Published online August 1, 2005

Nucleic Acids Research, 2005, Vol. 33, No. 13 ***4335–4344***
doi:10.1093/nar/gki739

miBLAST: scalable evaluation of a batch of nucleotide sequence queries with BLAST

You Jung Kim, Andrew Boyd[1], Brian D. Athey[1] and Jignesh M. Patel*

Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109, USA and [1]Michigan Center for Biological Information, University of Michigan, 3600 Green Court, Ann Arbor, MI 48109, USA

Received March 18, 2005; Revised May 26, 2005; Accepted July 11, 2005

ABSTRACT

A common task in many modern bioinformatics applications is to match a set of nucleotide query sequences against a large sequence dataset. Existing tools, such as BLAST, are designed to evaluate a single query at a time and can be unacceptably slow when the number of sequences in the query set is large. In this paper, we present a new algorithm, called miBLAST, that evaluates such batch workloads efficiently. At the core, miBLAST employs a *q*-gram filtering and an index join for efficiently detecting similarity between the query sequences and database sequences. This set-oriented technique, which indexes both the query and the database sets, results in substantial performance improvements over existing methods. Our results show that miBLAST is significantly faster than BLAST in many cases. For example, miBLAST aligned 247 965 oligonucleotide sequences in the Affymetrix probe set against the Human UniGene in 1.26 days, compared with 27.27 days with BLAST (an improvement by a factor of 22). The relative performance of miBLAST increases for larger word sizes; however, it decreases for longer queries. miBLAST employs the familiar BLAST statistical model and output format, guaranteeing the same accuracy as BLAST and facilitating a seamless transition for existing BLAST users.

INTRODUCTION

A common query in a number of bioinformatics applications is to search a large nucleotide sequence database using a set of nucleotide sequence queries. For example, when validating the Affymetrix (oligonucleotide) probe set against the UniGene (EST) database, one needs to search the quarter million Affymetrix probes against the most recent UniGene release. Another example is using one animal model microarray against a different species, searching the chip probe set against the ESTs of the new species to validate the probes in the new species (1,2). A final example is in designing small interfering RNAs (siRNAs) libraries, where one needs to validate that the siRNAs only interfere with a single mRNA. A common characteristic of these types of applications is that a large batch workload of queries must be evaluated against a large database. Often the databases that are being searched in such scenarios are updated frequently (such as the periodic updates to GenBank), which requires periodic re-evaluation of these batch workloads.

One way of evaluating such batch workloads is to execute each query in the workload one at a time using a local-sequence alignment tool, such as BLAST (3,4). However, in practice, with large batch sizes this method is computationally very expensive. Clearly, a tool that can significantly speed up the evaluation of such workloads is very valuable. The focus of this paper is on the design of such a tool called miBLAST (pronounced as 'me-BLAST').

We note that a number of previous research investigations have developed techniques for efficiently evaluating batch workloads. These tools (5–7) are designed for specialized biological applications, such as aligning ESTs to a genome of similar species, and improve performance by using an index of non-overlapping *q*-grams. However, these approaches often sacrifice some loss in sensitivity for performance gains. MegaBLAST (8) uses a greedy algorithm and can be an order of magnitude faster than regular BLAST. However, the use of MegaBLAST is limited in aligning highly similar sequences with large word sizes. MPBLAST (9) directly improves the speed of NCBI BLAST and WU-BLAST (http://blast.wustl.edu) by multiplexing query sequences, thus reducing the number of database searches. MPBLAST essentially concatenates the queries in the workload and sends a single long query to BLAST.

A new feature in NCBI BLAST essentially implements the MPBLAST-like multiplexing technique for batch queries and produces alignments that are identical to running the queries in

*To whom correspondence should be addressed. Tel: +1 734 647 1806; Fax: +1 734 763 8094; Email: jignesh@eecs.umich.edu

the batch one at a time. BLAST++ (10) exploits commonality of short words among queries and shares results with queries containing these common words. Consequently, the performance of BLAST++ is highly dependent on the level of commonality in queries. Another method for speeding up the processing of batch workloads is to make use of parallel processing techniques on a cluster of machines using the mpiBLAST (11) method. This method essentially parallelizes BLAST searches by segmenting a database and executing each segmented portion of the database in a node in the cluster. The drawback of this approach is that it requires access to a cluster.

In this work, we propose an efficient and practical method for evaluating a batch workload, which consists of a large number of queries. Furthermore, given the popularity and familiarity of existing users with BLAST, we want our tool to exactly mimic the behavior and functionality of BLAST. In other words, we want our tool to have the same sensitivity as BLAST and employ the same statistical model and data formats for input and output.

While miBLAST is a general algorithm for sequence similarity matching between a batch workload and a database of sequences, it is especially useful in settings where the database and the workload consist of a large number of sequences (rather than a few very long sequences). Common examples of relevant biological applications of miBLAST include evaluating a large number of oligonucleotide probes, cDNA sequences or ESTs against a large database of ESTs.

In order to develop a more suitable algorithm for efficiently evaluating batch workloads, it is first necessary to identify the reasons as to why the BLAST algorithm is not efficient for such batch workloads. The BLAST algorithm essentially scans the entire database for each query sequence. In each scan of the database, the algorithm checks each database sequence to find any common short words between the database and the query sequences. In case of a word hit, the word alignment is extended to produce a complete alignment. Consequently, to evaluate a batch workload with n queries, the BLAST method will require n scans of the database and will compare each sequence in the batch with each sequence in the database.

This observation enables us to design the miBLAST algorithm that speeds up the evaluation of batch workloads. At the core, the algorithm employs a q-gram filtering and an index join technique that processes two q-gram indices (12). In a single scan of the two indices, the join method efficiently computes an initial word hit list for all the query sequences. The join also determines a set of *filtered* database sequences that have potential matches for the queries. Then, miBLAST only examines these filtered database sequences to produce the actual alignments. In practice, only a few database sequences match a given query, and consequently miBLAST does not have to examine the majority of sequences in the database.

The use of the q-gram index has also been explored in several sequence-searching applications (5–7). However, these applications are limited by the database or the main memory size, since they require that the whole index resides in main memory. In contrast, our work uses disk-based indices and can work with arbitrary large datasets.

We have evaluated miBLAST on an a number of actual batch workloads, including a workload to validate the labels of the Affymetrix probes (Human Genome U133A Set) with the current version of UniGene. The Affymetrix probe set consists of ~250 000 queries, which are on average 25 nt long. Each sequence in this workload is searched against the UniGene Homo sapiens dataset, which has 5 064 621 sequences and ~3.19 gigabases. Using the default parameters in the NCBI BLAST (including a word size of 11), this workload can be evaluated on a single machine in 27.27 days (9.50 s per query). In contrast, miBLAST processes this same workload on the same machine in 1.26 days (0.44 s per query). The resulting performance improvement achieved by miBLAST in this case is about a factor of 22. However, depending on the choice of parameters, miBLAST can outperform BLAST by even larger margins. For example, using a word size of 23 can result in a 45-fold performance improvement over regular BLAST. These results clearly demonstrate the effectiveness of our search tool.

Finally, miBLAST is built as an module that is integrated with the existing NCBI BLAST source code. This design allows miBLAST to reuse the alignment extension and result formatting components of the BLAST code base. As a result, miBLAST employs the traditional BLAST statistical model and outputting format. Because many users are very familiar with these aspects of BLAST, we expect that current BLAST users who need to evaluate batch workloads can easily transition to using miBLAST.

MATERIALS AND METHODS

The key strategy employed by miBLAST is to use q-gram indices to quickly identify the set of database sequences that contain word hits for each query sequence. Since for any query sequence only a small fraction of the database sequences actually have a common word hit, miBLAST only has to examine the small set of sequences that are *filtered* through the index search. These filtered sequences are then retrieved and the word hits are expanded to produce the actual alignments. In the following sections, we first describe the q-gram index structure, and then we describe the miBLAST filtering algorithms in detail.

Notations

The miBLAST algorithm considers two sets of sequences: a query sequence set $Q = \{q_1, q_2, \ldots, q_i, \ldots, q_m\}$ and a database sequence set $D = \{d_1, d_2, \ldots, d_j, \ldots, d_n\}$, where q_i and d_j represent sequences. All sequences in Q and D are assigned a unique sequence ID. A word w is defined as a string having a fixed length l. A word hit is defined as an ordered pair (i, j) such that the query sequence q_i and a database sequence d_j share a word in common.

Index structure and construction

The structure of the miBLAST q-gram index (12) is as follows: the q-gram index over a set of sequences contains an entry for every unique overlapping word of length l in the set. Each index entry stores the list of sequence IDs that contain the corresponding index word. Note that the reference to the sequence is simply based on the sequence ID and does not include the offset in the sequence where the word is located. Consequently, if a word appears multiple times in a given sequence, the index entry only refers to the sequence once. This decision to not store the actual offset was made to reduce

the size of the miBLAST indices. (The algorithm can be generalized to work with a q-gram index in which the offset position is also stored.) An example of a q-gram index is shown in Table 1.

A key advantage of this index structure is its efficiency. The index construction time is linear [$O(n)$] in the size of an input sequence set. In addition, searches for a word hit is very efficient as it takes constant time [$O(1)$].

Because the q-gram index has one index entry for every unique word of length q, the size of the index can be large and can exceed the size of the available main memory. To overcome this problem, miBLAST employs a disk-based implementation of the q-gram index. The index is stored on disk, and index entries are fetched from disk on demand. In addition, miBLAST is designed so that the index accesses are largely sequential (as opposed to the much more expensive random I/O). We note that the use of disk-based q-gram index in miBLAST is in contrast to other q-gram-based approaches (5,6), which employ an in-memory index scheme. Since the q-gram index is typically larger than the actual database, the use of in-memory indices implies that the q-gram method (5,6) cannot be used with very large databases.

We also note that the q-gram index used by miBLAST indexes all overlapping words. This scheme ensures that the miBLAST algorithm has no loss in sensitivity over the BLAST method. In contrast, the use of non-overlapping words in other approaches (5,6) results in a loss in sensitivity, but produces a smaller index. We believe that the use of a larger index structure in miBLAST is justifiable since one of the design goals of miBLAST is to provide the same sensitivity as BLAST.

Furthermore, to ensure that there is not a big performance penalty in using disk-based indexes, miBLAST makes careful use of sequential accesses. The index entries are sorted by the logical file offset positions provided by the operating system, and miBLAST accesses these logical index blocks sequentially. Such logical sequential access often closely corresponds to a physically sequential disk access since operating systems try to store the data in a file in physically contiguous disk blocks. This technique of using sequential access has important performance implications since scanning physically adjacent disk blocks is much more efficient than randomly accessing disk blocks that are spread across the disk. In addition, operating systems often prefetch adjacent blocks of data, which further improves the performance of sequential accesses.

Word length is a critical BLAST parameter, which specifies the size of a word that is used to detect a word hit. As different queries may specify different word length, miBLAST must employ a strategy for dealing with different word lengths. One naive strategy that could be employed is to build a q-gram index for all possible word lengths. However, the cost of building all such indices can be prohibitively large. To avoid this high cost, miBLAST employs methods that allow an index to be reused even with queries that specify a different word lengths.

In fact this flexibility in the use of the q-gram index for different query word lengths is a key difference between the use of q-gram index in miBLAST and previous q-gram-based approaches (5,6). With miBLAST in practice one could store just a single q-gram index based on the smallest supported word length parameter. Alternatively, we could choose to maintain a small number of q-gram indices and for a given query pick the q-gram index with the q value that is closest to the specified query word size.

Table 1. An example of a q-gram index structure

Sequence ID	Sequence	Word (w)	Sequence ID
1	ACAAAAA	AAA	1 2
2	AAAAAAAAC	AAC	2 3 4
3	CAACAACAA	ACA	1 3 4
4	CAACAACAA	CAA	1 3 4

The two left columns represent a sequence dataset and the two right columns show the index built on the dataset using $l = 3$.

The miBLAST algorithm

The miBLAST algorithm consists of three primary steps. First, two q-gram indices are constructed—one on the query set and the other on the database set. Second, using these indexes, the filtering algorithm selects database sequences that contain potential word hits for each query sequence. Finally, the BLAST alignment module is invoked on the filtered database sequences to generate the actual alignment.

The miBLAST method is outlined in Algorithm 1. miBLAST takes as input a set of query sequences, a set of database sequences and a word length to be used in a BLAST search. Based on this input, miBLAST starts by building a database index. In practice, the database index is often prebuilt and is stored on disk so that the same index can be reused for many searches.

Notice that in lines 3–7 of Algorithm 1 miBLAST runs different filtering algorithms depending on the word length l.

Algorithm 1. miBLAST (Q, D, l)

INPUT:
- $Q = \{q_1, q_2, \ldots, q_i, \ldots, q_m\}$ is a set of query sequences
- $D = \{d_1, d_2, \ldots, d_j, \ldots, d_n\}$ is a set of database sequences
- ID is a unique number assigned to each sequence, q_i and d_j in Q and D
- m is the word length used in database index construction
- l is a word length in a BLAST search

VARIABLES:
- $D_{ID} = \{1,2, \ldots, n\}$ is a set of database sequence IDs in D
- $F_i \subseteq D_{ID}$ is a set of sequence IDs that have word hits with q_i
- $Filtered = \{F_1, F_2, \ldots, F_m\}$ is a set of F for all q_i in Q
- $DIndex$ is a q-gram index for D
- $QIndex$ is a q-gram index for Q
- $DIndex.L(w)$ is an operation returning the list of sequences containing a word w from DIndex.

```
1: Build a DIndex on D with a word length m, if the index don't already exist.
2:
3: if l ≤ m then
4:    Filtered=INDEX-JOIN FILTER (DIndex, Q, l, m)
5: else if l > m then
6:    Filtered=SLIDING-WINDOW FILTER (DIndex, Q, l, m)
7: end if
8:
9: for all sequence q_i in Q do
10:   for all sequenceID j in F_i of Filtered do
11:       Find alignments with a q_i in Q and a d_j in D
12:   end for
13: end for
```

If the query word length l is smaller or equal to the index word length m, then miBLAST uses an index join algorithm for sequence filtering (shown in Algorithm 2). However, if $l > m$, then miBLAST uses the sliding window filtering method, which is shown in Algorithm 3.

Algorithm 2. INDEX-JOIN FILTER (*DIndex*, *Q*, *l*, *m*)

```
1: Build a QIndex on Q with a word length l
2: Create a bitmap[|Q|, |D|]
3:
4: for all word w in QIndex do
5:   If QIndex.L(w) ≠ NULL & DIndex.L(w) ≠ NULL then
6:     L_q ← QIndex.L(w)
7:     L_d ← DIndex.L(w)
8:     for i ← 1 to |L_q| do
9:       for j ← 1 to |L_d| do
10:        bitmap[L_q[i], L_d[j]] = TRUE
11:      end for
12:    end for
13:  end if
14: end for
15:
16: for i ← 1 to |Q| do
17:   for j ← 1 to |D| do
18:     if bitmap[i, j] = TRUE then
19:       F_i = F_i ⋃ {j}
20:     end if
21:   end for
22: end for
23:
24: Filtered = {F_1} ∪ {F_2} ∪ ... ∪ {F_m}
25: return Filtered
```

Algorithm 3. SLIDING-WINDOW FILTER(*D*Index, *Q*, *l*, *m*)

```
1: Create a bitmap[|Q|, |D|]
2: Create a counter[|D|]
3:
4: for all sequences q_i in Q do
5:   for all words w of length l in q_i do
6:     initialize the counter
7:     for all all substrings w' of length m in w do
8:       if DIndex.L(w') ≠ NULL then
9:         L_d ← DIndex.L(w')
10:        for j ← 1 to |L_d| do
11:          counter[L_d[j]] ++
12:        end for
13:      end if
14:    end for
15:    for j ← 1 to |D| do
16:      if counter [j] = l − m + 1 then
17:        bitmap [i,j] = TRUE
18:      end if
19:    end for
20:  end for
21: end for
22:
23: for i ← 1 to |Q| do
24:   for j ← 1 to |D| do
25:     if bitmap [i, j] = TRUE then
26:       F_i = F_i ⋃ {j}
27:     end if
28:   end for
29: end for
30:
31: Filtered = {F_1} ∪ {F_2} ∪ ... ∪ {F_m}
32: return Filtered
```

Index lookup

The index lookup operation, $L(w)$, returns a list of sequence IDs containing a query word w of length l. When the query word length is the same as the index word length, i.e. $l = m$, the list can be retrieved by a single index lookup. However, when $l < m$, the index lookup function performs multiple index lookups. These lookups search for all index words whose prefix matches the entire query word. The final result list is constructed by simply computing a union of the sequence lists returned by each lookup. Since our q-gram index is lexicographically sorted, these multiple index lookups essentially result in a sequential index scan, making the multiple index lookup step very efficient.

Index join filtering

In order to find all word hits between the query sequence set Q and the set of database sequences D, miBLAST employs an index join technique. Two indexes, one for each D and Q, are built. After the index construction is completed, miBLAST joins these indexes based on words. In our implementation of the q-gram index, our index function simply uses the least-significant bits of each character in the word, which essentially produces index entries that are in lexicographically ordered.

Starting with the first word in the query index, for each word w we probe the query index to obtain a list $L_q(w) = (w, y, z, \ldots)$ such that q_x, q_y and q_z contain a word w as its substring. Next, the database index is probed with the same word to obtain a list $L_d(w) = (p, q, r, \ldots)$. Then, we take the cartesian product of the entries in these two lists, to generate all the potential word hit pairs, (x, p), (x, q), (x, r), (y, p), (y, q), $(y, r), \ldots$ Each pair indicates a query sequence that has a potential word match with a database sequence. These sequences are then examined for actual alignments using the BLAST alignment algorithm. This process continues until we have examined all word hit pairs in Q and D.

Notice that the join of the two indices produces a complete list of database and query sequence pairs that share a common word hit. There are two interesting properties associated with the entries in this complete list. First, this list can have a number of duplicate pairs since a single database sequence and a query sequence may share a number of common words. Second, this list is not sorted by the query sequence number. A naive way of producing actual alignment would be to simply expand each sequence pair and output the result as alignments are produced. However, because of the first property of this list, a single database sequence may have to be fetched multiple times for generating alignments for the same query, which is inefficient. Second, the output that is produced will have alignments that are not grouped by the query sequence numbers. Consequently, the resulting output will have to be sorted on the query sequence number before presenting the results to the user. This sort operation can be expensive, especially with a large batch workload.

To address these issues, we employ an efficient bitmap approach. We employ a 2D binary bitmap data structure, which has $|Q| \times |D|$ entries, where $|Q|$ and $|D|$ represent the number of query and database sequences each. All bitmap entries are initially set to FALSE. Then, whenever a word hit pair, (i, j), is found, the corresponding entry (i, j) in the bitmap is set to TRUE. At the end of the index join, the bitmap

entries that are set to TRUE represent database and query sequence pairs that have at least one common word. By sequentially scanning the rows of this bitmap, for each query sequence we can generate a sorted list of database sequence IDs that should be extended in the alignment phase. The results that we produce are naturally grouped by the query sequences (a local sort is still needed to order the results for each query by the scores), and a database entry is fetched at most once for each query sequence.

Sliding-window filtering

Since the cost of constructing a q-gram index for a large database can be expensive, it is desirable to consider methods that allow reusing an existing index whenever possible. However, since the word length for a search is a user-defined parameter, reusing an index with a word length other than the specified word length parameter is challenging. In this section, we discuss the sliding-window technique, which allows miBLAST to reuse an index when the query word length is greater than the index word length.

To explain the sliding-window method, we will use the following notations: let $s[1..k]$ denote a word of length k in a sequence s, and let m and l denote the lengths of the index and query words, respectively.

The sliding-window method is based on the observation that if a query word $s[1..l]$ exists in a database sequence d_i, then all of its substrings of length m, namely $s[1..m]$, $s[2..m+1]$, $s[3..m+2], \ldots, s[l-m+1..l]$, must also exist in the index of the sequence d_i. Consequently, the index entry for each of these substrings must contain the sequence ID i. This property provides a necessary condition for the word $s[1..l]$ to be found in a sequence d_i. Note that this condition is not a sufficient condition, which implies that this technique may produce false positives. However, the false positives can be easily eliminated by actually checking for the precise word hits when the database sequence is retrieved. We note that a similar technique is also used in A/G BLAST (J. Klivington, personal communication) to find word matches of length l between a query and a database using a lookup table for words of length m in a query sequence.

The algorithm for the sliding-window method is shown in Algorithm 3.

Implementation

The miBLAST implementation consists of three main components: q-gram index construction, filtering and alignment generation components. The index construction component takes a formatted database as its input, which is generated by the NCBI formatdb utility, and then builds a q-gram index on the database. The filtering component takes as inputs the database index and a batch of query sequences and computes a set of database sequences containing word hits for the query sequences. Finally, the alignment generation component computes the actual sequence alignments. The alignment generation component uses the standard NCBI BLAST alignment generation component, which is modified so that it only needs to examine a subset of database sequences produced by the filtering component.

miBLAST is written in C using the NCBI toolkit. The current miBLAST implementation uses NCBI BLAST v.2.2.8 and supports querying on nucleotide datasets. These modifications have added less than one hundred lines of source code, and these modifications are clearly marked in our public release.

RESULTS

In this section, we present results of an empirical evaluation of miBLAST. The database that we used for our empirical evaluation is the NCBI Human UniGene build #177. This database contains 5 064 621 sequences and roughly 3.19 gigabases.

For the empirical evaluation, we used a number of query workloads to test the impact of the following parameters: the batch workload size, word size parameter and the query sequence length. The first two query sets are drawn from the Human Genome U133A probe sets containing oligonucleotide sequences from Affymetrix and the RefSet Oligos for the human genome from Illumina Inc. The Affymetrix probe set contains 247 965 sequences with an average length of 25. The Illumina probe set contains 22 740 sequences with an average length of 70. These query sets are used to see if the probe labels given by the company are consistent with current UniGene clusters, since the clusters change over time. In addition, we also extracted query sequences of various lengths (from 16 to 512 bp) from the EST human database. We used these EST query sets to measure the impact of query lengths on the performance of various algorithms.

In evaluating the miBLAST performance, we considered comparing miBLAST with other tools, such as MegaBLAST (8), BLAST++ (10), NCBI-BLAST and MPBLAST (9). However, as described below a number of these methods are not directly comparable with miBLAST.

We did not compare miBLAST with MegaBLAST, as it is known that the sensitivity of MegaBLAST can be less than the sensitivity of standard BLAST.

For BLAST++, the current version cannot handle the UniGene database build #177 due to its large size, so we used the first half of the UniGene database to compare miBLAST with BLAST++.

In this paper, we extensively compare miBLAST with NCBI BLAST v.2.2.8. For batch workloads, NCBI BLAST can be used in two ways. The first approach, which we call naive BLAST, iteratively runs BLAST for each query in the workload. The second approach uses the relatively new '-B' option in BLAST. This approach, which we call BLAST-B, essentially implements the multiplexing method used in MPBLAST (9). In this approach, a specified number of queries in the batch are multiplexed (i.e. concatenated) to produce a single large query string. Then, the traditional BLAST method is invoked on this concatenated query string. While BLAST-B reports all the alignments that are found with naive BLAST, it produces slightly different output than naive BLAST. BLAST-B does not produce summary statistics for each query, but only produces a single summary statistics for the entire concatenated query. In contrast, miBLAST produces the same output as naive BLAST.

To run BLAST-B, the user must specify the number of queries to be concatenated. In the current version of BLAST, the upper limit on the number of concatenated queries is 255. However, although a user can specify a batch size from

anywhere between 2 and 255, we have noticed that there is often an optimal batch size. In general, the performance initially improves as the degree of concatenation is increased, but starts dropping gradually after a certain point. This optimal point can change depending on query sizes, datasets and BLAST search parameters. For BLAST-B, we ran several experiments to manually find the optimal batch size for each workload and used the optimal batch size for each BLAST-B run.

Default BLAST parameters were used for running the three different methods. All experiments were run on a machine with a 2.2 GHz AMD Opteron processor and 4 GB RAM running the Linux 2.6.9 kernel. All measurements are based on the performance with a cold cache, which essentially means that before each experimental run the system cache has no pre-cached data. All times reported in the following experiments are actual wall-clock time taken to run the queries.

Effect of batch workload size

To examine the effect of batch workload size on miBLAST, naive BLAST and BLAST-B, we ran experiments by increasing the batch workload size (the number of queries in the batch) from 1000 to 4000. Figure 1a and b shows the relative speedup to naive BLAST for the Affymetrix (25 bp) and Illumina (70 bp) workloads, respectively.

As shown in Figure 1, miBLAST is significantly faster than the other two methods. The performance of miBLAST improves as the batch size increases, and for a batch size of 4000 queries, miBLAST is 21.6 and 9.9 times faster than naive BLAST for the 25 and 70 bp queries, respectively, and 4.5 and 2.7 times faster than BLAST-B for the 25 and 70 bp queries, respectively.

To understand why the performance of miBLAST improves as the batch size increases (Figure 1), consider Table 2, which shows the breakdown of the filtering cost (the index join component), and the cost of calling the BLAST alignment method for different workloads. As can be seen from this table, both the filtering and alignment costs per query decrease as the workload size increases. For the filtering cost, with a larger workload size, the cost of the index join is amortized over a larger number of queries, resulting a reduction in the per query filtering cost. For the alignment cost, the reduction in per query cost come from the benefits of using the operating system cache. For the initial sequences in the batch, fetching a database sequence often results in an actual disk I/O. However, as the batch size increases the chances of a single database sequence matching more than one query sequence increases. Repeated accesses to the database sequence are likely to find the database sequence in the operating system cache and does not have to incur an expensive disk I/O. Consequently, as the batch size increases, the alignment costs per query also reduce.

It is also notable that the database caching effect further improves the relative performance of miBLAST. For instance, when we ran five workloads of 4000 queries consecutively, the relative speedup in processing the last workload increases up to 25.4 times, while the relative speedup of the first batch workload is 21.6 times. The reason for this improvement is that the later runs benefit from seeing data in the cache that has been retrieved by the processing of previous runs. miBLAST benefits more from this caching as it is more disk I/O intensive, as has a much larger data structure (the index). In fact, we can expect that if there was enough space to hold the entire miBLAST index in memory, its relative performance would be even greater.

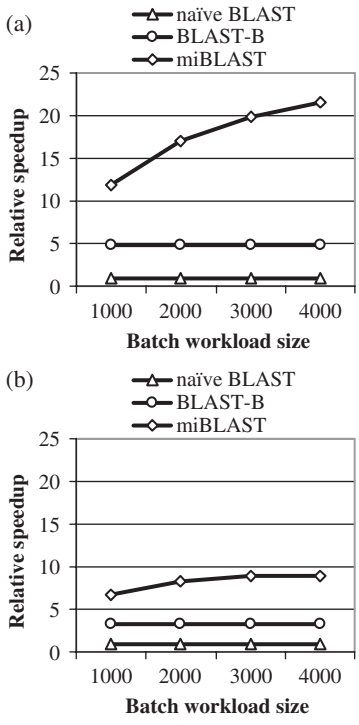

Figure 1. This graph shows the relative speedup of each method compared with naive BLAST, for various workload sizes using a word size of 11. (a) Affymetrix (25 bp). (b) Illumina (70 bp).

Table 2. The detailed execution time of miBLAST for the experimental results shown in Figure 1

Workload size	Filtering cost per query (25 bp)	Alignment cost per query (25 bp)	Filtering cost per query (70 bp)	Alignment cost per query (70 bp)
1000	0.44 (54%)	0.37 (46%)	0.46 (31%)	1.02 (69%)
2000	0.23 (42%)	0.33 (58%)	0.25 (21%)	0.96 (79%)
3000	0.17 (36%)	0.31 (64%)	0.18 (16%)	0.94 (84%)
4000	0.14 (31%)	0.30 (69%)	0.17 (15%)	0.94 (85%)

All times reported here are in s.

Effect of the BLAST word size parameter

In this section, we examine the effect of the query word size parameter, which is a commonly tuned BLAST parameter.

For this experiment, we use the same dataset and query workload as used in the previous experiment, but increase the word size gradually from 11 to 23. In each case, miBLAST used an index that had a word size of 11. The results of this experiment are shown in Figure 2. As seen from this figure, the performance of miBLAST improves significantly as the word size increases. For a word size of 23, miBLAST is 45.2 (25 bp) and 13.6 (70 bp) times faster than naive BLAST, and 7.5 (25 bp) and 2.2 (70 bp) times faster than BLAST-B.

To understand why a larger word sizes benefits miBLAST, we measured a metric called filtration ratio. Filtration ratio is

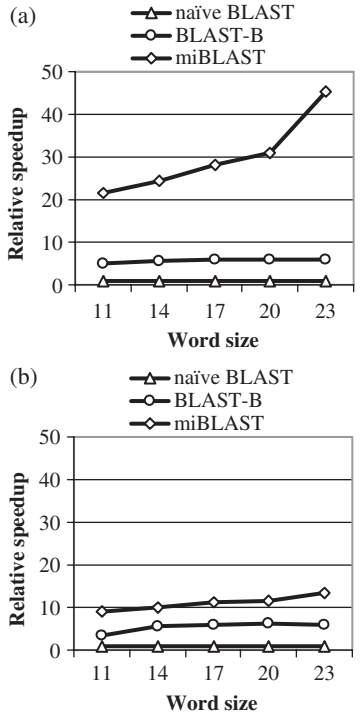

Figure 2. This graph shows the effect of the BLAST word size parameter on the query performance for each method, plotted as relative speedup over naive BLAST. The batch size used in this experiment is 4000 queries. miBLAST uses an index word size of 11 and uses a sliding-window filtering method for query word sizes between 14 and 23. (a) Affymetrix (25 bp). (b) Illumina (70 bp).

Table 3. The effect of the BLAST word size parameter on the filtration ratio and the alignment cost in miBLAST

Word size	Average filtration ratio	Alignment cost per query
11	0.54964%	0.30
14	0.03924%	0.13
17	0.02231%	0.11
20	0.02115%	0.09
23	0.02009%	0.08

The query set used for collecting this data is 4000 queries from the Affymetrix dataset. All times reported here are in s.

defined as the ratio of the number of sequences that are identified as potential results using the word hit over the total number of sequences in the database (7). The filtration ratio measures the proportion of the database that must to be examined to generate actual alignments. With miBLAST a lower filtration ratio leads to better performance as a smaller fraction of the database is searched during the alignment phase. As the word size increases, the probability of finding a word hit decreases exponentially, leading to a exponential decrease in the filtration ratio. This filtering behavior with respective to word size and its effect on the performance of miBLAST is shown in Table 3.

As shown in Figure 2, with miBLAST a larger word size does lead to a reduction in the the number of retrieved sequences. However, both naive BLAST and BLAST-B scan the entire database (during the word hit generation phase), so the lower filtration ratio does not reduce the number

Table 4. The average execution time per query for the results shown in Figure 2a, for a batch size of 4000

Word size	naive BLAST	BLAST-B	miBLAST
11	9.50	1.95	0.44
14	9.26	1.61	0.38
17	9.46	1.58	0.37
20	9.45	1.58	0.30
23	9.44	1.56	0.21

All times reported here are in s.

of database sequences that are retrieved. There is a reduction in the number of alignments that are computed, but both naive BLAST and BLAST-B spend most of their execution time in the word hit generation phase. Consequently, the overall reduction in execution time with increasing word size is very small for these two methods.

An astute reader may have noted that in Table 4 increasing the word size does not have a significant impact on the performance of naive BLAST and BLAST-B. While this effect may seem contrary to the intuition of improved performance for larger word sizes, it turns out that in the case of nucleotide sequence searches, increasing the word size does not have a significant impact on the performance of both naive BLAST and BLAST-B due to the way the database sequence representation database is packed into bytes and interactions of this packing with the processor word length. Increasing the BLAST word size parameter may have negligible effect on performances as the processor may still be doing equivalent work, because it is fetching and processing data in block sizes that are set by the underlying computer architecture. In fact, in some cases increasing the word size may actually result in a small decrease in performance (for example when the larger BLAST word parameter requires the processor to operate on a larger number of memory blocks). This effect of increasing word size has also been reported for WU-BLAST (http://blast.wustl.edu/blast/TOFLY.html).

Effect of query length

To measure the effect of query length on the performance of miBLAST, we ran the following experiments. We generated a number of query sets from the EST human database. Each query set contained 1000 queries of a fixed length, which was randomly picked from the EST database. We generated query sets with query lengths ranging from 16 to 512 bp and each query set is run against UniGene database. The results of this experiment (see Figure 3) show that the performance speedup of miBLAST decreases as query size increases. The reason for this behavior is that miBLAST's performance is highly dependent on filtration ratio, as it primarily speeds up the filtering component of BLAST searches. In general, a lower filtration ratio leads to better relative performance for miBLAST. As the query length increases, it is likely that the query will have more word hits with sequences in the database, increasing its filtration ratio, and resulting in a relative reduced performance for miBLAST.

We also note that the comparisons with BLAST-B in Figure 3 are based on the most optimal batch size for BLAST-B, which we picked by manually trying various batch sizes for each query set. For different workloads,

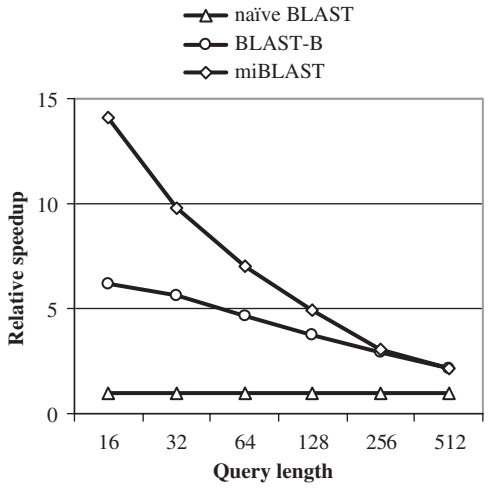

Figure 3. This graph shows the relative speedup to naive BLAST for various query lengths, using a word size of 11. Queries are drawn from the EST human dataset, and each batch has 1000 queries.

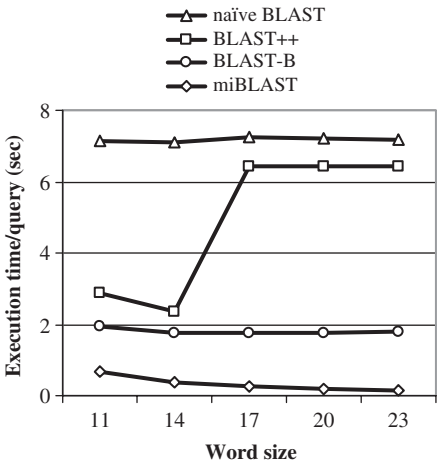

Figure 4. This graph shows the execution time of each method for various word sizes using a batch of 1000 queries from the Affymetrix probe set (25 bp).

the optimal batch size changes and to get the best performance using BLAST-B the user has to manually determine the optimal batch size. For example, the optimal batch sizes are ∼100, 50, 50, 25, 25 and 12 for queries of lengths 16, 32, 64, 128, 256 and 512, respectively. In contrast, with miBLAST there is no such manual tuning requirement. The optimal batch size can have a significant impact on the performance for BLAST-B; for example, with 256 bp queries, using a batch size 200 instead of 25 increases the total execution time by 50%.

Performance comparison with BLAST++

In this last experiment, we compare miBLAST with BLAST++. As noted earlier, the current version of BLAST++ cannot handle the size of the current Human UniGene dataset. Hence, for this experiment we used only half of the Human UniGene data set (for all the methods). The query set that we use in this experiment is the Affymetrix oligonucleotide sequences (25 bp). The BLAST++ configuration is similar to BLAST-B, and there is an optimal batch size, which in this case is about 200 queries in a batch. The results of this experiment are shown in Figure 4. As can be seen in this Figure, BLAST++ only outperforms the naive BLAST and is worse than miBLAST and BLAST-B in all cases. miBLAST is 5 and 48 times faster than BLAST++ at a word size of 11 and 23, respectively. Also, note that as the word size increases, the performance of BLAST++ generally degrades. The performance improvement for BLAST++ comes from sharing database sequence information for common words in the queries. However, when the word size is large, the number of such common words is reduced, and BLAST++ ends up accessing larger number of database sequences for each query sequence.

DISCUSSION

Index storage and construction costs

The index used in miBLAST requires $(8 \times S^m + 4 \times N)$ bytes, where S is the size of the alphabet for the symbols (typically 4 for nucleotide sequences), m is the index word length and N is the total number of symbols in the database. $8 \times S^m$ bytes are used for index header, and $4 \times N$ bytes are used for saving sequence ID information. However, since we do not save a duplicate sequence ID when the same word occurs more than once in the same sequence, the actual index size is significantly smaller than indicated by the above formula. For the human UniGene database containing 3.19 gigabases, using a word length 11, the index for the database is 11.94 GB. Only 32 MB of this space is used for the index header, and the remaining portion of this index space is used for storing the sequence ID information.

Constructing a q-gram index can be expensive for large databases. However, the index construction cost is a one time cost and this cost is amortized over all the batch workloads that use this index. For example, the index on the Human UniGene takes 38 min on our test machine. Assuming that we process the Affymetrix query set consisting of 247 965 query sequences, the index construction cost per query is 0.0092 s, and this cost can be reduced further when we have a larger batch workload or when the index is used to process multiple batch workloads.

Biological applications of miBLAST

In this section, we discuss the biological applications that can benefit from miBLAST. The characteristics of miBLAST make it immediately applicable when evaluating a large number of oligonucleotide probes, cDNA sequences or ESTs against databases of ESTs. Next, we elaborate on some applications with these characteristics.

One important application is the validation of a probe set, such as the Affymetrix probe set, against the UniGene database. The Affymetrix probe sets are searched against the most recent UniGene, and the search is often conducted periodically triggered by updates to the UniGene dataset. In this search, if one is only looking for labels, then a regular expression search is sufficient. However, the mismatches found in BLAST are also often important. For example, a step in evaluating the Affymetrix probes is an hybridization energy calculation for each probe that has sequence similarity with an EST.

Mismatches may not bind with the same efficiency as a perfect match, but knowing the number of mismatch probes and the hybridization energy helps determine the specificity of the individual probes (I. Lee, personal communication). This task requires a sensitive search tool. BLAST, which uses overlapping words in its search technique, when run with a small word size parameter results in a sensitive search that is suitable for this application. Since this task needs periodic evaluation of a large number of probes against a large EST database, miBLAST is a good alternative to BLAST.

A similar application is when designing new probe sets for DNA microarrays. miBLAST can be used to search new probe sets against the EST library of a species as a first pass for sensitivity. The search result of mismatch and perfect matches can be used to calculate hybridization energy for new microarray design. This task has been a major step in programs developed for probe design (13,14) and miBLAST can help speed up the basic computation task.

Another biological application is when using one animal model microarray against a similar species. While the creation of a new chip set for every species is technically feasible, the cost involved and skills necessary are not widespread. Until that time, using a similar species chip set will be an inexpensive solution. In this case, searching the chip probe set against the collection of ESTs or cDNA of the related species are often needed to validate the probes in the new species (15).

While the focus of the applications in the discussion above has been on microarray studies, miBLAST can also be used in other applications. As more individuals create siRNAs, to silence genes within cells, short nucleotide sequence searches against EST databases will increase to look for cross hybrid activities with other ESTs to narrow the specificity of the siRNAs. miBLAST can decrease the computation demands of this task.

We note that miBLAST improves the efficiency of sequence searches for short nucleotides against ESTs. If one is concerned with EST-to-genome or genome-to-genome alignment, other existing methods (5,16,17) are likely to be more practical for such tasks.

Finally, we note that our miBLAST implementation, like WU-BLAST (http://blast.wustl.edu), also allows the user to specify a scoring matrix, which can be used to model complex scoring models, such as discriminating between the scoring of transitions and transversions.

CONCLUSION

In this paper, we have presented miBLAST, a fast BLAST-like search algorithm for efficiently evaluating batch workloads, which consist of a large number of nucleotide query sequences that must be matched against a nucleotide sequence database. Current methods for evaluating such workloads essentially employ a 'nested-loops' paradigm in which each query sequence is individually evaluated using the BLAST search tool. This existing approach can be very expensive, especially for large batch sizes. Using a combination of *q*-gram indexes and an index join algorithm, miBLAST can dramatically speed up the evaluations of such workloads. miBLAST is particularly effective for workloads that consist of short queries, such as oligonucleotide probe sets.

The miBLAST search tool is implemented using the NCBI toolkit and employs the same statistical model and output format that is familiar to BLAST users. Consequently, we expect that existing BLAST users can make a seamless transition to miBLAST. The source code and executable for miBLAST are freely available.

AVAILABILITY

miBLAST is available as free open-source software. The software and instructions for installing the software can be downloaded from http://www.eecs.umich.edu/miblast/.

ACKNOWLEDGEMENTS

This work was supported in part by grants from the Michigan Economic Development Corporation (GR-238 and MTTC 05-095) and a gift from Microsoft. The authors would also like to thank Bob Thompson for sharing the 70mer probes set data with then, and Fan Meng for discussions on this topic. Finally, the authors also want to thank Warren Gish, Tom Madden and Jason Klivington for discussions on different BLAST implementations. Funding to pay the Open Access publication charges for this article was provided by the MTTC 05-095 grant.

Conflict of interest statement. None declared.

REFERENCES

1. Lachance,P.E. and Chaudhuri,A. (2004) Microaarry analysis of developmental plasticity in monkey primary visual cortex. *J. Neurochem.*, **88**, 1455–1469.
2. Uddin,M., Wildman,D.F., Liu,G., Xu,W., Johnson,R.W., Hof,P.R., Kapatos,G., Grossman,L.I. and Goodman,M. (2004) Sister grouping of chimpanzees and humans as revealed by genome-wide phylogentic analysis of brain gene expression profiles. *Proc. Natl Acad. Sci. USA*, **101**, 2957–2962.
3. Altschul,S.F., Gish,W., Miller,W., Myers,E.W. and Lipman,D.J. (1990) Basic local alignment search tool. *J. Mol. Biol.*, **215**, 403–410.
4. Altschul,S. and Gish,W. (1996) Local alignment statistics. *Meth. Enzymol*, **266**, 460–480.
5. Kent,W.J. (2002) BLAT-the BLAST-like alignment tool. *Genome Res.*, **12**, 656–664.
6. Ning,Z., Cox,A.J. and Millikin,J.C. (2001) SSAHA: A Fast Search Method for Large DNA Databases. *Genome Res.*, **11**, 1725–1729.
7. Burkhardt,S., Crauser,A., Ferragina,P., Lenhof,H.P., Rivals,E. and Vingron,M. (1999) *q*-gram based database searching using a suffix array QUASAR. *Proceeding of the third International Conference on Computational Molecular Biology*. Lyon, France, pp. 77–83.
8. Zhang,Z. (2000) A greedy algorithm for aligning DNA sequences. *J. Comp. Biol.*, **7**, 203–214.
9. Korf,I. and Gish,W. (2000) MPBLAST: improved BLAST performance with multiplexed queries. *Bioinformatics*, **16**, 1052–1053.
10. Wang,H., Ooi,B.C., Tan,K.L., Ong,T.H. and Zhou,L. (2003) BLAST++: BLASTing queries in batches. *Bioinformatics*, **19**, 2323–2324.
11. Darling,A.E., Carey,L. and Feng,W. (2003) The design, implementation and evaluation of mpiBLAST. *Proceeding of the Fourth International Conference on Linux Clusters*, San Jose, CA.
12. Navarro,G. and Yates,R.B. (1998) A practical q-gram index for text retrieval allowing errors. *CLEI Electronic Journal*, **1**, 1725–1729.
13. Rouillard,J., Zuker,M. and Culari,E. (2003) OligoArray 2.0: design of oligonucleoide probes for DNA microarray using a thermodynamic approach. *Nucleic Acids Res.*, **31**, 3057–3062.

14. Wright,M.A. and Church,G.M. (2002) An open-source oligomicroarray standard for human and mouse. *Nat. biotechnol*, **20**, 1082–1083.
15. Lee,I., Dombkowski,A. and Athey,B. (2004) Guildlines for incorporating non-perfectly matched oligonucleotides into target-specific hybridization probes for a DNA microarray. *Nucleic Acids Res*., **32**, 681–690.
16. Florea,L., Hartzell,G., Zhang,Z., Rubin,G.M. and Miller,W. (1998) A computer program for aligning a cDNA sequence with a genomic DNA sequence. *Genome Res*., **8**, 967–974.
17. Schwartz,S., Kent,W., Smit,A., Zhang,Z., Baertsch,R., Hardison,R., Haussler,D. and Miller,W. (2003) Human-mouse alignments with BLASTZ. *Genome Res*., **1**, 103–107.