabular output format.

`-use_index` allows to choose whether to use database indexing functionality.

`-index_name <index>` is optional and only needed when using database indexing functionality. `index` in this case is the base name of the database index volumes created via the `makembindex` utility that sets up the data structures described in Supplementary Material. For example, if the index volumes are named `dbi.00.idx`, `dbi.01.idx`, …, then the value of `index` should be `dbi`.

`-query <query>` specifies the name of the FASTA formatted file containing the query sequence.

`-filtering_db <osr_db>` is optional and is used to enable masking the query for organism specific repeats. In this case `osr_db` is the name of the database containing the repeats.

Runs in which `-use_index` is set to `false` are referred to as either 'baseline' or 'non-indexed'.

Databases. The tests were run against human build 36 and mouse build 36 contig genome databases. The indices were created from unmasked databases and databases masked with WindowMasker (Morgulis *et al.*, 2006a), including low-complexity filtering by DUST (Morgulis *et al.*, 2006b). Each database was split into several volumes, so that each volume index was ∼1 GB in size, or less for the last database volume. The runs were done for each volume individually, and the sum over all volumes was recorded as the running time.

Queries. Queries were selected with a focus on human and mouse because a central goal was to use indexed MegaBLAST in the production NCBI web service to search the nucleotide sequences of these two genomes. For each organism, four query sets, each containing 100 queries, were used to test the performance of MegaBLAST. For production testing, we used 100 queries from human only. We call these query sets Qsmall, Qmedium, Qlarge, Quser and Qproduction. To form Qsmall, Qmedium and Qlarge, sample queries for three approximate sizes were randomly selected from within bacterial artificial chromosome (BAC) sequences from the genome being queried. The sizes are: small (∼500 bases, range: 501–506), medium (∼10 Kbases, range: 10000–10446), and large (∼100 Kbases, range: 100001–102087); 100 queries of each size were selected for each of human and mouse. The query sets Qsmall, Qmedium, Qlarge are available in subdirectories of ftp://ftp.ncbi.nlm.nih.gov/pub/agarwala/indexed_megablast/queries

The fourth set of queries, Quser, was created by first collecting real user submissions over a period of one week to the NCBI BLAST web service, with the restriction that the user clicked the link for querying the human genome or the link for querying the mouse genome. For each organism, 100 queries were selected at random from among these real user queries. When preparing Quser, we did not take note of any information about the user submitting the query, only the query sequence and parameter settings. Unlike the other query sets we used, Quser cannot be made public, so as to protect the confidentiality of users and usage of the NCBI Web BLAST service.

For production tests, we randomly selected 100 human sequences from the nt database (ftp://ftp.ncbi.nih.gov/blast/db/) as our query set, and we call this set Qproduction. The 100 query sequences ranged in size from 29 bases to 181 166 bases. The 50th percentile query length was 1044 bases and the 75th percentile query length was 2120 bases, roughly reflecting the length distribution of DNA queries submitted to the NCBI web pages.

2.2.2 Comparison of indexed MegaBLAST with miBLAST We compared the performance of miBLAST (Kim *et al.*, 2005) and NCBI MegaBLAST with indexed database support by running Qsmall and Qmedium against the unmasked database comprising contigs on human chromosomes 1 through 5. These chromosomes were selected because the size of the database is ∼1 GB. For all runs, the queries were masked using DUST algorithm (Morgulis *et al.*, 2006b), but not masked for organism-specific repeats.

The comparison to miBLAST is imperfect because of differences in the intended usage and engineering of miBLAST and indexed MegaBLAST. Some comparison on usage where functionality is similar provides more useful information than showing no comparison at all. When comparing against miBLAST, we used Qsmall and Qmedium but not Qlarge, because miBLAST is intended only for short queries. Conversely, indexed MegaBLAST includes some compromises in data structure design that damage its performance on Qsmall slightly, so that its performance on longer queries is competitive with or better than baseline MegaBLAST. miBLAST uses the BLASTN module of BLAST to compute alignments, which means that it takes longer and finds more imperfect local alignments than MegaBLAST. To quantify this tradeoff, we ran miBLAST and indexed

MegaBLAST on a large-scale test of 10 000 50mer queries extracted from human chromosomes 1-5. Because the queries are from the sequences in the database, one might expect each query to yield at least one perfect, full-length alignment, but no perfect match is reported for some queries because of the use of DUST filtering.

2.2.3 Comparison of stand-alone indexed and non-indexed MegaBLAST In all cases, each command line call to `blastn` involved one query. In other words, the query concatenation feature of `blastn` application was never used, so that query length can be used as a parameter in evaluating performance. Recent versions of `blastn` also support splitting of long queries. All queries used in the tests were short enough so that no query splitting happened.

To compare how the query length and the number of results affected the performance of the indexed database search compared to non-indexed case, the query sets Qsmall, Qmedium and Qlarge were run against unmasked indexed databases. No additional query masking was applied for these tests.

2.2.4 Comparison of indexed and non-indexed MegaBLAST in a production environment The NCBI BLAST web service utilizes a custom queueing system (ftp://ftp.ncbi.nih.gov/blast/documents/blast-sc2004.pdf) to process search requests and present results to the user. Depending on the database size, a database may or may not be split into smaller chunks that can be searched in parallel on worker nodes. Similarly, long queries can be split and short queries, even submitted by different users, can be combined into a single task. The system encourages cache reuse by steering tasks to worker nodes that have recently performed similar searches. While traditional MegaBLAST searches are parallelized and run in this manner, indexed MegaBLAST searches are run in serial on dedicated machines. A script runs periodically on these dedicated machines to ensure that the entire index resides in memory.

We used a script (http://www.ncbi.nlm.nih.gov/blast/docs/web_blast.pl) to submit one indexed and one non-indexed search for each query in Qproduction against the human genome, allowing 2 s between submissions. Reflecting the defaults of the NCBI BLAST web pages, the non-indexed queries were masked for low complexity regions and human repeats. The indexed queries were not masked, since the index incorporates masking information. Times recorded were the start-to-finish wall-clock time for each run, where each non-indexed search was spread over 10–20 worker nodes and the time reported does not combine wall-clock times for individual nodes.

3 RESULTS

To make BLAST databasse indexing feasible in production, a first requirement is that the amount of memory used by the index data structures should be reasonable. Table 1 shows the sizes of the index volumes in relation to the actual database length. The data are provided for the databases used for timing tests described in Subsections 2.2.3 and 2.2.4 as well as for a shorter *Drosophila* genome. The indices were created using $s=5$ and $k=12$. The last column of the table shows the upper bound provided by formula 1. The bound is pretty tight for unmasked databases. It is less tight in the masked case, because it does not take into account the actual number of unmasked bases.

A second requirement is that the amount of preprocessing time needed to create the index data structures should be reasonable. We measured the time it takes to create human unmasked, human masked, mouse unmasked and mouse masked indices as 1319.11 s, 1000.67 s, 1176.56 s, and 911.27 s, respectively. For each database, the time in the previous sentence is the sum of index volume creation times taken over all volumes of the database.

Table 1. Database index size for different source databases

Source database	Database size (Mbp)	Index size (Mbytes)	Bound (Mbytes)
Drosophila, unmasked	116.78	229.60	250.62
Drosophila, masked	116.78	204.74	250.62
Human chr. 1-5, unmasked	1025.20	1197.79	1204.46
Human chr. 6-13, unmasked	1077.86	1254.13	1259.75
Human chr. 14-Y, unmasked	767.77	926.56	934.16
Human chr. 1-8, masked	1493.03	1229.81	1695.68
Human chr. 9-Y, masked	1377.79	1137.62	1574.68
Mouse chr. 1-7, unmasked	1140.61	1297.20	1325.64
Mouse chr. 8-16, unmasked	1074.64	1222.98	1256.37
Mouse chr. 17-Y, unmasked	428.82	538.65	578.26
Mouse chr. 1-10, masked	1526.66	1258.19	1730.99
Mouse chr. 11-Y, masked	1117.42	932.09	1301.29

Windowmasker software (Morgulis *et al.*, 2006a) was used to generate masked databases. Windowmasker was run with default parameters and low complexity masking enabled. The bound column shows the upper bound calculated from formula 1.

Having shown that the data structures for database indexing can be created in a reasonable amount of memory and time, we proceed to evaluate the time for searching, especially the time for finding seeds. The performance advantage or disadvantage of indexed seed search, by which we mean the seed-finding phase of BLAST with an indexed database, depends on the size of the seed candidate lists (see Supplementary Material). When the database (or the query originating from the corresponding genome) is masked for repeats and low complexity regions, then the seed candidate lists are likely to be short. However, for an unmasked database and query, the indexed search may actually have worse performance compared to the non-indexed case, if the query contains highly repetitive regions.

3.1 Comparison of indexed MegaBLAST and miBLAST

Table 2 summarizes a performance comparison of miBLAST and indexed MegaBLAST. miBLAST performs best on very short queries; the query length of 500 bp used in Qsmall query set is on the upper end of the useful query length range for miBLAST. Since the performance of indexed MegaBLAST depends on the total number of matches, Table 2 contains separate rows for queries producing less than 5000 results and for the ones producing at least 5000 results. The performance of indexed MegaBLAST is better than that of miBLAST by at least 2.5 times in all cases except for the queries from Qsmall producing over 5000 results. In the latter case, the performance advantage of miBLAST is under 12%. The rightmost columns show that indexed MegaBLAST is faster than miBLAST on the vast majority of queries considered.

On a large-scale test of 10 000 50-mer queries, miBLAST required 76 times more time (20 h and 25 min versus 16 min) to find 22 times more local alignments (122.7 million versus 5.5 million). However, indexed MegaBLAST found a perfect length 50 match for 9648 queries, while miBLAST found a perfect match for 9269 queries. The superior performance of indexed MegaBLAST in finding perfect matches is unexpected, and the description of miBLAST in Kim *et al.* (2005) does not suggest to us why a perfect match should be

Table 2. Performance comparison of miBLAST and indexed MegaBLAST

Set	No. of results	Time (s)		Faster		
		MI	MB	MI	MB	Tied
Qsmall	<5000	170.20	23.21	0	89	0
	≥ 5000	161.11	182.72	10	1	0
Qmedium	<5000	129.16	39.26	0	17	0
	≥ 5000	15340.13	4237.12	6	75	2

The time represents the sum of median results per query. In this table and in Table 3, the number of results refers to the number of alignments reported, not the number of database sequences with at least one reported alignment. The queries are grouped based on the number of results. The three rightmost columns count the number of queries for which either software package was faster or the running times were considered a tie. The two running times for a query were considered a tie, if the time difference is <0.1s.

Table 3. Running time in seconds for baseline and indexed versions of MegaBLAST in the case of the unmasked human genome database

Set	<5000 results			≥ 5000 results		
	Count	Baseline	Indexed	Count	Baseline	Indexed
Qlarge	1	5.08	3.49	61	44173.96	43888.14
		1.46	0.29		961.24	1396.93
Qmedium	13	82.17	71.95	87	13009.04	13325.62
		24.06	6.29		305.10	745.59
Qsmall	88	118.05	56.51	12	528.85	570.84
		68.19	7.24		13.70	65.86

For each query set the top row contains the number of queries in the corresponding group and the total search time. The second row shows the time taken by the seed search phase only. For 38 queries in Qlarge, at least one version of MegaBLAST ran out of memory due to the large number of results.

missed by miBLAST on ∼4% of queries, where it is not blocked by DUST filtering.

The index structure used by miBLAST differs fundamentally from that described here in that for each k-mer, the miBLAST index stores only the sequence identifiers containing that k-mer but not the offsets. As demonstrated by Kim *et al.* (2005), the resulting space savings can be advantageous when queries are short. However, as query length grows, the fraction of database sequences that contain a k-mer of the query grows closer to one. Therefore, the fraction of database sequences that have to be considered for alignment to the query, which Kim *et al.* call the 'filtration ratio' grows closer to one and the relative performance of miBLAST deteriorates.

3.2 Comparison of stand-alone indexed and non-indexed MegaBLAST

Table 3 compares the performance of baseline and indexed searches. Queries are categorized depending on whether they yield a small number (<5000) of result alignments or a large number. In the former case, indexed seed search is 5–9 times faster than non-indexed search. In the latter case, however, indexed seed search is slower due to more cache misses. When the number of seeds is very large, the proportion of time spent in seed search is small; almost all time is spent in seed extension procedures. As a consequence, the relative decrease in total running time is rather small—between 0% and 8%.

For masked databases, indexed MegaBLAST is consistently faster than baseline across all query sets, as shown by Table 4. In these runs, the baseline search is not performed with the query unmasked because baseline MegaBLAST cannot utilize database masking information. Masking of the query results in a small number of seeds in both the indexed and baseline runs, making the amount of time spent in later processing of seeds and alignments comparable. The performance advantage is achieved partly because in the indexed run, a substantial fraction of the masked database is not scanned for potential seeds. The decrease in total running time is also more substantial in the masked case because the extension procedures have much fewer seeds to process.

For masked databases, indexed MegaBLAST searches with the query also masked for repeats were done to evaluate if masking the query makes sense when the database is masked already. The results (Table 4) show that although the seed search phase is somewhat

Table 4. Running time in seconds for baseline and indexed versions of MegaBLAST in the case of masked genomes and query masked (indicated by yes) or unmasked (indicated by no)

Set	Human masked			Mouse masked		
	Baseline	Indexed		Baseline	Indexed	
	Yes	Yes	No	Yes	Yes	No
Qlarge	1097.11	511.76	418.31	1702.35	736.74	613.17
	447.42	108.32	127.60	447.50	98.71	114.07
Qmedium	266.43	90.80	95.36	437.45	146.63	140.94
	176.21	23.97	28.64	162.15	22.82	25.63
Qsmall	104.79	27.24	26.44	148.98	64.93	67.76
	76.83	5.27	5.33	70.10	5.32	5.11
Quser	144.06	46.12	NA	164.21	82.82	NA
	93.49	8.27	NA	83.18	7.01	NA

For each query set the top row is the total search time, and the second row is the time taken for the seed search phase only. NA—not applicable.

faster in the case of masked queries, the total running time may be worse. This anomaly occurs because masking the query involves calling a separate BLAST search against a repeats database and the overhead of doing that actually offsets any performance benefit due to reduced number of seeds.

3.3 Comparison of indexed and non-indexed MegaBLAST in a production environment

Figure 2 presents run times for 100 indexed searches and 100 non-indexed searches in a production setting, with the exception that the compute cluster used for this test was not being used for other tasks. Time reported is for the second of the two searches done for each query and each method. The average wall-clock time was 0.5 s for an indexed search and 0.8 s for a non-indexed search. The 75th percentile times offer a better measure of the user experience; this was 0.38 s for an indexed search and 1.37 s for a non-indexed search. Times are plotted against logarithm of query size to show

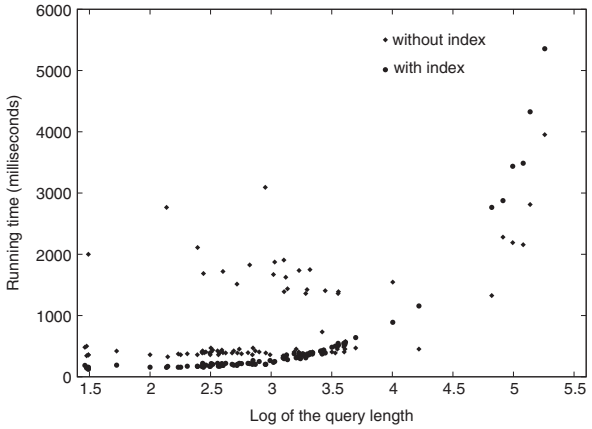

Fig. 2. Wall-clock times for 100 indexed and non-indexed searches in a production setting, as a function of logarithm of query length. Considering times to be tied if they are within 0.01 s, indexed search is faster 75 times, non-indexed search is faster 19 times, and they tie on 6 queries. Indexed search is faster on shorter queries and slower on the longest queries.

that indexed search is generally faster for shorter queries and non-indexed search is faster for longer queries.

Since the non-indexed search uses 10–20 nodes while the indexed search uses one node, non-indexed search may use a lot more total CPU time than indexed search even when the non-indexed wall-clock time is less. So long as the dedicated machines for indexed search can keep up with incoming MegaBLAST queries to the human and mouse genomes, users perceive a much faster response time for the genome-specific queries because of the combined effects of faster runs and no competition from queries to other databases. Quantifying the benefits to throughput for all types of queries is difficult, since the workload for the dedicated and shared machines is variable over time.

4 DISCUSSION

We presented a new implementation of the seed search phase of MegaBLAST (Zhang *et al.*, 2000) in which seeds are found by searching an index structure of *k*-mers derived from preprocessing the database. We showed that this 'indexed MegaBLAST' is faster than the 'baseline MegaBLAST', which preprocesses the query, in most cases and especially for masked databases. When indexed MegaBLAST is slower because there are too many seeds, performance degradation is limited enough that the code can be used in production. Indexed MegaBLAST satisfies the requirement that the outputs are identical or nearly identical to baseline MegaBLAST, so long as all options are set consistently. In addition, usage of the preprocessor program `makembindex` is being tested at NCBI for other projects such as alignment of short sequences generated by 'next-generation sequencing technologies' to a reference genome.

Undertaking this project was positively influenced by several research papers suggesting that preprocessing the database for initial lookup could give faster search times than preprocessing the query. We decided to put our effort into a relatively simple data structure design, the seed search phase of the MegaBLAST module, and support for masked databases, so as to get some use of database indexing into production BLAST usage more quickly. Now that this overarching goal has been achieved, there are many possibilities to extend and improve our implementation.

First, our implementation could be extended to the BLASTN module by changing the strides and *k*-mer lengths. To get good performance, some alternative data structure for long lists may be necessary, as discussed subsequently. Extending our work to protein searching would be much more challenging due to the increased alphabet size. Williams and Zobel (2002) implemented a research prototype for nucleotide and protein searching, called CAFE, in which they preprocessed the database into an inverted index. They demonstrated quite impressive performance for CAFE, but to get that performance, they substantially changed the searching methods and scoring used in phases of the homology search beyond the initial word lookup. Thus, to make a production version that replaces some BLAST module would be more difficult than the seed search phase replacement we carried out.

There are some possible algorithmic improvements to our implementation that could be tried without a major redesign. We summarize two of these possible changes. Instead of using a uniform stride (by default $s=5$) when selecting *k*-mer occurrences to put in the index, one could stride by a non-uniform increment in such a way that every potential seed (default length 28) is adequately represented by one or more *k*-mer substrings. The stride at each position could be chosen so that the *k*-mers whose occurrences are in the index are as infrequent as possible. This would make the *k*-mer lists longer but speed up the seed search phase. A preliminary assessment of the human genome suggested that using non-uniform strides could simultaneously increase the length of the *k*-mer lists by 9% and shorten the weighted average length by 25%; 'weighted average' means that each *k*-mer is weighted by its number of occurrences in the genome. Using non-uniform strides would make the preprocessing take longer, but we do not consider database preprocessing time as a primary measure of performance, since preprocessing is done outside MegaBLAST. Another possible improvement is that long singly linked lists of *k*-mers could be replaced by a more sophisticated structure such as a hash table or some sort of tree (Giladi *et al.*, 2002).

Possible improvements suggested by research implementations of database indexing, which require a major redesign, include the following. Using '*q*-gram filtering', which considers the number of *k*-mer matches within a region of dynamic programming alignment matrix, can provide a more stringent filter for candidate matches than single seeds (Rasmussen *et al.*, 2006). The database sequences could be transformed into data feature vectors as proposed in Lee *et al.* (2007). In the same spirit as the *k*-mer reordering we described under optimizations already implemented (Supplementary Material), some subset of 'pier' *k*-mers could be stored in a smaller, more rapidly accessible data structure as proposed in Cao *et al.* (2004). Finally, the sequence representation implemented in MICA (Stokes and Glick, 2006) could allow for a space efficient way to store database indices for smaller values of *k* [Stokes and Glick (2006) state $k=7$ as the effective upper limit for *k*]. Such indices could be used for more sensitive searches currently provided by BLASTN.

In sum, one part of NCBI's web BLAST service has been using indexing of the database rather than indexing of the query in the seed search phase since October 2007. Production usage of indexed MegaBLAST and the tests herein validate the performance improvements that have been claimed for database indexing.

Further improvements to our implementation, possibly requiring a major redesign, may be achievable.

ACKNOWLEDGEMENTS

Thanks to Yuri Merezhuk for providing statistics on the use of the indexed MegaBLAST service and for help in assembling the Quser test set. Thanks to Ivan Ovcharenko for testing the command line version of indexed MegaBLAST and suggesting improvements to the documentation.

Funding: This research is supported by Intramural Research Program of the National Institutes of Health, NLM.

Conflict of Interest: none declared.

REFERENCES

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

Cao,X. *et al.* (2004) Piers: an efficient model for similarity search in DNA sequence databases. *ACM SIGMOD Record (Special Issue on Data Engineering for Life Sciences)*, **33**, 39–44.

Gertz,E.M. *et al.* (2006) Composition-based statistics and translated nucleotide searches: improving the TBLASTN module of BLAST. *BMC Biol.*, **4**, 41.

Giladi,E. *et al.* (2002) SST: an algorithm for finding near-exact sequence matches in time proportional to the logarithm of the database size. *Bioinformatics*, **18**, 873–879.

Jiang,X. *et al.* (2007) Survey on index based homology search algorithms. *J. Supercomput.*, **40**, 185–212.

Kent,W.J. (2002) BLAT— the BLAST-like alignment tool. *Genome Res.*, **12**, 656–664.

Kim,Y.J. *et al.* (2005) miBLAST: scalable evaluation of a batch of nucleotide sequence queries with BLAST. *Nucleic Acids Res.*, **33**, 4335–4344.

Lee,A.J.T. *et al.* (2007) A novel filtration method in biological sequence databases. *Pattern Recog. Lett.*, **28**, 447–458.

Morgulis,A. *et al.* (2006a) A fast and symmetric DUST implementation to mask low-complexity DNA sequences. *J. Comp. Biol.*, **13**, 1028–1040.

Morgulis,A. *et al.* (2006b) WindowMasker: Window-based masker for sequenced genomes. *Bioinformatics*, **22**, 134–141.

Ning,Z. *et al.* (2001) SSAHA: A fast search method for large DNA databases. *Genome Res.*, **11**, 1725–1729.

Rasmussen,K.R. *et al.* (2006) Efficient q-gram filters for finding all ϵ-matches over a given length. *J. Comp. Biol.*, **13**, 296–308.

Shiryev,S.A. *et al.* (2007) Improved BLAST searches using longer words for protein seeding. *Bioinformatics*, **29**, 2949–2951.

Stokes,W.A. and Glick,B.S. (2006) MICA: desktop software for comprehensive searching of DNA databases. *BMC Bioinformatics*, **7**, 427.

Williams,H.E. and Zobel,J. (2002) Indexing and retrieval for genomic databases. *IEEE Trans. Knowl. Data Eng.*, **14**, 63–78.

Zhang,Z. *et al.* (2000) A greedy algorithm for aligning DNA sequences. *J. Comp. Biol.*, **7**, 203–214.