

BLAST++: A Tool for BLASTing Queries in Batches

Hao Wang¹, Twee-Hee Ong¹, Beng Chin Ooi¹, Kian-Lee Tan^{1,2}

¹Department of Computer Science, National University of Singapore 3 Science Drive 2, Singapore 117543

²Genome Institute of Singapore, 1 Science Park Road, The Capricorn #05-01, Singapore 117528
E-mail: {wanghao,ongtweeh,ooibc,tankl}@comp.nus.edu.sg

Abstract

BLAST is the standard tool that molecular biologists use to search for sequence similarity in genomic (and protein) databases. It employs a brute force approach of comparing a query sequence against every database sequence – for each pair of the sequences to be matched, BLAST searches for short fixed-length word pairs (seeds) in the sequences and then extends them to higher-scoring regions. To search multiple queries, the basic approach is to run BLAST on each of the queries one at a time. This is clearly inefficient and fails to exploit common subsequences that the collection of queries may share. In this paper, we propose a new genome search tool, BLAST++, that allows multiple, say K , queries to be searched against a database concurrently. The design of BLAST++ is based on our observation that the seed searching step of BLAST is a bottleneck that consumes more than 80% of the total response time! BLAST++ essentially treats a collection of queries as a single virtual query so that the seed searching step needs to be performed only once for common subsequences. We implemented BLAST++ as an extension of the NCBI BLAST, and evaluated its performance. Our study shows that the results obtained by BLAST++ are *identical* to that obtained by executing BLAST on each of the K queries, but the single-process version of BLAST++ completes the processing in a much shorter time, about only 25% of the original single-process version of NCBI BLAST.

keywords: BLAST, batch query processing, seed, common subsequence, cluster.

1 Introduction

Batching queries for processing is becoming a common task of molecular biologists. For example, to investigate the orthologs between two species, one needs to search a database of sequences of one species against that of the other. As another example, there is an increasing need to mine sequence databases for useful information. This typically requires all pairwise similarity between all the sequences to be computed. Even for queries that are issued by different users, there is a need to batch them for processing. For example, it has been reported that the number of queries directed at public databases (e.g. PDB (*PDB Database* 2002), PIR (*PIR Database* 2002), GenBank (*Genbank Database* 2002)) are very large (over 40,000 queries per day (Williams & Zobel 2002)) and the user numbers and query rates are growing. As such, it makes sense to batch several queries from different users to optimize resources and throughput.

BLAST (Altschul, Gish, Miller, Myers & Lipman 1990) is the standard tool that molecular biologists use to search for sequence similarity in genomic (and protein) databases. It employs a brute force approach of comparing a query sequence against every database sequence – for each pair of sequences to be matched, BLAST searches for short fixed-length word pairs (seeds) in the sequences and then extends them to higher-scoring regions. To evaluate a batch of queries, the default strategy is to run BLAST on each of the queries one at a time. This is clearly inefficient and fails to exploit any common subsequences that may exist among the queries to share their computation. Thus, there is an urgent need to design efficient schemes to search a collection of queries against a genomic or protein database.

A promising approach is to search a batch of queries collectively to exploit sharing of results on common subsequences. One such program, MEGABLAST (Zhang, Schwartz, Wagner & Miller 2000), is available from NCBI. However, MEGABLAST employs a greedy sequence alignment algorithm that is optimized for aligning sequences that differ slightly as a result of sequencing or other similar “errors”. While this allows it to employ longer seeds than BLAST, it is less effective for more general similarity search. In other words, answers returned from MEGABLAST may miss out answers produced by BLAST.

In this paper, we present BLAST++, a new search engine that allows multiple, say K , queries to be searched against a database concurrently. The design of BLAST++ is based on our observation that the seed searching step of BLAST is a bottleneck that consumes more than 80% of the total response time! BLAST++ essentially treats a collection of queries as a single virtual query so that the seed searching step needs to be performed only once for common subsequences. We implemented BLAST++ as an extension of the NCBI BLAST (*NCBI-Blast readme* 2002), and report the results of our performance study in this paper.

We believe that BLAST++ is practical and useful for the community for the following reasons. First, because it is based on BLAST, the results obtained by BLAST++ are identical to that obtained by executing BLAST on each of the K queries. This is important and will facilitate the acceptance of BLAST++. In fact, if the batch contains only one query, then BLAST++ behaves essentially like BLAST. Second, BLAST++ is much more efficient and can complete the processing of the K queries in a much shorter time than BLAST. This translates to higher throughput, better resource utilization and

better user satisfaction!

The rest of this paper is organized as follows. In Section 2, we report an experimental study on BLAST to study the breakdown of its processing cost. Section 3 presents our BLAST++ algorithm. In Section 4, we present an experimental study to evaluate BLAST++ and report our findings. Section 5 reviews some related work, and finally, we conclude in Section 6.

2 A Peek into BLAST

In this section, we shall peek into the internals of BLAST to study the breakdown of its processing cost. This will allow us to identify bottleneck steps that we can seek to improve.

BLAST (Altschul et al. 1990) operates in four steps. First, it creates query words of length k (k -tuples). For protein query sequence, BLAST also enumerates closely related tuples from every k -tuples using a user-given scoring matrix (e.g., PAM 120, BLOSUM 62, etc.).

In the second step, BLAST scans through every database sequence to see if it contains a k -tuple that can pair with a query k -tuple to produce a word pair with a score greater than or equal to a predetermined threshold T . These word pairs are referred to as *seeds*. The sequential scan of the complete database sequences employs a finite state machine algorithm.

In the seed extension step, each seed is extended to possibly longer similar segment pairs. The final step is to produce alignments between the data sequences that are most similar to the query sequence.

In BLAST-2 (Altschul, Madden, Schaeffer, J.Zhang, Zhang, Miller & Lipman 1997), seed extension is restricted to sequences that share at least two non-overlapping seeds, with the same relative offset and within a certain distance of one another. This stricter criterion trades sensitivity for speed. BLAST-2 also tries to produce gapped alignment from multiple local maximum segment pairs (MSPs) for each sequence, whereas the original BLAST program often finds several alignments involving a single database sequence.

BLAST permits a tradeoff between speed and sensitivity, by adjusting the settings for the threshold T and the length of k -tuple. A higher T or k value yields greater speed (by avoiding spurious short seeds arising by chance in background sequences that cause unnecessary comparisons). However, it may increase the probability of missing weak (i.e., less sensitive) similarities. For DNA queries, the default settings are $T = 0$ and $k = 11$. For protein queries, k has a default value of 3.

Figure 1 illustrates how BLAST works. Here, $k = 11$ and $T = 0$, and the query sequence has a length of 17 base pairs leading to 7 query windows (step 1). In step 2, four of these windows are identified as having matching segments in the data sequence. Step 3 then extends the first of these seeds to give the alignment shown in step 4.

BLAST has been very successful - it is not only widely accepted by molecular biologists, it has remained one of the most efficient search tools (despite being an exhaustive search tool!). As such,

Sample Query and Database sequence	
Query	GCAACGGGCAATATGTG
Data	ATTGAAACGGGCAATATGTC
Settings	$k = 11$ (default for BLAST), $T = 0$
Step 1: Create query windows	GCAACGGGCAA CAACGGGCAAT AACGGGCAATA ACGGGCAATAT CGGGCAATATG GGGCAATATGT GGCAATATGTG
Step 2: Search seeds	AACGGGCAATA ACGGGCAATAT CGGGCAATATG GGGCAATATGT
Step 3: Extend seed	AACGGGCAATATGT
Step 4: Align sequences	---GCAACGGGCAATATGTG ATTGAAACGGGCAATATGTC

Figure 1: Illustration of BLAST search steps.

we proceed to study the various steps in BLAST to determine their strengths and weaknesses. By identifying the bottlenecks of BLAST, we hope that efficient alternatives can be designed to replace these while allowing us to preserve its strengths. This will also allow us to produce new BLAST-like sequence search tools that are likely to be more readily accepted by biologists.

We downloaded the NCBI-BLAST (*NCBI-Blast readme* 2002) source codes and recompiled the whole BLAST package with a profiler (*GNU profiler* 2002) option. At run-time, the recompiled executables generate additional information which can be analyzed with a profiler (*gprof* (*GNU profiler* 2002)) to produce statistics on the time spent on each function. These statistics are grouped to produce the time spent on database retrieval, k -tuple matching, and seed extension. Table 1 shows the main functions in NCBI-BLAST source codes.

We ran BLAST with default settings on a GenBank DNA database of Release 125 (*Genbank Database* 2002) containing 487,465 number of sequences in a total of 1.7GB, and on a protein database PIR (*PIR Database* 2002) database of release 68 containing 216,912 number of sequences in a total of 75MB.

For each query length, we generated 100 random queries and obtained the average running time. The results, plotted in Figure 2, show that 80% or more of the time was spent in the seed searching step alone. In other words, the dominant cost is the CPU computation for seed searching and the I/O cost is relatively insignificant. On hindsight, this is expected since the seed searching step involves extracting the k -tuples from each database sequence, and the number of k -tuples is close to the length of each sequence. This observation calls for new mechanisms to be designed to minimize the number of seed searching operations without sacrificing on the quality of the seeds generated. We shall see shortly how this can be easily realized when multiple queries are being batched for processing.

Activities	Program Functions	
	DNA	Protein
Read database function	readdb_get_sequence	readdb_get_sequence
Word match	BlastNtWordFinder	BlastWordFinder_mh_contig
Word extension	BlastNtWordExtend	BlastWordExtend_prelim
Produce alignments	PerformNtGappedAlignment	PerformGappedAlignment

Table 1: Major functions in BLAST for DNA and protein sequence comparison (with default settings).

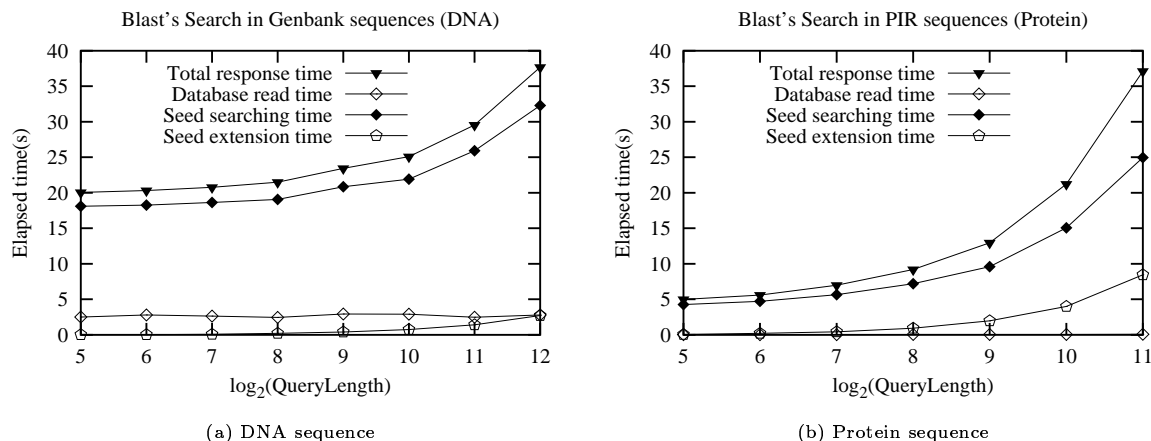


Figure 2: Breakdown of BLAST's search time.

3 BLASTing Multiple Queries

In this section, we shall present BLAST++, a tool designed to search multiple sequences concurrently. In designing BLAST++, we have two main objectives. The first goal, albeit a non-technical one, is to have a tool that will be readily accepted and used by molecular biologists. There are many good sequence search engines that have been proposed in the literature but these have not been successful to penetrate to the life sciences community. To realize this goal, we sought to extend BLAST instead of redesigning a new strategy. Hence, the name of our tool BLAST++. The second goal is to improve on BLAST either in terms of sensitivity or efficiency.

Figure 3 shows the algorithmic description of BLAST++. As shown, the algorithm is structured in the same manner as BLAST and comprises four similar phases. In the k -tuple extraction phase (lines 1-4), we first create a *virtual query*, consisting of all queries. A sliding window of size k is then applied to the virtual query to extract the k -tuples (Of course, care is taken to note the boundaries between two queries). Since the same k -tuple may be repeated multiple times within a query and may appear in multiple queries, we also maintain for each k -tuple a list of (query id, list of offsets) pairs. If we are dealing with protein sequences, then we will also generate k -tuples that are similar based on the similarity matrix used.

In the seed matching phase (lines 5-7), the database is scanned to identify seeds. Seeds that are promising are stored in a lookup table as a list of (query id, list of offsets within query, database sequence id, list of offsets within database sequence) tuples. We note that this step is the key step to reducing the computation cost of the algorithm. Recall that the seed matching step is the most costly step. Here, the proposed scheme essentially exploits

common k -tuples that appear in multiple queries so that the extraction of k -tuples from the database sequences needs to be done only once (as compared to N times for N queries).

Finally, in the seed extension phase and alignment phase (lines 8-14), we extend the seeds and align each query. We reuse BLAST's seed extension and alignment schemes in our implementation. We note that there are two approaches to this step. The first requires only one scan of the database, and for each database sequence, seed extensions are performed for all query sequences. The second, which is the adopted approach, is to scan the database once for each query. While the first method is more efficient, it requires more memory since we need to keep track of all answers for all queries. The second approach, though more costly, has the advantages of smaller memory requirement (only for one query at a time) and allows answers of each query to be returned as soon as they are obtained.

Figure 4 illustrates the steps of BLAST++. Here, we assume that we have a set of 22 queries. In step 1 (Figure 4(a)), we have identified that the first k -tuple appears in query sequence 1 at offsets 3, 22 and 39, and in query sequence 2 at offsets 32 and 48. In step 2 (Figure 4(b)), we also found that data sequence 1 contains the first k -tuple at offset 45. Thus, we have the entries [Q1,(3,22,39),D1,(45)] in the lookup table. The other entries are derived in a similar way. Finally, in the final two steps, the seeds are extended for each query one at a time and alignments between the database sequences and the query sequence that met the user specifications are returned to users.

We note that the queries in a batch is a critical factor. First, if the batch size is too large, the virtual query may become too large for the system to handle (because of insufficient memory). In this case, we will need to split the batch into multiple batches.

Algorithm BLAST++

Input: A batch of N queries

```
//  $k$ -tuples Extraction Phase
1. Create a "virtual" query concatenating all  $N$  queries
2. Extract all  $k$ -tuples from the "virtual" query
3. If queries are protein sequences
4. Expand the  $k$ -tuples with the scoring matrix
// Seed Matching Phase
5. Scan the database to find matches of  $k$ -tuples
6. If match found
7. Store match of  $k$ -tuple with corresponding position information in a lookup table
// Seed Extension and Alignment Phases
8.  $i \leftarrow 0$ 
9. repeat
10.  $i++$ 
11. Retrieve matches of query  $i$  from the lookup table for extension
12. Align the highest scoring extensions
13. Return all the alignments found
14. until all  $N$  queries have been examined (i.e.,  $N = i$ )
```

Figure 3: BLAST++.

Second, intuitively, one expects the performance gain to be less significant if the queries do not share many common subsequences. Thus, for a large number of queries in a batch, we can split it into smaller number of batches containing similar queries. In both cases, splitting into smaller batches can be easily done using a clustering algorithm, and for each such smaller batch, BLAST++ can be employed.

4 A Performance Study

We have implemented BLAST++ by extending NCBI's Standard BLAST (source code obtained from NCBI). The current implementation only considers searching DNA sequences in the forward direction, disabling the filtering of low complexity region of query sequences. To verify its effectiveness and evaluate its efficiency, we conducted some performance study. We report representative experiments on two datasets here. The first dataset is the 1.62 GB Genbank database. There are 487,490 sequences and the sequences range from 10 to 75,7191 bp. The second dataset is the entire 2.89 GB human genome sequence. In this set, there are 24,587 sequences ranging from 22,856,748 to 242,029,397 bp.

We use the *formatdb* program in BLAST package to encode the files into binary files in BLAST-specific format. For the purpose of evaluating response time, the search results of BLAST are sent to the null device (`/dev/null`) to avoid the unnecessary cost in writing outputs to files.

All the queries used for the experiments are randomly generated with the specified lengths. We note that all the queries in a batch are of the same length. This is done solely for ease of presentation of the results. BLAST++ certainly works even if queries within a batch are of variable lengths. Our performance study on such batches (not reported here) have shown that the relative performance between BLAST++ and BLAST remains largely the same as that reported in this paper.

All experiments were conducted on a SUN E-450 machine, with two 500 MHz CPU and 4 GB of main memory. The command line for the program is as follows: `blastall -p blastn -d dataFile -i queryFile -o outputFile -F F -J T -S 1`. The results (scores

and alignment) obtained using BLAST++ were compared to that retrieved by NCBI-BLAST. Unless otherwise stated, we used the default settings recommended by BLAST for both systems. In these cases, the results, as expected, were found to be identical and so we only present the efficiency of BLAST++ as compared to NCBI-BLAST for batched queries.

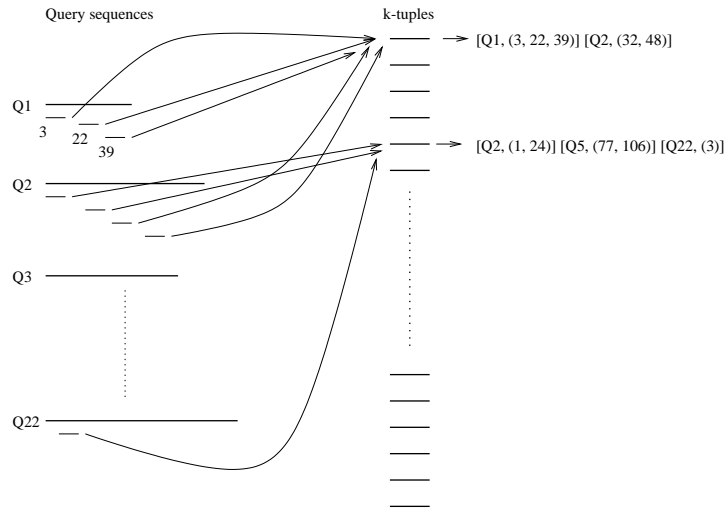
4.1 Effect of Query Length

In the first experiment, we study the effect of query length on the relative performance of the two schemes. We fix the number of queries to be processed collectively to 10. Figure 5(a) and (b) show the time taken by BLAST and BLAST++ in searching the two datasets respectively.

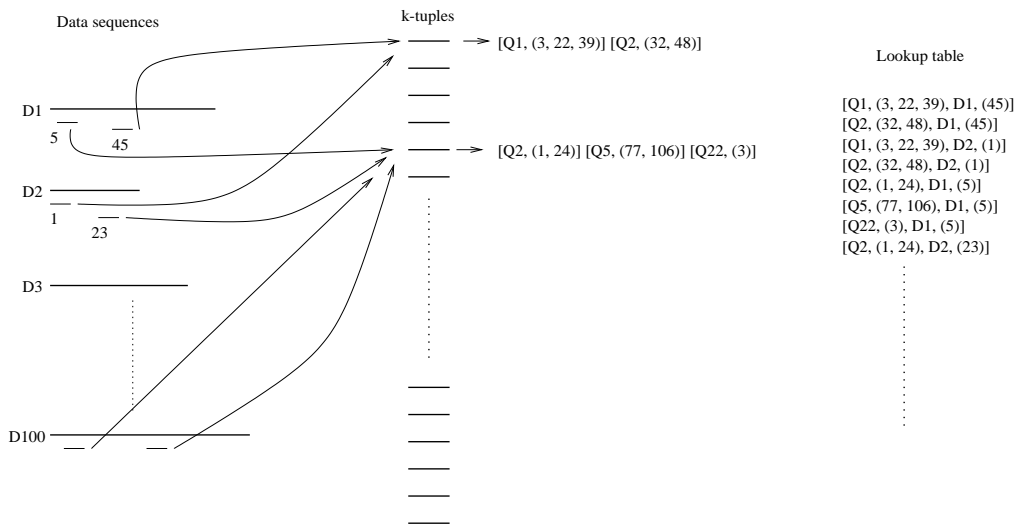
First, we observe that BLAST++ is able to achieve a gain of at least 50% in efficiency as compared to BLAST for both datasets. In fact, for short queries, BLAST++ takes less than 20% of the time required for BLAST to complete.

Second, we note that for queries longer than 512 bp, the rate of increase in time taken for BLAST++ increases more rapidly as compared to BLAST. This is due to the larger number of k -tuples matches in the queries and the datasets. The increase in the k -tuples matches results in more processing time to store and retrieve the matches. In addition, more verifications for a good alignment have to be performed in the alignment phase.

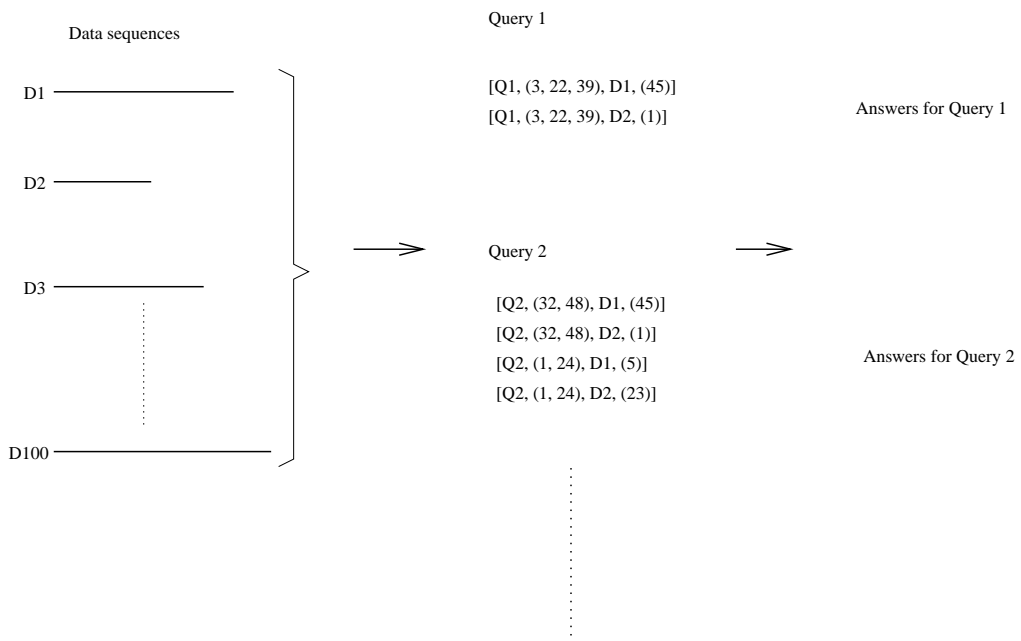
Third, we observe that BLAST++ gains more in searching the Human Genome dataset than in the Genbank dataset. We note that the sequences in the Human Genome dataset are very much longer than those in the Genbank dataset. Recall that the bottleneck of BLAST is in the seed searching phase which requires examining the entire sequence. Since BLAST++ is designed to minimize the number of seed searching, its benefit becomes more significant for long sequences. As we expect most of the biological research to be performed on Human Genome dataset, we believe BLAST++ would be a practical and useful tool.



(a) Step 1: Extraction of k-tuples from multiple queries



(b) Step 2: Seed matching



(c) Steps 3 and 4: Seed extension and sequence alignment

Figure 4: Illustration of BLAST++.

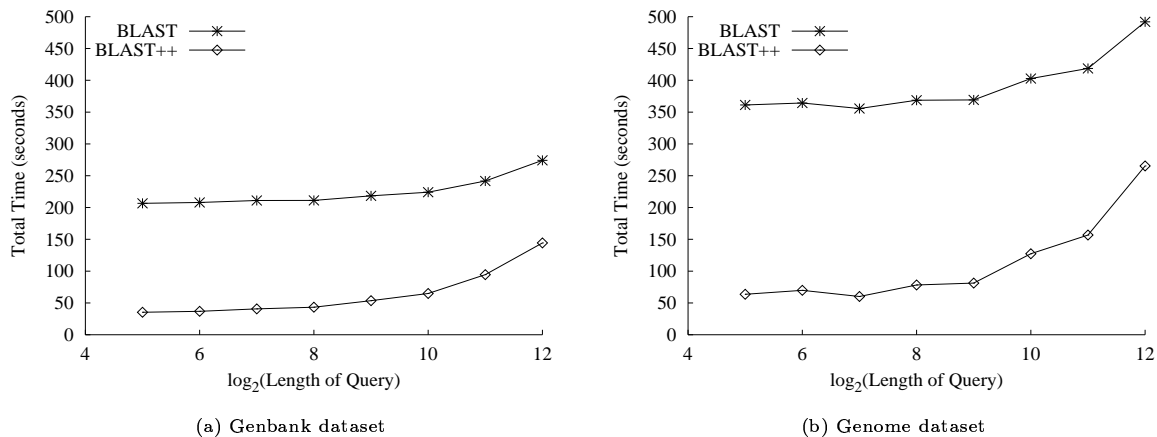


Figure 5: Effect of query length (batch size = 10).

4.2 Effect of Batch Size

In this section, we vary the batch size (i.e., number of queries to be batched) from 10 to 300. Here, we fix the length of each query to 128 bp. Figure 6(a) and (b) show the results of the experiment.

First, we note that the total time of both BLAST++ and BLAST increases (almost) linearly with the batch size. However, BLAST++ grows at a much slower rate. This is because of the savings in computation of not repeatedly scanning the database, and locating the matches of the k -tuples for each query. Again, this demonstrated the superiority of BLAST++ over BLAST.

Second, as in the earlier experiment, the gain in BLAST++ over BLAST is more significant for the Human Genome dataset than the Genbank dataset for the same reasons mentioned earlier.

4.3 Effect of Batching

In the earlier experiments, regardless of the number of queries, we have considered them as a single batch. Intuitively, given N queries, we can potentially group them into j ($1 \leq j \leq N$) batches such that $\sum_{i=1}^j n_i = N$ where n_i is the number of queries in the i th batch. Now when $j = 1$, we have all queries in a single batch (which is what we have done in the earlier experiments). When $j = N$, we have one query per batch which essentially corresponds to BLAST. In this experiment, we would like to study whether there is a certain j value that is optimal. For simplicity, we organize the batches such that they are of the same size, i.e., they contain the same number of queries. The query length is set to 128 bp. We conducted two experiments on the Human Genome dataset. The results are shown in Figure 7.

In the first experiment, we vary the number of batches while keeping the number of queries constant. Here, we have set the number of queries to 300. For simplicity, we randomly organize the queries into the various batch. Figure 7(a) shows the results of the experiment. It is interesting to note that the optimal j value is 1! On hindsight, this is not surprising. Consider 2 batches of queries. Let the number of distinct k -tuples that matches the database sequences be p and q respectively. In total, there are $p + q$ seed searching operations. On the other hand, when

the two batches are combined, it is highly possible that there are significant overlap between the p and q k -tuples. Thus, the seed searching step is significantly reduced. Now, as the batch size increases, the overlap also increases. Thus, batching more queries benefits more. Of course, there is the overhead of having to split the answers to the respective queries in a batch, but this overhead is less significant (since the seed searching step is the bottleneck).

We note that the above results hold because the batches are randomly produced. If the batches are “distinct”, we see a slightly different picture. In this experiment, we fix the number of batches to 2, and vary the size of the batches. We generate the batches as follows. We choose two cluster centroids (a pre-assigned query string), and a fix cluster radius (which is set to 2 in this experiment). We vary the number of queries in each cluster by randomly generating query strings which have edit distance no greater than the given radius to the cluster centroid. We shall refer to this as the clustering scheme. We also produce two batches of queries obtained using the same query set generated by the clustering scheme but randomizing them. This scheme is referred to as the randomizing scheme. We ignore the overhead of clustering time in our results.

We deployed three strategies: BLAST++(2,C) that runs the two batches generated by the clustering scheme; BLAST++(2,R) that runs two batches generated by the randomizing scheme; and BLAST++(1) that combines the two batches into one. Figure 7(b) shows the results. The results show that BLAST++(1) is always superior over BLAST++(2,R). This result is consistent with the results in the first experiment. However, we note that BLAST++(2,C) can outperform BLAST++(1) when the cluster size is large (starting at 200 in this experiment)! This is because the two query sets are quite distinct, and hence merging them into a large set does not benefit much. The clustering effect does not improve for small clusters due to the high initial setup cost.

5 Related Work

In this section, we briefly review some related work on sequence searching and batch sequence searching.

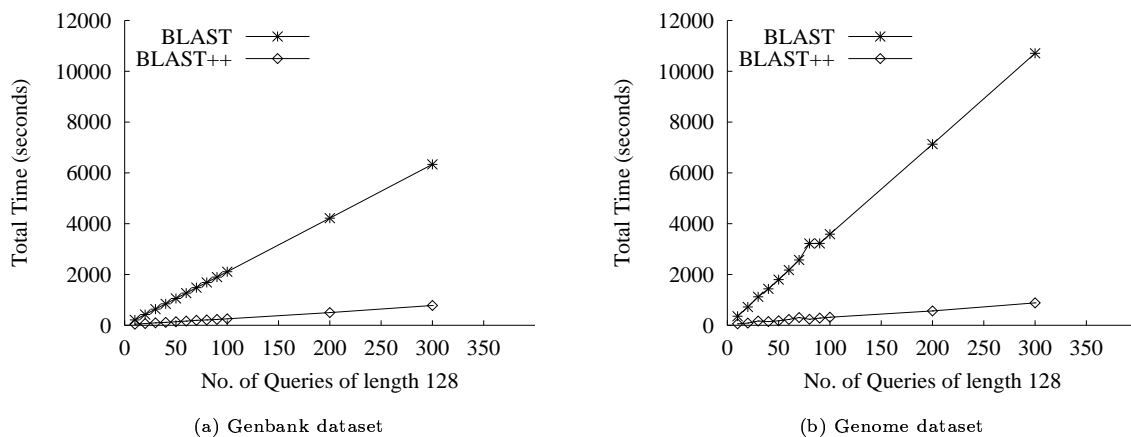


Figure 6: Effect of batch size (length of queries = 128 bp).

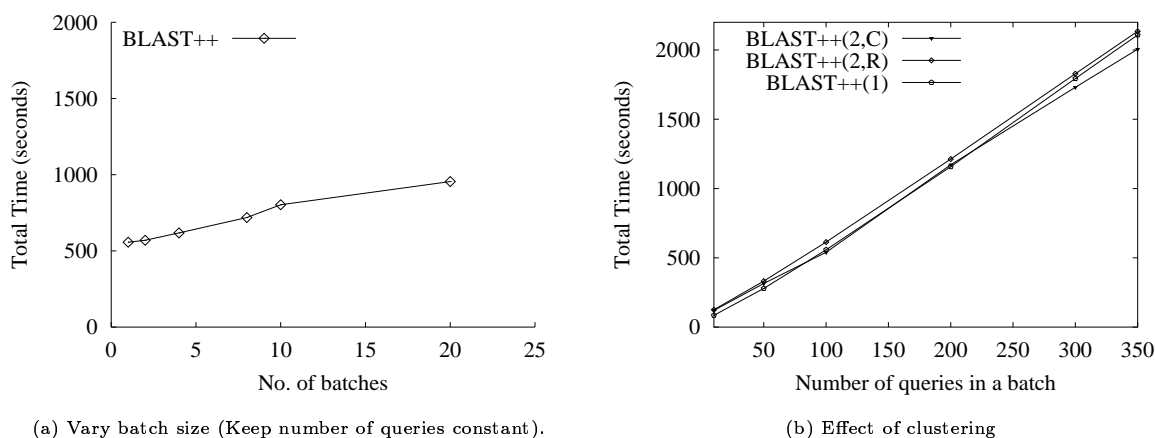


Figure 7: Effect of clustering (query length = 128 bp).

5.1 Sequence Searching Algorithms

As described earlier, BLAST (Altschul et al. 1990) is an exhaustive search tool that checks all the entries in the database to formulate an answer set to a query; other similar tools include the early works such as SSearch (States, Gish & Altschul 1991) and FastA (Pearson & Lipman 1988), and recent works such as PatternHunter (Ma, Tromp & Li 2002) and LSH-ALL-PAIRS (Buhler 2001).

LSH-ALL-PAIRS (Buhler 2001) finds ungapped local alignments in genome sequences. Similar to BLAST, it seeks only alignments that can be found by extending a fixed-length seed. The key idea of the algorithm is to look for much longer seeds allowing substitutions, unlike Blast that looks for exact match of short seeds. A randomized search technique called Locality-Sensitive Hashing (LSH) is used to generate seeds. Given a d -length string, a LSH function concatenates characters from k random and distinct positions of the string to form a length- k string called an LSH value. Given a LSH function, if two d -length strings have the same LSH value, then they are considered the same. False negatives occur when two similar d -length strings have different LSH value if the LSH function samples any position at which the strings disagree. False negatives are minimized by repeating the hashing with multiple independent random hash functions.

PatternHunter (Ma et al. 2002) is another BLAST-like genome search tool claiming to run much faster than BLAST. It speeds up the search process by generating longer consecutive seeds allowing masking off some positions in the seed. The longer seed increases the speed while masking some positions off in the seed allows inexact match of seeds and thus makes the algorithm sensitive (less likely to miss some short local similarity). PatternHunter accepts a given masking filter. By default it chooses an 18-bit long mask filter 111010010101111 that has 11 ones for DNA homology search. The algorithm first looks for seed of the same length as the length of filter. Seed candidates must match at all the positions corresponding to 1's in the mask filter. All mismatches at the positions corresponding to 0's in the mask filter are ignored. From the seeds found, PatternHunter performs hits generation and extension and outputs an optimal local alignment. The effectiveness of PatternHunter is restricted to static datasets where the "magic" mask filter can be determined.

The alternative to exhaustive search is to use an index-based approach. We review two such schemes here. RAMdb (Rapid Access Motif database) (Fondrat & Dessen 1995) is a system for finding short patterns (called *motifs*) in genome databases. Each genome sequence is indexed by its constituent

overlapping intervals in a hash table structure. For each interval, an associated list of sequence numbers and offsets is stored. This allows a quick lookup of any sequences that match a query sequence. A long query sequence is split into shorter non-overlapping motifs that are used to query the database. RAMdb is best suited for query motifs whose length is equal to or slightly longer than the indexed interval length, and has been shown to produce up to a 800-fold speedup in search time over exhaustive approximate pattern matching schemes. It requires a large index that is twice the size of the original flat-file database (including the textual descriptions) and suffers from a lack of special-purpose ranking schemes for identifying initial match regions. In addition, the non-overlapping interval of query motifs can lead to false dismissals.

CAFE (Williams & Zobel 2002) is based on a partitioned search approach, where a coarse search using an inverted index is used to rank sequences by similarity to a query sequence, and a subsequent fine search is used to locally align only a subset of database sequences with the query. The CAFE index consists of three components: a search structure, which contains the index terms or distinct intervals, that is, fixed-length overlapping subsequences from the collection being indexed; inverted lists, which are a carefully compressed list of ordinal sequence numbers, where each list is an index of sequences containing a particular interval; and a mapping table that maps ordinal sequence numbers to the physical location of sequence data on disk. A compression scheme is employed to try to make the index size more manageable. CAFE evaluates a query by representing it as a set of intervals, retrieving the list for each interval, then using a ranking structure to store a similarity score of each database sequence to the query. The authors noted that although it is more computationally efficient than the exhaustive methods, "exhaustive systems generally have better retrieval effectiveness".

5.2 Batched sequence searching

It is interesting to note that NCBI does not provide a BLAST version that optimizes batch processing. However, as noted in the introduction, a MEGABLAST (Zhang et al. 2000) tool is available from NCBI that employs a greedy sequence alignment algorithm instead. Unfortunately, MEGABLAST's answers may be different from that produced by BLAST and hence limited its wide applicability.

Another batch-oriented sequence alignment scheme is MPBLAST (Multiplex BLAST) (Korf & Gish 2000). MPBLAST is implemented as a software layer on top of Washington University BLAST. A preprocessor concatenates numerous short sequences into relatively fewer long sequences. These multiplex sequences are then used as queries in the BLAST searches. A postprocessor parses and deconvolves the resultant BLAST reports, including conversion of multiplex query coordinates back to their component sequence origins. Our BLAST++ is different from MPBLAST in two ways. First, BLAST++ is tightly coupled to BLAST, while MPBLAST is not. As such, BLAST++ can exploit BLAST more effectively. Second, MPBLAST's approach impacts the estimation of statistical significance of the results, and the *E*- and *P*-values are different from that had individual sequences been aligned. As such, there is no guarantee that the resultant sensitivities under MPBLAST is comparable to that BLASTing

each sequence one at a time. BLAST++, as already noted, generates exactly the same answers as BLAST.

A batch-based service named "Batch Blast" (Center 2002) provides batch processing capability. However, it merely takes a batch of queries and runs them in parallel in a cluster of machines. Its selling point is the faster speed (follows naturally from exploiting parallelism) and more output functions that can provide more information than that provided by standard BLAST (e.g., a graphical alignment of High Scoring Segment Pairs (HSPs), more complete descriptions of subject sequences, and links to the Genbank entries of the matched subject sequences). However, the context of study is different as we focus on running a single-process version of BLAST in a single machine.

Yet another parallel BLAST algorithm is NBLAST (Dumontier & Hogue 2002). NBLAST is a cluster variant of BLAST that allows different nodes in the cluster to be performing sequence alignment on different subset of sequences at the same time. It employs a partitioning strategy that can distribute the workload across all nodes equally. However, within each node, each sequence is aligned one at a time.

Our work is clearly distinct from these works where we focus on improving the performance in BLAST in a single process for a batch of queries. To our knowledge, there is no other reported work on optimizing the processing of a set of sequence matching queries.

6 Conclusion

In this paper, we have presented BLAST++, a tool to search a set of queries concurrently. BLAST++ is able to produce the same set of answers as BLAST (given the same settings), and yet able to achieve significant savings in computation cost as compared to BLAST. We believe BLAST++ is also able to achieve similar results for protein searches. We are currently working in this direction. Our current implementation of BLAST++ is single-threaded. BLAST++ can be further extended to be multiple-threaded to further boost its speed.

Availability

The source code would be made available upon request (will be available for public downloading at <http://xena1.ddns.comp.nus.edu.sg/~genesis> at a later date).

Acknowledgements

This project is partially supported by a university research grant on the GENESIS project.

References

- Altschul, S. F., Gish, W., Miller, W., Myers, E. W. & Lipman, D. J. (1990), 'A basic local alignment search tool', *Journal of Molecular Biology* **215**, 403–410.
- Altschul, S. F., Madden, T. L., Schaeffer, A. A., J.Zhang, Zhang, Z., Miller, W. & Lipman, D. J.

- (1997), 'Gapped blast and psi-blast: a new generation of protein database search programs', *Nucleic Acids Res.* **25**, 3389–3402.
- Buhler, J. (2001), 'Efficient large-scale sequence comparison by locality-sensitive hashing', *Bioinformatics* **17**(5), 419–428.
- Center, S.-W. B. (2002), Batch blast service, in 'http://www.nbif.org/products/bioinfo/bioinfo.php'.
- Dumontier, M. & Hogue, C. (2002), 'NBLAST: A cluster variant of BLAST for nxn comparisons', *BMC Bioinformatics* **3**(13).
- Fondrat, C. & Dessen, P. (1995), 'A rapid access motif database (ramdb) with a search algorithm for the retrieval patterns in nucleic acids or protein databanks', *Computer Applications in the Biosciences* **11**(3), 273–279.
- Genbank Database* (2002), in 'ftp://ftp.ncbi.nih.gov/genbank/'.
- GNU profiler* (2002), in 'http://www.gnu.org/manual/gprof-2.9.1/gprof.html'.
- Korf, I. & Gish, W. (2000), 'MPBLAST: improved BLAST performance with multiplexed queries', *Bioinformatics* **16**(11), 1052–1053.
- Ma, B., Tromp, J. & Li, M. (2002), 'Pattern-hunter: faster and more sensitive homology search', *Bioinformatics* **18**(3), 440–445.
- NCBI-Blast readme* (2002), in 'ftp://ncbi.nlm.nih.gov/blast/db/README'.
- PDB Database* (2002), in 'http://www.rcsb.org/pdb/'.
- Pearson, W. & Lipman, D. (1988), Improved tools for biological sequence comparison, in 'Proceedings Natl. Acad. Sci. USA Vol. 85', pp. 2444–2448.
- PIR Database* (2002), in 'ftp://nbrfa.georgetown.edu/pir_databases/'.
- States, D. J., Gish, W. & Altschul, S. F. (1991), 'Improved sensitivity of nucleic acid database searches using application-specific scoring matrices', *Methods: A Companion to Methods in Enzymology* **3**(1), 66–70.
- Williams, H. E. & Zobel, J. (2002), 'Indexing and retrieval for genomic databases', *IEEE Transactions on Knowledge and Data Engineering* **14**(1), 63–78.
- Zhang, Z., Schwartz, S., Wagner, L. & Miller, W. (2000), 'A greedy algorithm for aligning dna sequences', *Journal of Computational Biology* **7**(1-2), 203–214.