The Developer's Guide to BLAST

Jason Papadopoulos

Introduction

This document is intended to explain, clarify and document the source code to NCBI's BLAST suite of sequence alignment algorithms. It is intended for developers who want to understand, modify or debug the BLAST source code. The major goals are as follows:

- Attempt to document the "institutional memory" of the developers here at NCBI. In many cases, the algorithms used within BLAST, and the design decisions embodied in the code, have never been documented.

- Reduce the learning curve required to become familiar with the BLAST code

- Attempt to explain the performance characteristics of the BLAST algorithm, and to document the performance optimizations present in the code.

This document will be necessarily incomplete, as BLAST is still under constant development. At best it is an aid to following a moving target. The scope of the document is limited to the 'new' BLAST engine; this is a wholesale rewrite of the original BLAST code that incorporates current algorithmic ideas for similarity searching, major improvements in readability, much more thorough documentation, and a more consistent interface for applications to use. Reference to 'the BLAST engine' or 'the BLAST code' will by default refer to this new engine; statements that apply to the original BLAST engine will be specifically labeled as such.

Introduction to Similarity Searching

This section is a hideously abbreviated introduction to similarity searching, a subject whose nuances are far beyond the scope of this document. If you already know about this subject, just skip to the next section.

BLAST is designed to match up biological sequences to other biological sequences. The assumption underlying algorithms like BLAST is that biological sequences which 'look the same' perform a similar biological function and may be related through evolution. This is not always true, but is true often enough to be useful to molecular biologists.

A 'sequence' in this context is a complex organic molecule, composed of chains of atoms. Many of the biological properties of such molecules are determined by their three-dimensional shape, but comparing molecules through their shape is an inexact science. It is much simpler to assume that a molecule is a linear chain of atoms and to compare two such molecules without regard to their three-dimensional structure. Once this structure is removed, molecules can be treated as a chain of acids, and the chains can be compared.

Molecular biology assigns a symbol (or 'letter') to each acid in the chain. DNA sequences are chains of letters drawn from a 4-letter alphabet of nucleic acids ('nucleotides' or 'bases'). Protein sequences are chains of letters drawn from a 22-letter alphabet of amino acids (20 standard plus 2 unusual ones). A protein sequence can be derived from a nucleotide sequence by collecting contiguous triplets of bases together and turning the triplet into a protein letter; the mapping from nucleotide triplet to protein letter is many-to-one and controlled by a 'genetic code' (essentially a lookup table). It is often useful to compare protein sequences to nucleotide sequences: sequencing DNA is easy in this day and age, but sequencing proteins is not nearly as common. Thus, searching DNA databases for a collection of bases that happen to convert (or *translate*) to a known protein lets one discover proteins without having to construct them.

A list of nucleotides may be converted to a protein sequence by passing each triplet of nucleotides through the genetic code. However, the resulting protein sequence can look very different depending on the exact nucleotide where the conversion starts. Start the conversion process one nucleotide into the list and you get a protein sequence in a different 'frame'. In general, translated protein sequences only make biological sense when you guess the correct frame for translating them out of the three choices possible (i.e. 0, 1, or 2 nucleotides into the list, or into the reverse complement of the list). The translation may even have to account for a 'frame shift', where translation must switch to a different frame midway through converting the nucleotide sequence because of a sequencing error or sequence mutation. The different frames are assigned numbers by convention; the frames 1, 2, and 3 refer to converting the nucleotide sequence (the 'plus strand'), while frames -1, -2 and -3 refer to converting the reverse complement of the nucleotide sequence (the 'minus strand'). When dealing with translated sequences, BLAST works with all six translations simultaneously, so you won't have to guess which frame is biologically correct.

Protein sequences are usually small (the largest one we have is about 40000 letters, with the average length being 200-400 letters); because their composition is drawn from a large alphabet, and also because the genetic code contains redundancy which does not affect the protein sequence after translation, the amount of 'information' in a protein sequence can be high. Nucleotide sequences, on the other hand, can be extremely large (human chromosomes are strings of DNA that are hundreds of millions of bases in size) but possess low information per letter (since their alphabet is small). Large-scale studies have shown that both kinds of sequences, when derived from real organisms, are significantly different from random collections of letters. These characteristics guide the sorts of algorithms used to match up ('align') both kinds of sequences.

BLAST and many other pairwise alignment algorithms attempt to quantify the 'goodness' of a match between two sequences by assigning a score to the match. Statistical theory can be used to determine how big a score would have to be before the alignment that generated that score can be considered unlikely to have arisen by chance. Scores are calculated by assigning a score to each letter of one sequence based on the letter of the other sequence to which it aligns, then adding up all of these pairwise scores. An alignment is 'global' if it includes all the letters of both sequences, and is 'local' if it includes only a subset of one or both sequences. Less common are 'semi-global' alignments, for

example aligning ungapped regions of a protein domain globally to a protein sequence. Most of the interest today is in local alignments (which BLAST finds); they let researchers isolate particular pieces of a sequence as interesting. The statistics behind local alignments are also more straightforward. A local alignment is ‘optimal’ if making any change to the alignment (changing its size, adding or moving gaps) lowers its score.

An alignment is ‘ungapped’ if gaps are not allowed inside the alignment. Allowing gaps within the alignment makes the alignment process much harder, but allows finding alignments that could never be found with ungapped methods. For example, this alignment:

```
tggggtc----gca
|||||||    |||
tggggtcggttgca
```

can manage more exact matches because of the gap in the first sequence. This gapped alignment can achieve a higher score than an optimal ungapped alignment, depending on how much of a penalty is associated with a gap, and how much of a bonus is associated with matching two letters.

The job of sequence alignment algorithms like BLAST is to find most (preferably all) of the highest-scoring (disjoint, or at least minimally overlapping) local alignments between a query and one or more database (‘subject’) sequences. The cutoff score below which alignments are excluded from the BLAST output depends on statistical considerations described elsewhere. It is often true that there are many different-looking alignments in a given query-subject sequence neighborhood that happen to have the same score; BLAST is considered to work correctly if enough alignments are found to cover the extent of the query and subject sequences in this neighborhood. Often, the existence of an alignment between two sequences matters more than the exact form of that alignment.

BLAST Programs

BLAST is actually a collection of algorithms, each tailored to compute different kinds of local alignments (see the O’Reilly book in the Bibliography for much more information about using BLAST):

- BLASTP aligns one or more protein *query* sequences to one or more *subject* protein sequences. The subject sequences can be grouped into a BLAST database using NCBI software tools like formatdb, or merely assembled into a text file.

- RPSBLAST aligns one or more protein query sequences to a database of protein sequence *profiles*. In its present incarnation, each profile in the database is a position-specific score matrix derived from a simultaneous alignment of protein sequences that have functionally similar pieces. Rpsblast is used to detect whether portions of the query sequence are likely to perform a known biological function.

- PSIBLAST aligns a protein query to a protein database, but attempts to build up a query-specific scoring model that will be sensitive only to database sequences that lie within the same 'protein family'. The scoring model scoring model is constructed iteratively and is embodied in a position-specific scoring matrix that takes the place of BLAST's generic score matrices.

- PHIBLAST computes local alignments between a single protein sequence and a database of protein sequences. Unlike ordinary blastp, which works strictly on sequence similarity between query and database sequences, the local alignments produced by phiblast begin with a collection of matches, on the query and database sequences, to a specified regular expression pattern. Using a properly chosen regular expression makes this blast variant much more specific than blastp, since there are human-curated databases of protein functional sites that are encoded as regular expressions. Alignments that include these regular expressions may not have enough sequence similarity to be noticed by ordinary blastp.

- BLASTN aligns one or more nucleotide query sequences to one or more nucleotide subject sequences. *Megablast* is a subspecies of blastn that is optimized for finding very similar alignments in very large sequences. In the old BLAST engine, blastn searches were handled very differently from megablast searches. The current BLAST engine has combined elements of the two types of searches together and will internally switch back and forth between them in order to compute nucleotide alignments as quickly as possible.

- BLASTX takes as input a collection of nucleotide sequences, translates each sequence into its six hypothetical reading frames to produce six different hypothetical protein sequences, then aligns the collection of protein sequences against a collection of one or more subject sequences. Blastx is useful for tasks like determining if a newly discovered nucleotide sequence could contain a known gene within it

- TBLASTN takes as input a collection of protein query sequences and aligns against a collection of nucleotide subject sequences, after first translating each nucleotide sequence into its six hypothetical protein sequences. Tblastn is useful for tasks like searching a newly sequenced genome for known genes.

- TBLASTX searches a translated collection of nucleotide queries against a translated collection of nucleotide subjects. Unlike the other translated search types, it's not obvious what sorts of jobs tblastx is useful for performing. One use appearing in the literature is to align hypothetical protein sequences whose underlying nucleotides have diverged significantly from each other

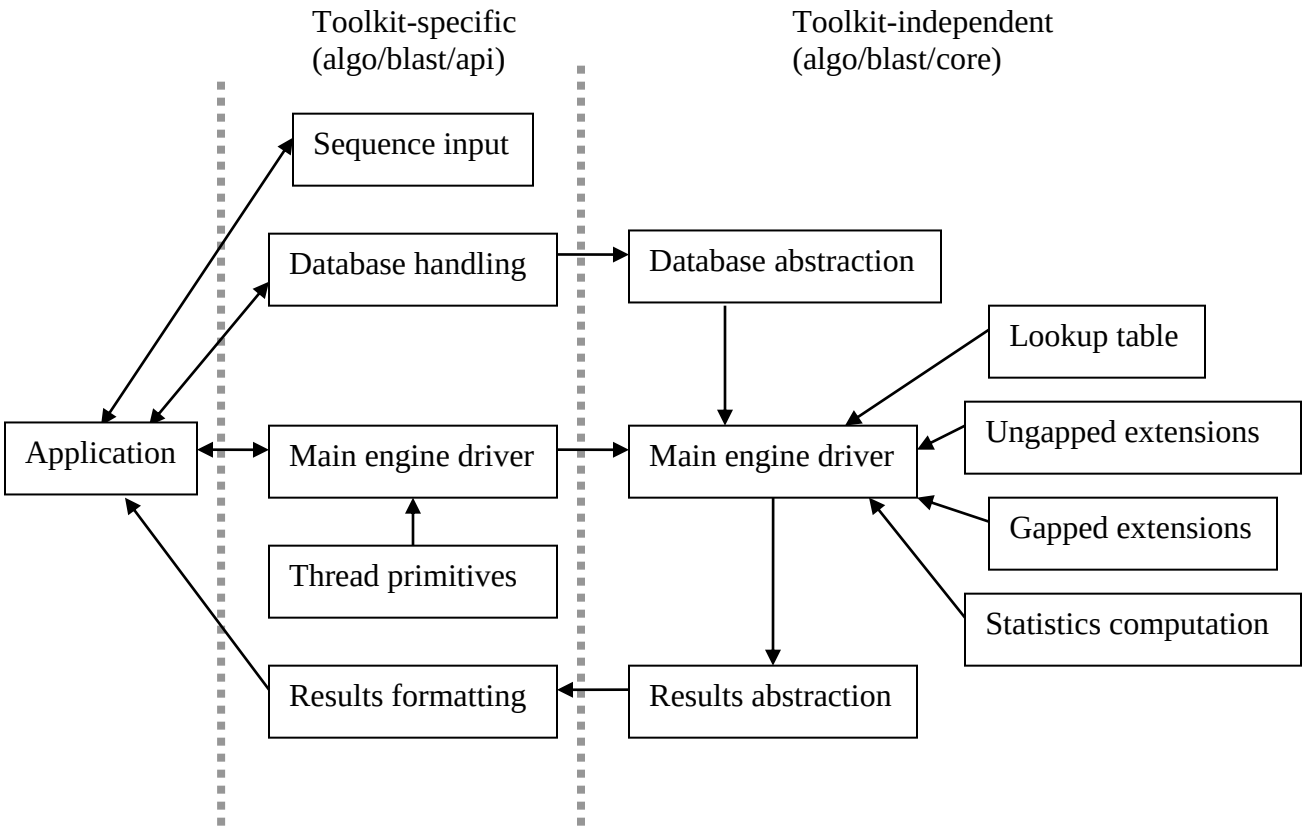
There is a distinction between the format of subject sequences suitable for input to the blast engine, and the format for subject sequence suitable for blast *applications*. For example, the blast engine is flexible enough to work with any collection of subject

sequences that the corresponding toolkit API layer supports, while the C toolkit blastall application may only be given subject sequences collected into a blast database.
The translated searches assume that each alignment found lie completely on a single translation of the converted sequence. However, the engine also allows blastx and tblastn searches to compute *out-of-frame* (OOF) alignments. This option finds local alignments that can slip from one translated reading frame to another in the course of an alignment. An OOF translated search is useful for recovering alignments that contain an embedded frame shift.

All of the above BLAST programs are unified into a single codebase. This simplifies the interface into the BLAST engine, and allows improvements in one low-level component to automatically propagate to all BLAST programs that use that component. The disadvantage to this approach is that portions of the engine must execute conditionally, depending on the BLAST program in use.

Overview of the BLAST Source

The NCBI C and C++ Toolkits both contain source code for the current BLAST engine. The source code is divided into toolkit-specific and toolkit-independent directories, and the latter is C code that is mirrored between the two toolkits (with the minor exception of slightly different header files to establish basic data types and such). A schematic breakdown of the engine source is as follows:



A blast search is divided into three general phases, the seed collection, ungapped extension, and gapped extension phases. The gapped extension phase is further divided into score-only and score-with-traceback extension phases. The following sections cover each of these phases in turn, then describe the construction of the main engine driver and the toolkit-specific BLAST API layout.

Representation of Sequences in Blast

The blast engine core represents a biological sequence in one of only a few different ways, and sometimes adds additional information used in the course of a search. It is the job of the 'api' directories of the C and C++ toolkits to convert sequences from whatever format they happen to be in to the standardized format that the engine core expects. The tables that describe these formats can be found at
http://www.ncbi.nlm.nih.gov/IEB/ToolBox/SDKDOCS/BIOSEQ.HTML

At the lowest level, all biological sequences are represented by a BLAST_SequenceBlk structure (defined in blast_def.h). The contents of this structure, however, depend on the type of sequence represented. This data structure is designed to represent multiple concatenated sequences in a transparent way.

Amino Acid Sequences

The core blast engine expects all protein sequences to be in **ncbistdaa** format; sequence letters assume a value of 0 to 27 inclusive, representing the 22 amino acids and a collection of miscellaneous characters (gaps, end-of-sequence, various ambiguity characters). Note that portions of the blast engine will fail if any sequence letters are actually zero (the gap character), so these must be filtered out before calling the engine.

Protein sequences are represented as an array of unsigned chars, with one amino acid residue per array element. If a SequenceBlk is supposed to represent multiple sequences, all of the sequences are concatenated one after the other with a single zero byte (a 'sentinel') in between each pair of sequences. The number of residues in the sequence is given by BLAST_SequenceBlk::length; in the case of multiple sequences, the length field refers to the combined length of all sequences, plus the sentinel bytes in between them. Note that if a SequenceBlk contains multiple sequences, all the information needed to determine where one sequence ends and another begins, as well as other per-sequence information, is placed in a BlastQueryInfo structure.

The SequenceBlk::sequence field points to the array of sequence data. Elements 0 to length-1 of this array are assumed to point to valid sequence letters. The SequenceBlk::sequence_start field is used to point to sequence data that is imported from other structures. This field is necessary because sometimes, for example when reading blast databases, the first and last sequence letters are actually sentinel bytes. Rather than knowing when and when not to ignore the first letter, the sequence_start field points to the array as it really is and the 'sequence' field always points to valid sequence data.

Nucleotide Sequences

Nuceleotide sequences may be stored in one of three ways in a BLAST_SequenceBlk.

ncbi4na format stores the sequence as an array of unsigned chars, with one nucleotide per array element. ncbi4na represents nucleotides as a bitfield four bits wide, with each bit representing one of the four nucleic acids. This bitfield is necessary to properly represent ambiguity characters, where it is unknown which one of several nucleic acids may actually be present at that position in the sequence. As with protein sequences, 0 is treated as a sentinel byte and should not occur within the sequence.

ncbi2na is a compressed representation of a nucleotide sequence. Each nucleic acid is represented as a 2-bit value, and four nucleotides are packed into each byte of an unsigned char array. Within a byte, bits 6-7 store nucleotide 0, bits 4-5 store nucleotide 1, etc. The NCBI2NA_UNPACK_BASE macro (blast_util.h) is used to extract individual nucleotides from their packed representation.

Ambiguities cannot be represented in ncbi2na format. The preliminary stage of blast will only process the ncbi2na version of each subject sequence, while the traceback stage will add the ambiguities back in and recompute any alignments found. To save disk space, nucleotide blast databases store sequences in ncbi2na format along with auxiliary information needed to restore ambiguities. When the database is created, each ambiguous character in the ncbi2na version of the sequence is randomly assigned one of the nucleic acids that the ambiguity allows. This random assignment can change depending on the position of the sequence within the database, and this is *expected*. Basically one cannot rely on a particular combination of letters appearing when there is no guidance on what those letters should be.

The length field of the BLAST_SequenceBlk gives the number of nucleotides in the sequence, not the number of bytes used to store it. Finally, note that no sentinel byte value is unique for ncbi2na (since 4 nucleotides in a byte use up all bit combinations possible), so that it is *impossible* to have multiple concatentated sequences in ncbi2na format. Future changes could allow this, but implementations cannot rely on sentinel bytes to distinguish between sequences.

Finally, **blastna** format is a permutation of ncbi4na. The difference is that values 0-3 stand for nucleotides, and values > 3 stand for ambiguities. This allows easy comparisons with letters in ncbi2na format; it also allows a 4x4 nucleotide score matrix to become the top left corner of the 16x16 full score matrix for nucleotides. If a SequenceBlk represents multiple concatenated nucleotide sequences, then 0x0f is treated as a sentinel byte (since 0 is a valid sequence letter).

The following table describes the various internal encodings used for nucleotide blast searches:

BLAST Program	Query Sequences	Database Sequences
blastn	read in as blastna	ncbi2na for preliminary search, blastna for traceback search
blastx	read in as ncbi4na, converted to ncbistdaa	ncbistdaa (protein database)
tblastn	ncbistdaa (protein query)	read in as ncbi4na, converted to ncbistdaa
tblastx	read in as ncbi4na, converted to ncbistdaa	read in as ncbi4na, converted to ncbistdaa

The choice of encoding for database sequences is given by the function Blast_TracebackGetEncoding (blast_traceback.c), while the C API uses the first portion of s_SeqLocFillSequenceBuffer (blast_seq.c) and the C++ API uses GetQueryEncoding (blast_setup_cxx.cpp).

If one has an array of bytes and the number of elements in the biological sequence, whether amino acids or nucleotides, then calling BlastSeqBlkNew and then BlastSeqBlkSetSequence will convert the array to a BLAST_SequenceBlk.

When performing a blastn search, by default both the plus and minus strand of the nucleotide query participates in the search. The setup code outside the core engine assumes that nucleotide sequences are plus-strand only when passed in, and explicitly constructs the minus strand as a second query sequence.

Translated Sequences

Additional fields are needed within the SequenceBlk to represent a nucleotide sequence that has been translated into its six reading frames. There are two representations of translated sequences, in-frame and out-of-frame.

In-frame translation assumes that the entire nucleotide sequence exists in the same frame. The nucleotide sequence is translated one frame at a time, and the six complete translations are concatenated together and placed in the SequenceBlk. From this point onward, the SequenceBlk behaves as a collection of six concatenated protein sequences (frames 1, 2, 3, -1, -2, -3, in that order). For ncbi4na format the translation is performed by the function BLAST_GetTranslation, while ncbi2na format uses the function BLAST_TranslateCompressedSequence. Both assume that the genetic code that governs the conversion of nucleotide triplets to protein letters has already been set up. This routine translates a single sequence; the engine assumes that if multiple nucleotide sequences are concatenated then their translations are concatenated in the same order. Hence N nucleotide sequences should be turned into 6N protein sequences before being passed to the core engine as a query for a translated search. The C toolkit function BLAST_SetUpQuery and the C++ toolkit function SetupQueries perform this work.

An out-of-frame (OOF) sequence may slip from one frame to another on a protein-letter by protein-letter basis. The engine uses the in-frame representation for computing ungapped alignments, but when performing out-of-frame gapped alignments it is necessary to keep use separate 'mixed frame' representation of the translated sequence. This contains all the same information as the in-frame sequence, but letters from different frames are interleaved (see function BLAST_InitDNAPSequence). Frames 1, 2, and 3 are mixed together, as are the protein letters of frame -1, -2 and -3. Sentinel bytes are all kept, and stacked three deep at the beginning and end of the mixed-frame sequence so that the in-frame and OOF sequences have the same length. After creation, the mixed-frame sequence is stored in BLAST_SequenceBlk::oof_sequence, and the oof_allocated boolean tells whether this data should be explicitly freed when the SequenceBlk is deallocated (this is currently always necessary). Note that OOF alignments require that the SequenceBlk contains both the in-frame and the OOF version of a sequence; out-of-frame format is a convenience that is only used deep in the blast engine when gapped alignments must be performed; for all other purposes the in-frame version of the sequence is used.

OOF tblastn reads each nucleotide database sequence, converts to the in-frame translated sequence, and then also produces the OOF version of that sequence. OOF blastx is more complicated, because filtering can change the sequence letters. The nucleotide query is read in as ncbi4na, and converted to its in-frame translated form immediately. The production of the OOF version of the sequence must wait until BLAST_MainSetup, so that filtering can first be applied to the set of 6 protein sequences. The filtered versions of the sequences are then interleaved into OOF format. This asymmetry saves having to filter twice (and the filtering code does not know about OOF format anyway).

The conversion of nucleotide letters to protein letters happens under the control of a genetic code. However this code is represented, when translation actually happens the genetic code is a 64-entry lookup table that converts a triplet of nucleotides into a single protein letter. One genetic code is used for virtually all nucleotide sequences, but sometimes a different genetic code is required (for example, in the case of exotic microbial sequences).

One subtle issue is that all query sequences are assumed to use the same genetic code, and all database sequences use the same genetic code (possibly different from the one used in the query). The latter is very important, because it means that arbitrary nucleotide sequences cannot just be thrown together into a database if translated searches are a possibility. Some of those sequences may want a genetic code different from that of other sequences, and while the engine is flexible enough to change the genetic code on a per-subject-sequence basis, there is currently no room in the blast database format for a variable genetic code.

An even more subtle issue relates to how ambiguities are handled. Translating a sequence in ncbi2na format is easy, and can be highly unrolled and optimized (see the routine BLAST_TranslateCompressedSequence). When ambiguous nucleotide letters are a possibility, the routine that performs translation is s_CodonToAA. Genetic codes do not give any guidelines for converting ambiguous nucleotides, so this routine adopts a reasonable compromise: whenever an ambiguity is encountered in one of the three nucleotide letters,

the protein letter corresponding to every possible choice of that ambiguity is computed. If all combinations produce the same protein letter, that letter is the result; otherwise the result is an X character. This process can be made much more efficient, but translation from ncbi4na format is not currently on any critical path.

Maintaining per-Sequence Information

The preliminary stage of the blast engine currently aligns multiple query sequences simultaneously against a single database sequence (if multithreaded, the query sequences are the same in all threads). Sometimes the core must distinguish between individual query sequences, while other times the complete collection is treated like a single query. Note that different frames of a translated query, or different strands of a nucleotide query, are treated by the engine as if they were distinct sequences. This generalization means that the engine may be aligning multiple queries simultaneously even if only a single query was provided.

Information that is unique to individual query sequences is stored in the BlastQueryInfo structure, described in blast_def.h. The C toolkit uses s_QueryInfoSetUp and the C++ toolkit uses SetupQueryInfo_OMF to initialize BlastQueryInfo.

Every individual query sequence, whether a true distinct sequence or one frame of a translated sequence, is assigned a unique number. This 'context' unambiguously identifies that sequence among all the others concatenated together. The rest of the blast engine can mostly deal with contexts only. Information specific to a single context is stored in a BlastContextInfo structure, and BlastQueryInfo::contexts contains the list of all contexts for the query to a blast search. The data in this structure must match up to the actual sequence data in a BLAST_SequenceBlk.

Each context contains several pieces of information:

query_offset	The offset of the first valid sequence letter within BLAST_SequenceBlk::sequence that belongs to this context. If the context is not valid, this number is set to (query_offset of previous context) + (length of previous context) + 1. Note that if element 0 of the context array is invalid, its query offset doesn't matter, since loops that scan through all contexts will skip over element 0
query_length	The number of residues in this context. If the context is not valid, this must be zero
eff_searchsp	The effective search space, which is used for calculating the e-values of blast hits involving this context. If the context is not valid, this must be zero
length_adjustment	Used to compute the effective search space

query_index	Used for determining the actual query sequence to which a context belongs. For example, this is useful for mapping frames of a translated sequence back to the query sequence that produced those translated frames. Query indices run from 0 to (number of actual query sequences - 1)
frame	Number describing which translation frame corresponds to this context. Protein sequences have frame = 0 for all contexts. Nucleotide sequences have frame = 1 or -1, corresponding to the plus and minus strands of each query sequence (the contexts occur in that order). Translated nucleotide sequences have frame = +-1, +-2, or +-3. As with the SequenceBlk, contexts for translated nucleotide queries have the plus strand frames in order and then the minus strand frames in order
is_valid	TRUE if context participates in the search, FALSE if not

For protein queries, all contexts are always valid. For other types of queries, contexts within the context array may not be valid if the search is expected to ignore the plus strand or the minus strand of the query. By default, both strands are searched and all contexts are valid. We've gone back and forth a number of times on this issue, and the current thinking is that it is better to have invalid contexts than to remove them from the context array. There are many places in the core engine that perform arithmetic that assumes nucleotide queries have pairs of frames, and translated queries have frames in groups of six. The special treatment of the query_offset field is to make sure that binary searches through the context array can locate the correct context for a given query offset, even if the array contains invalid contexts. Note that if the plus or minus strand is ignored, BLAST_SequenceBlk does not contain the sequence data for that strand, but BlastQueryInfo *does* contain (invalid) BlastContextInfo structures for the missing strand.

For example, given two nucleotide queries to a blastn search, of size 50 and 100, if searching both strands then the SequenceBlk contains 4 sequences of length 50, 50, 100 and 100, separated by sentinel bytes.

context	query_offset	query_length	query_index	frame	context is
0	0	50	0	1	valid
1	51	50	0	-1	valid
2	102	100	1	1	valid
3	203	100	1	-1	valid

while for the plus strand only, the SequenceBlk has two sequences of size 50 and 100, separated by a sentinel byte.

context	query_offset	query_length	query_index	frame	context is
0	0	50	0	1	valid
1	51	0	0	-1	invalid
2	51	100	1	1	valid
3	152	0	1	-1	invalid

For the minus strand only, the SequenceBlk has the same size but contains reverse complement sequences. The QueryInfo looks like

context	query_offset	query_length	query_index	frame	context is
0	0	0	0	1	invalid
1	0	50	0	-1	valid
2	51	0	1	1	invalid
3	51	100	1	-1	valid

Note that in order to locate the context where a particular hit occurs, the engine uses binary search on BlastQueriyInfo::query_offset (see BSearchContextInfo). This binary search finds the smallest query_offset that is greater than a given sequence offset, and picks the context one back from that. Even when two query_offset values are the same, this will locate the correct context.

Other fields of interest in BlastQueryInfo:

num_queries	the number of query sequences before translation or creation of minus strands. May be less than the number of contexts
first_context	the offset in the contexts[] array of the first valid context. This number may be nonzero

last_context	the offset in the contexts[] array of the last valid context. This number may be less than 2*N-1 for N nucleotide queries, and may be less than 6*N-1 for N translated nucleotide queries
max_length	the maximum number of letters in any context

Any loops over contexts should check is_valid field of each context in the list, and should iterate only from first_context to last_context inclusive.

Representing Database Sequences

Since at present the core engine deals with a single database sequence at a time, there is no need to set up a counterpart to BlastQueryInfo for every database sequence. The minus strand of database sequences is never searched by blastn; instead, hits to the minus strand of a query are turned into hits to the plus strand of the query but the minus strand of the database sequence. In the case of translated database sequences (i.e. tblastn), each database sequence is individually converted into its in-frame (and possibly out-of-frame) translation. Then s_BlastSearchEngineCore iterates through each of the frames, performing one pairwise search at a time between all the query sequences and that one frame.

Another exceptional case is RPS blast, which reverses the role of query sequence and database sequence. RPS blast concatenates the entire protein database into a single giant sequence, then switches the role of query and database before running the rest of the engine, which must then see a query sequence with a huge number of contexts (the whole database worth). Because of this, a partial BlastQueryInfo structure has to be constructed using s_RPSOffsetArrayToContextOffsets

Blast Lookup Table Phase

The initial stage of every pairwise alignment in blast, independent of the precise blast program run, involves a hashtable-based matching of small regions of the query sequence to small regions of the database sequence. The following is a guided tour of this process, along with a description of the optimizations used to make it fast.

The big reason that dynamic programming alignment of sequences can rigorously find all high-scoring alignments is that every letter of the query sequence is tested against every letter of a database sequence. This is also responsible for the number of operations being proportional to the product of the two sequence lengths.

Blast achieves lower asymptotic complexity on average because it makes a simplifying assumption: any alignment of interest between query and database sequence will contain at least one region of W consecutive letters that is high-scoring. Making the value of W too small increases the work, and making W too large dramatically reduces the work but also causes most alignments to be missed. Once W is fixed, the query sequence can be

preprocessed so that the time needed to find these tiny alignments (‘seeds’) is reduced when scanning a large number of database sequences.

For example, consider the ungapped alignment:

```
Query: 6  ALWDYEPQNDDELPMKEGDCMTI 28
          AL+DY+ + ++++ +  GD +T+
Sbjct: 6  ALYDYKKEREEDIDLHLGDILTV 28
```

With *W*=3, the alignment contains several triplets of amino acids whose alignment achieves a high score. Here are the high-scoring 3-letter sub-alignments (‘words’ in the source code, i.e. groups of letters):

offset	query	subject	score
6	ALW	ALY	10
7	LWD	LYD	12
8	WDY	YDY	15
9	DYE	DYK	14
10	YEP	YKK	8
22	EGD	LGD	9
23	GDC	GDI	11
26	MTI	LTV	10

Given the capability to pick out these (and other high-scoring) words very quickly from the original sequences, and the ability to notice that out of all the words found the above group of words clusters together, it should be possible to construct the above alignment without having to test every combination of letters between the two sequences. This is exactly what blast does.

Notice that none of the words that achieve a high score in the alignment are completely identical. All of the aligned words are ‘neighbors’, i.e. contain some exact matches and some matches to which BLOSUM62 assigns a good score. Neighboring words are a good idea if the alphabet is large, but for nucleotide sequences the alphabet is only four letters. In this case the words involved must be longer, since alignments of letters drawn from a small alphabet will contain chance strings of consecutive matches more often. Also, it’s customary to only consider matches or mismatches for nucleotide words. The table-driven methods described below work equally well for finding neighboring words or for finding exact matches between words.

For nucleotide blast, the lookup table only deals with exact matches. Even if later stages actually use a score matrix when aligning nucleotides, the initial phase that involves matching up query words with database words only considers words to match if all letters are identical.

Use of Lookup Tables

Given the query to a blast search, the first step is to isolate all the W-letter words in the query sequence, and the offset into the sequence where each word occurs. These offsets are what is tabulated. Later, when a database sequence appears, the task is walk through each offset of the database sequence and to recover from the lookup table the offsets of query words that score well when aligned at that database offset. The collection of [query_offset, db_seq_offset] pairs is the output of the 'DB scanning' phase (described later).

If the alphabet has *A* letters, then the lookup table is of size at least A^W. Basically, each possible *W*-letter word hashes (via some 1-to-1 function) to a unique location in the lookup table. The current hash function is to consider each letter as a number from 0 to *(A-1)* which can fit in *B* bits (=ceil($\log_2(A)$)), and to concatenate the integer values of *W* letters together to form a *B*W*-bit value. This value is the index into the lookup table that corresponds to the given *W*-letter word. Typical parameters for blast searches are listed below:

program	alphabet size	*W*	*B*	number of hashtable cells
blastp, blastx, tblastn, tblastx	28	3	5	32768
blastn	4	8	2	65536 (see below)
megablast	4	11 or 12	2	2^{22}, 2^{24} (see below)

The fact that these lookup tables are quite large is the basis for many of the performance optimizations present in the code. Each entry of the lookup table must point to a list of numbers, whose size and contents are determined when the query sequence is preprocessed.

The procedure for filling the lookup table is as follows:

```
for (i = 0; i <= query_length - W; i++) {
   query_word = [ query[i],query[i+1],...query[i+W-1] ]
   /* neighboring loop */
   for all W-letter combinations of A characters {
      db_word = current list of letters

      if (db_word exactly matches query_word ||
          score of aligning query_word with db_word >= T) {
         add i to list of query offsets in lookup_table[HASH(db_word)]
      }
   }
}
```

The quantity *T* is the 'neighboring threshold', a measure of how good the word alignment must be in order for its associated query offset to make it into the lookup table. Basically, for the i^{th} query word the above adds offset i to the lookup table entries associated with any possible word a database sequence may contain, that is sufficiently similar to word i of the query sequence. For nucleotide searches, *T* is always infinite. Note that 'word i' above means the query word starting at letter i of the query. All blast lookup tables index words by their starting offset. This is a reversal of the old engine, which indexes words by their *ending* offset.

Later, given a subject sequence, for subject offset j we form the *W*-letter word at that offset, and copy out all of the query offsets in lookup_table[HASH(word j)]. The list of offset pairs (list[0],j), (list[1],j), etc. then is exactly the list of *seed* alignments between the query and database sequence. Especially when the query is short (i.e. has few words to index) and the database is large (i.e. the time needed to create the lookup table is amortized over searching many database sequences) the scanning process is tremendously faster than a brute force attempt to align all of every sequence. It finds all seeds with score >= *T* in time that essentially depends linearly on the database size (on average; the worst-case running time is still proportional to the product of query and database size).

The HASH() function is very simple, since the lookup table is large enough to not need a complicated function to reduce the chance of collisions. Basically it involves concatenating the bits that make up the word to be hashed, then using that to index directly into the table. For protein searches using concatenated letters drawn from a 28-letter alphabet, each letter is embedded in 5 bits and the set of 5-bit quantities is concatenated to form the lookup table index. For nucleotide searches the lookup table is typically 8-12 letters wide, and HASH() is thus a concatenation of 8-12 values, each two bits wide. Note that in the protein case this means that large expanses of the lookup table will never be used, because a protein letter will never have a value of 27-31 that its 5-bit representation allows. The code currently does not take advantage of that fact to compress the lookup table, and maybe it should. In any case, one small optimization for protein lookup tables is that when allocating the memory for the table, the new engine allocates 32 * 32 * ... * 32 * 28 entries. HASH() is unchanged, but the last letter doesn't have to account for the 'hole' in the table after it, so that the lookup table will have 28k cells instead of the full 32k.

One final complication for nucleotide lookup tables: since HASH() expects letters in a word to be exactly two bits wide, ambiguities cannot be handled by the lookup table. When the table is built, query words that contain an ambiguity do not contribute any lookup table entries, and a database sequence has all of its ambiguities converted into one of the four bases before the blast code ever sees it. The ambiguities are added back in at a much later stage in the alignment process.

Lookup tables are constructed by LookupTableWrapInit (lookup_wrap.c). Lookup table construction proceeds in two phases for both protein and nucleotide searches. The first constructs a generic lookup table by repeated calls to BlastLookupAddWordHit (blast_lookup.c). The generic lookup table is simply an array of pointers (the 'backbone') to lists of 32-bit integers, with one list per hashtable cell. Each list has one element that lists the number of words allocated for that list, one word that gives the number of query offsets in the list, and then the collection of query offsets. The list for each hashtable cell grows dynamically, and when the query is large this sort of layout saves half the memory that the old blast engine would need.

Once the generic lookup table is filled, it is then packed into an optimized structure. The type of optimized structure depends on several factors, and each type is described below. Note that a megablast-style optimized structure is constructed directly, without forming a generic lookup table first.

Optimizing Generic Lookup Table Construction

When the query is large or the database is small (or even when not performing a database search but just aligning two sequences), the time needed to construct the generic lookup table starts to become more important. Fortunately, there are many optimizations possible that can reduce the required time by up to two orders of magnitude.

The nucleotide case is easiest: in this case we only care about exact matches, so there is no need to construct all of the possible neighboring words. Every query word updates only one lookup table location, so that the inner for() loop in the code above is not necessary at all. Generic lookup tables are constructed by BlastLookupIndexQueryExactMatches (blast_lookup.c), and the construction process is fast enough to never be a bottleneck.

Protein blast does require neighboring word comparisons. Here there are two problems to solve: generating all *W*-letter words efficiently, and performing the alignment comparison efficiently. Both of these jobs are performed by BlastAaLookupIndexQuery. The most important case is *W*=3, since this is most common; *W*=2 is already fast enough, but ideally higher values of *W* should be possible as well. Protein generic lookup tables are constructed by BlastAaLookupIndexQuery (blast_aalookup.c), and there are separate code paths for ordinary score matrices and position-specific score matrices. We will only describe the former in detail.

With a 28-letter alphabet and $W = 3$, there are 28*28*28 = 21952 three-letter words possible. For large queries (especially translated queries, where the underlying nucleotide sequence can be large) it is reasonable to expect the same 3-letter word to appear again

and again in the query. It is a waste of effort to recalculate the inner for() loop above for a query word that has appeared before; the current query offset will go into all of the same lookup table locations calculated the last time the same word appeared.

To take advantage of this fact, s_AddNeighboringWords actually performs the generic lookup table generation process twice. The first lookup table will only contain exact matches; this has the effect of grouping together query offsets that refer to the same *W*-letter word. Next, the code walks through this temporary lookup table. If a table entry contains one or more query offsets, the full neighboring computation is performed on the first query word in the list. Whenever this first offset is placed into an entry in the generic lookup table, the entire chain of offsets is also added to that entry. This basically means that no matter how large the query sequence actually is, the neighboring loop will only run at most 28^W times (and probably much less than that). This optimization is completely orthogonal to the problem of making the neighboring loop run fast.

The original blast code created the entire list of *W*-letter words, and then scanned it for words that score highly when aligned with the query word. When *W*=3 the memory used is modest, but for larger *W* this is impractical.

If *W* was fixed it would be tempting to build up the next word incrementally:

```
for (i = 0; i < 28; i++) {
   change position 0 to letter i
      for (j = 0; j < 28; j++) {
         change position 1 to letter j
         for (k = 0; k < 28; k++) {
            change position 2 to letter k
        }
    }
}
```

This uses no memory at all, but the number of loops must be known beforehand. A compromise would be to work recursively, as the function s_AddWordHitsCore (blast_aalookup.c) does:

```
MakeWord(int pos, char buf) {
   if (position == W-1) {
      for (i = 0; i < 28; i++) {
         set buf[pos] to letter i
         calculate alignment score and add
               the query offset to cell HASH(buf) of
               the generic lookup table if the score
               exceeds T
      }
      return;
   }
   for (i = 0; i < 28; i++) {
      set buf[pos] to letter i
      MakeWord(pos+1, buf)
   }
}
```

The overhead here is somewhat higher, but at least arbitrary *W* can be handled without running out of memory. The big advantage to this method, however, is that it makes pruning whole groups of database words very easy. Remember that we're looking for database words that score above a threshold *T* when aligned with the query word. Suppose a database word has *W*-1 of its letters specified, and has a score *S* for aligning those letters with the first *W*-1 letters of the query word. If the last letter cannot achieve a score of at least *T* - *S* when aligned with the last letter of the query word, then there is no need to check any of the 28 database words; we know beforehand that they cannot achieve the threshold *T*. This check can also be extended to positions before the last letter too, and if pruning happens then even more database words are skipped.

The old blast engine implemented pruning on the last two letters of the word, but with the recursive formulation above we can implement pruning at all letter positions, with only minor changes. The score begins with the best score possible for the given query word, and as more letter positions in database words are fixed the score gradually converges to the correct value for a given database word.

```
find_neighbors(query_word) {
   max_score[i] = the maximum score of row i
                  of the score matrix
   curr_score = 0
   for (i = 0; i < W; i++)
      curr_score += max_score[query_word[i]]
   MakeWord(0, buf, curr_score)
}

MakeWord(int pos, char buf, int curr_score) {
   curr_score -= max_score[pos]  /* remove best-case score */
   if (pos == W-1) {
      for (i = 0; i < 28; i++) {
         set buf[pos] to letter i
         calculate alignment score and add
                the query offset to cell HASH(buf) of
                the generic lookup table if the score
                exceeds T
      }
      return;
   }
   for (i = 0; i < 28; i++) {
      /* replace the best-case score with the exact
         score of aligning query_word[pos] to letter i,
         and only recurse if there's still a possibility
         to exceed the threshold */
      new_score = curr_score + matrix[query_word[pos]][i];
      if (new_score >= T) {
         set buf[pos] to letter i
         MakeWord(pos+1, buf, new_score)
      }
   }
}
```

For sensibly chosen values of *T*, the pruning reduces the number of database words that must be generated by an order of magnitude. Further, because pruning takes place for all positions in the word, lookup table construction is much faster for $W > 3$.

There is a subclass of protein searches that need special consideration when the lookup table is being constructed. Query sequences that are accompanied by a Position-Specific Score Matrix (PSSM) have a separate score matrix row for each offset in the query sequence. Unlike the square-matrix case explained above, the offsets of identical query words cannot be lumped together (since their score matrix rows need not be identical). Further, while the same recursive procedure can be used to generate neighboring words to compare to query words, the maximum score of each PSSM row must be determined. The top-level function that performs this is s_AddPSSMNeighboringWords and the recursive function that fills the generic lookup table is s_AddPSSMWordHits.

Optimized Protein Lookup Table

Originally blast used a structure equivalent to the generic lookup table to conduct the actual database search. However, in the late 1990s it became clear that using the generic table to retrieve query offsets caused problems with memory latency. By the time the table was constructed, query offsets were sitting in arrays of integers that were scattered all over memory, and the hashtable itself exceeded the cache size of current microprocessors. Hence the act of retrieving query offsets from the lookup table would involve one read of the hashtable (causing a cache miss), and if the resulting pointer indicated that query offsets were present at that position then they would be retrieved from the relevant list (causing another cache miss). Actually, early versions of blast stored all of the query offsets in a single array, with query offsets that belonged to the same hashtable cell being linked together through pointers. Not only did this double the memory consumption, but the act of traversing the array involved a cache miss for every query offset retrieved, since offsets were generally not contiguous in memory.

Because accessing the lookup table for query offsets is potentially critical for performance, there is a second round of processing performed on the generic lookup table after all of the query offsets are loaded into it. This phase, originally contributed by Compaq in the late 1990s and currently implemented via the function s_BlastAaLookupFinalize (blast_aalookup.c), converts the generic lookup table into a packed form, and adds auxiliary data structures that help reduce cache misses later.

In the common case, the query sequence is not very large; this in turn means that often the generic lookup table is mostly empty. BlastAaLookupTable::pv_array (pv = ‘presence vector’) is used to exploit this; it is a bitfield with one bit for each cell in the hashtable. If an entry in the hashtable contains query offsets, the corresponding bit in the PV array is set. Later code first checks the PV array to see whether there are any query offsets in a particular lookup table entry. Because the bitfield takes up much less memory than the actual thin backbone, it can potentially stay in cache even when the hashtable cannot. In principle, when the query is short this means that many cache misses are avoided.

A second optimization replaces the hashtable with a 'thick backbone', an array of type AaLookupBackboneCell. This structure contains one word for the number of query offsets in that lookup table entry, and AA_HITS_PER_CELL (=3) additional words. Since the common case has most backbone cells empty, it follows that backbone cells which are not empty will not be very full either, on average. If the number of query offsets at a given position in the lookup table is AA_HITS_PER_CELL or less, then all query offsets are stored directly in the AaLookupBackboneCell. All backbone cells that contain more than this number of query offsets have those offsets packed contiguously into BlastAaLookupTable::overflow array. In those cells, one of the words in the AaLookupBackboneCell is the offset into the overflow array where the query offsets for that lookup table position begin, and the other words are left unused.

Accessing the lookup table therefore involves checking the PV array (which is usually in cache), then checking the thick backbone (causing a cache miss). If the number of offsets to retrieve is small, they are available for free (since the cell is now in cache). Otherwise the offsets are accessed sequentially from the overflow array, usually causing a cache miss but possibly not (the overflow array may fit in L2 cache, and in any case retrieving one group of offsets can pull its neighbor(s) into cache as well).

Increasing the amount of reuse is even more important today than it used to be, since cache has gotten much faster than main memory (i.e. a factor of 20 faster instead of the factor of 3 that was typical when these optimizations were first implemented). My belief is that the sort of tuning that made sense for an Alpha with 4MB of cache (the original target) is no longer optimal for e.g. a Pentium with 512kB of extremely fast cache and much slower RAM.

Optimized Nucleotide Lookup Table

Nucleotide blast can also benefit from a compressed lookup table, but the characteristics of a nucleotide search are different from that of a protein search. Nucleotide queries can be very small or very large, the word size *W* can be small or large, and in the nucleotide case the generic lookup table only indexes exact matches. Thus the lookup table size and number of entries varies widely (although the average lookup table is very sparsely populated, much less so than the blastp lookup table). The performance regime in which nucleotide blast operates is therefore quite wide, and optimal performance requires data structures that can adapt to the specifics of the search.

Experience shows that the vast majority of nucleotide searches have tiny queries. In that case, the lookup table is very sparsely populated and hits to database sequences are pretty rare. Most of the time the generic lookup table indexes 8-mers, i.e. groups of 8 contiguous bases. While it is possible to use a nucleotide version of the BlastAaLookupTable structure, in practice this is overkill. The vast majority of bits in the PV array are empty, so the latency to access those bits is the limiting factor when scanning a database sequence. Unfortunately, profiling shows that very often on x86 hardware, the PV array cannot stay in the fastest processor caches during the search, possibly because the other parts of the search kick it out. Even when the whole array is in cache, checking the PV array takes a lot of time. Possibly this is because the sequence of machine instructions

that construct the array index to test, load the word containing the appropriate bit and then test for bit set introduce data dependencies that mean that a modern out-of-order CPU cannot process many scanning iterations in parallel. Combined with the fact that the PV array is in secondary cache much of the time, nucleotide blast in these circumstances would be faster on average if the backbone was accessed directly, assuming the backbone itself is small enough to reside in secondary cache.

Cameron and Williams were the first to realize that retuning the optimized lookup table would yield increased search performance, and their techniques have been ported to NCBI blast. If the nucleotide search is such that

- the maximum query offset to index is less than 32768
- the number of query words to index is less than 32768

then blast will create a lookup table of type BlastSmallNaLookupTable (blast_nalookup.h) by calling BlastSmallNaLookupTableNew(blast_nalookup.c). This function will create the generic lookup table and then call s_BlastSmallNaLookupFinalize to create the optimized table. The latter function can fail, since it is theoretically possible that a query sequence be sufficiently compositionally biased that it generates too many words to index, especially if both query strands are considered together. If that happens, or if the query sequence clearly is not suitable for creation of a BlastSmallNaLookupTable, then blast will fall back to creating a nucleotide version of the protein lookup table (BlastNaLookupTable) by calling BlastNaLookupTableNew(blast_nalookup.c). If the SmallNa style of lookup table is possible, it should *always* be used; while there could be no performance benefit (depending on the underlying machine), I have not seen a case where it would be more efficient to use the larger lookup table, given the choice.

The SmallNa table contains a backbone, but each backbone cell contains just one 16-bit integer. The overflow array is also a list of 16-bit integers. If the value at a particular cell is nonnegative, then that position in the hashtable contains exactly one query offset, equal to the cell value. If the value is -1, the cell is empty. Finally, if the cell value is -x ($x > 1$) then the query offsets for the cell begin at offset x of the overflow array, and continue until a negative query offset is encountered. The use of 16-bit integers gives rise to the restrictions on the query described above, and the structure of the overflow array means that one cannot tell whether the SmallNa table is allowed until the overflow array is actually built (by calling s_BlastSmallNaLookupFinalize), since it contains extra entries that are not query words.

When the SmallNa table indexes 8-mers, the working set for the backbone is 128kbytes and the overflow array is typically quite small. There is no PV array, so accessing the lookup table involves computing the table index and accessing the hashtable with a single load instruction. This short instruction path means that out-of-order processors can overlap many lookup table accesses simultaneously, and if a PV array was going to be in external cache anyway then not using it actually reduces cache accesses. The end result is a 15-20% overall speedup on Intel hardware for small queries.

Megablast Lookup Table

Megablast was designed for nucleotide queries that are very large, and for finding alignments that are very high-scoring; this implies a large value of *W*.

The lookup table used by megablast indexes 9-12 letter exact matches with query words, but many of the principles explained above still apply. The main megablast lookup table structure is BlastMBLookupTable, defined in mb_lookup.h.

Unless the query is extremely large (millions of bases), the size of the lookup table will guarantee that the vast majority of table cells are empty. Most lookup table entries that are not empty will have very few query offsets (usually just one). There are two main data structures used to hold query offsets: BlastMBLookupTable::next_pos is an array of integers the size of the query sequence; offset i in this array denotes query offset i. Also, BlastMBLookupTable::hashtable is a thin backbone with one word for each of the lookup table entries. If a lookup table entry has no query offsets, the corresponding entry in hashtable[] is zero; otherwise it is an offset into next_pos[].

The procedure for building the megablast lookup table is as follows (note that the functions s_FillContigMBTable and s_FillDiscMBTable in mb_lookup.c are much more complicated):

```
for (i = 0; i < query_length - W; i++) {
   query_word = [ query[i],query[i+1],...query[i+W-1] ]
   index = HASH(query_word)
   next_pos[i] = hashtable[index]
   hashtable[index] = i;
}
```

When complete, the megablast lookup table is a giant collection of linked lists. Most of these lists are empty; the head of list i is hashtable[i]. This value is the first query offset, and in turn gives the offset into next_pos[] of the next query offset. Making next_pos the size of the query sequence means that query offsets don't have to be explicitly listed; rather, the position in next_pos[] is synonymous with the query offset, and the actual value at that position serves as a 'next pointer' to the succeeding query offset in the chain. A value of zero universally means end of chain, so that all offsets must actually be in the range [1,query_length] and not [0,query_length-1].

Accessing the lookup table involves calculating HASH(database_word), reading the hashtable[] word at this index, and if nonzero retrieving the linked list of query offsets and decrementing each such offset. Since non-empty lists are rare, and all lists are short, there is no need for extra packing of next_pos[]

For much more on the process of optimizing use of the lookup table in megablast, see the Appendix "Optimization of Megablast"

Merging Blastn and Megablast

Megablast was originally developed as a separate collection of source in the old engine; it has its own lookup table, its own extension routines and its own gapped alignment infrastructure, all distinct from the machinery of blastn searches. Recent progress with the new engine has allowed blastn and megablast to look much more like each other, to the point where the blast engine can automatically choose the most efficient method used to align nucleotide sequences. In fact, other than using greedy gapped extension, megablast is just blastn with different data structures.

The improvements start with two fundamental questions: why does the blastn lookup table have to be 4 or 8 letters wide, and why does the megablast lookup table have to be 12 letters wide? In the beginning the tables had to be a multiple of four letters wide because the scanning of database sequences happens 4 letters at a time, and making everything a multiple of four meant that a compressed database sequence could be scanned a full byte at a time. However, blast currently uses improved scanning methods that need strides different from 4 anyway. The other big reason these sizes were used was that the code was written that way and changes were too difficult. The new blast engine is modular enough that these fundamental statements about the way blast works can all change, and change adaptively to suit each particular search.

The runtime needed for the database scanning and initial word extensions depends on several factors. If a lookup table word has more letters, the table is accessed less often and so the number of initial extensions is reduced; but more letters mean a larger table whose cache performance is worse. A smaller table has the opposite effect: the latency of table access is reduced but the number of hits and the number of extensions performed on those hits both increase. Thus, for each search there is an optimal table size that represents a good balance between these conflicting goals. A small query will not fill up even a narrow lookup table, and so should benefit from the improved cache performance of a table with fewer letters per word. Likewise, a large query will saturate an ordinary-size lookup table, and it's more efficient to use a wide table that is accessed much less often. To add more variables, the initial word extension phase of the new engine does not care what sort of lookup table produced hits that need extending; all it wants to know is where the hits are located. This means we can use a megablast lookup table for a blastn search or a blastn lookup table for a megablast search.

Currently, blastn lookup tables can contain 4-8 letters per word (inclusive) and megablast lookup tables can contain 9-12 letters per word (inclusive). The type of lookup table and its width depend on the number of words in the query that need indexing (which can be accurately estimated, and is typically close to 2x the number of bases in the query) and the word size.

The following table gives the choices the new engine will make; all of these were experimentally determined, and the optimal choice will probably vary between different microprocessors. The logic to choose the lookup table type and width is located in the function BlastChooseNaLookupTable (blast_nalookup.c). Note that 'blastn' below means

the BlastSmallNaLookupTable type by default, or BlastNaLookupTable if the SmallNa table cannot be used on the query.

number of words to index	search word size	chosen table type	chosen table width
any	4	blastn	4
any	5	blastn	5
any	6	blastn	6
0 – 250	7	blastn	6
250+	7	blastn	7
0 – 8500	8	blastn	7
8500+	8	blastn	8
0 – 1250	9	blastn	7
1250 – 21000	9	blastn	8
21000+	9	megablast	9
0 – 1250	10	blastn	7
1250 – 8500	10	blastn	8
8500 – 18000	10	megablast	9
18000+	10	megablast	10
0 – 12000	11	blastn	8
12000 – 180000	11	megablast	10
180000+	11	megablast	11
0 – 8500	12	blastn	8
8500 – 18000	12	megablast	9
18000 – 60000	12	megablast	10
60000 – 900000	12	megablast	11
900000+	12	megablast	12
0 – 8500	13+	blastn	8
85000 – 300000	13+	megablast	11
300000+	13+	megablast	12

Collecting seeds

There are several routines whose job is to collect seed hits from the blast lookup table. They are:

- BlastAaScanSubject (blast_aalookup.h)

- BlastRPSScanSubject (blast_aalookup.h)

- A large collection of routines in blast_nalookup.h

- PHIBlastScanSubject (phi_lookup.h)

For most database searches these routines will touch every letter of the database being searched. All of the routines fill an array of type BlastOffsetPair (lookup_wrap.h) with the offsets of hits found. For every blast program except phiblast, each BlastOffsetPair represents one high-scoring alignment between a concatenated set of query sequences and a single subject sequence. In the case of phiblast, the query sequence is scanned once and PHIBlastScanSubject is called once for each subject sequence; every BlastOffsetPair then contains the start and stop point of each pattern hit to the subject sequence.

In general, a ScanSubject routine works from the beginning to the end of a subject sequence. The ScanSubject can be called several times on the same sequence, if there happen to be many hits to that sequence. There is a limit to the number of hits that a single ScanSubject call can retrieve; this limit is calculated by GetOffsetArraySize (lookup_wrap.c) and is blast-program- and blast-lookup-table specific. The limit is chosen so that every time the lookup table is accessed, there is guaranteed to be room for all of the query offsets at that lookup table entry. An upper bound on the number of lookup entries is computed when the lookup table is constructed. Blast lookup table structures all contain a 'longest_chain' field that gives this upper bound (note that to reduce the construction time for the megablast lookup table, longest_chain in this case is given a conservative value that avoids having to examine every lookup table cell). The maximum number of hits returned by a ScanSubject call is longest_chain plus a fixed amount; choosing the limit this way has several advantages:

- Since the lookup table is accessed once per subject sequence word, if room in the array of BlastOffsetPairs is guaranteed for at least one lookup table access then a single ScanSubject call is guaranteed to make forward progress through the subject sequence

- Being able to finish a complete lookup table access means that no extra bookkeeping information is needed to remember how many more query offsets need to be extracted from a single cell on the next ScanSubject call. This in turn allows the lookup table structure to be abstracted away to a certain extent, and to allow different lookup tables to do the same job at higher efficiency

Optimizations to ScanSubject

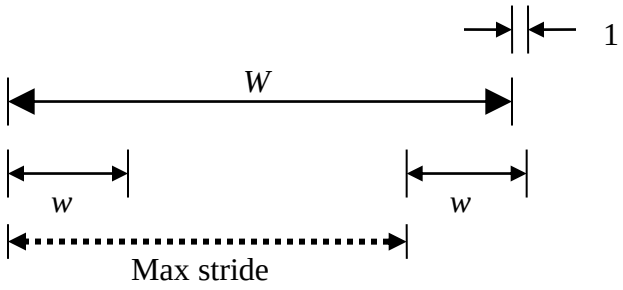
For protein sequences, the number of letters in a subject word matches the 'width' of the lookup table; this means the protein-based ScanSubject routines can walk one letter at a time through the subject sequence. However, for most nucleotide searches the number of letters in a lookup table word is less than the actual word size for the blast search (other-

wise, for example, a word size of 28 in a megablast search would require a lookup table with 4^{28} entries, far larger than is practical). Apple and Genentech, in their version of the blast source, use this difference in word width to accelerate nucleotide ScanSubject. Their changes to the old blast engine needed to achieve this speedup have been implemented in the current blast engine and hence can benefit searches running on any platform.

In the old blast engine, since the subject sequence is always assumed to be in ncbi2na format for the nucleotide wordfinders, a subject word is formed and used to access the lookup table for every fourth base (i.e. every byte of the compressed subject sequence). This 'stride' of 4 is independent of the number of letters in a lookup table word and independent of the word size used in the blast search. It becomes increasingly wasteful when the disparity between these two quantities is large.

Words used in the lookup table for blastn are usually 8 letters wide; if the word size of the blast search is 20 then any hits between query and subject involve at least 20 contiguous exact matches. You don't need to test the subject sequence every four letters to find runs of 8 matches that may be part of a 20-letter run of exact matches; you can get away with less testing than that. Since every test of the subject sequence involves acessing the lookup table, as the disparity between the blast word size and the lookup table word size increases then in principle the number of lookup table accesses (and hence the runtime for the ScanSubject routine) can decrease linearly. Especially for megablast, where large word sizes are common and up to 70% of the runtime can be tied up in ScanSubject, there is a potential for a significant speedup.

What is the largest possible stride that will not miss any hits between query and subject? Let the blast word size be *W* and the lookup table word size be *w*. A hit of size *W* will be missed if in the course of a ScanSubject call we do not form a subject word that lies completely inside the hit. The worst case picture to consider is thus:



If the two words of size *w* are any farther apart than the stride above, then a word of size *W* can potentially fit between them so that neither of the two smaller words lie completely inside. The size of this maximum stride is $W - w + 1$. With $W = 20$ and $w = 8$ this means a stride of 13, more than triple the ordinary stride of 4. Hence for these settings a strided ScanSubject can be expected to run more than three times faster than the old engine. Note also that any stride from 1 to $W - w + 1$ will work equally well, though obviously the larger the stride the faster the scanning process will go. Because a range of strides is possible, it may sometimes be advantageous to use a stride slightly smaller than

the maximum possible; for example, if the stride were a multiple of 4 then the bit-twiddling needed to form subject words is slightly reduced. With the current state of the nucleotide subject scanning code this turns out not to be needed, and the largest possible stride is always chosen.

Another advantage to this striding method occurs when $W < 11$. Without striding we are forced to use lookup tables of size 4 and ScanSubject takes huge amounts of time (due to the enormous number of 4-letter matches that can occur by chance in nucleotide sequences). Use of AG striding allows a lookup table width that is much larger. For example, word size 7 can have a lookup table of size 7 with a stride of 1, or a table of size 6 with a stride of 2. The wider lookup table drastically reduces the number of seeds to process, and thus also reduces the number of extensions of those seeds. This can make nucleotide searches with small word sizes up to several times faster. Whether a slightly larger stride turns out faster depends on whether the improved cache performance of a smaller lookup table outweighs the extra time needed by an increased number of ungapped extensions.

Discontiguous words

Discontiguous megablast works slightly differently from conventional megablast. It does not require complete runs of matches in query and subject words, but instead only x nucleotide matches in y positions (for x = 11 or 12 and y = 16, 18 or 21). The exact positions in a word where a match is required are described by a bitfield called a 'template'. Use of discontiguous words is justified on theoretical grounds; one reason is that every third nucleotide is not so important in a nucleotide sequence when translation to a protein sequence occurs. Hence discontiguous megablast can be expected to find hits that are too dissimilar for megablast to find.

Lookup tables are built up in the same way as conventional megablast, except that the template is applied to generate offsets into the lookup table. ScanSubject routines work the same way as in the contiguous case, except that the stride is always fixed and the template is applied to generate subject words. Another difference with discontiguous megablast is that one can optionally use two simultaneous lookup tables, in the hope of recovering more alignments that would be missed if just a single discontiguous template was used. Both tables are accessed for every subject word, which means the two templates must evaluate to the same lookup table size. Also, longest_chain is a worst case value for both lookup tables combined, to accomodate the possibility of twice as many hits in a single ScanSubject call.

The only allowed stride for discontiguous megablast is 1. In principle any stride will work, although any stride larger than 1 will potentially miss some hits because here the word size matches the lookup table width ($W = w$). Blast uses two types of templates, 'coding' and 'optimal'; the former are designed to selectively ignore every third base, while the latter are constructed to optimize the probability that a pair of random words of size (template_length) containing W matches will be found.

Designing templates for nucleotide scanning is something of a cottage industry; several researchers have used a variety of techniques to develop templates that optimize some

kind of probability. The search space is very large when the template length is long. In principle it is straightforward to incorporate new templates into discontiguous megablast, but to date there has never been a comparative study of the effectiveness of even the existing templates.

Nucleotide ScanSubject Architecture

The nucleotide ScanSubject operation is usually critical for performance, but packing all of the needed flexibility for any ScanSubject operation into a single routine is necessarily wasteful. For example, if a ScanSubject routine has to handle word size 8 with arbitrary stride, then it must assume a whole new 8-mer must be constructed each time the lookup table is accessed. Given that that 8-mer may not be aligned on a 4-base boundary, in the general case the subject sequence must have three bytes loaded to form each 8-mer. However, if the stride was 1, then loading one byte of the subject sequence is enough for 4 lookup table accesses. As another example, if the stride is a multiple of 4 then several mask and shift operations become unnecessary. Even when the masks and shifts are required, knowing the stride beforehand allows the compiler to use compile-time constants instead of variables, and this can improve performance.

In order to achieve maximum ScanSubject performance, for nucleotide searches the blast engine will choose a ScanSubject routine from a pool of about two dozen candidates; that routine is used for the entirety of the search. The choice depends on the lookup table type, word size, and scanning stride. A pointer to the appropriate ScanSubject routine is determined by BlastChooseNucleotideScanSubject, and the pointer is stored as a void pointer within the lookup table structure (to avoid artificial dependencies between lookup table construction and scanning). This routine in turn calls lookup-table-type-specific routines that know the details of which specialized ScanSubject routine is available for a particular search, while the scanning routines themselves are static functions in blast_nascan.c.If a specialized routine is not available for a particular combination of word size and scanning stride, generic routines that can handle any combination (at reduced efficiency) are chosen.

The specialized routines each use as many of the following optimizations as are applicable:

- If the stride is 1 or 3 plus a multiple of four, the scanning loop is unrolled by 4. If it is 2 plus a multiple of 4 the scanning loop is unrolled by 2. The unrolling eliminates several shift and mask operations, and turns the shift and mask quantities into compile-time constants. The unrolled routines use a switch statement to jump into the middle of the unrolled loop when scanning starts

- If the stride is less than 4, the scanning uses an accumulator (to store subject words across scanning iterations instead of forming the subject word from scratch). This reduces the number of memory accesses to the subject sequence, and the number of pointer increments

- If the word size is a multiple of 4, mask operations are removed when the offset into the subject sequence is known to be a multiple of 4. Because the loops are unrolled, all strides can take advantage of this optimization once in a while

- In the specific case of lookup table width 8 and stride 4, all the unrolled loop iterations look the same, so that scanning can be made to finish only at the bottom of the loop. This eliminates several bounds checks, and makes this case the fastest of all the scanning routines. Happily, this loop is also the most common for ordinary blastn

- Discontiguous megablast always uses an accumulator since it scans at stride 1. For the most common discontiguous templates, the mask and shift operations to form a single subject word are permuted so that the accumulator does not have to be shifted first. The reduction in arithmetic for these templates makes them more than 20% faster

Finally, note that when the stride is exactly a multiple of 4, the generic scanning routines have a special case that handles this essentially as fast as possible, so the generic routine is used. This cuts down greatly on what is already a large number of static functions.

The speedup obtained from this drastic increase in complexity will vary with the search. Megablast, and discontiguous megablast with the most common templates and a short-to-moderate size query, runs twice as fast now. For blastn the more common speedup is 10-15%.

RPSBLAST Optimizations

rpsblast is a special case of protein blast. We concatenate all the protein sequences in a small database. Each sequence has a position-specific score matrix, and both the concatenation of the score matrices and the lookup table derived from it are precomputed and stored on disk. Given the query to an rpsblast search, we scan the query and collect the hits against the database. Compare this to ordinary blastp, which would scan the sequences in the database and collect hits against the query. In rpsblast the lookup table is derived from the database, while in blastp the lookup table is derived from the query.

There are several optimizations in the current engine that significantly improve the performance of rpsblast. These optimizations depend on the fact that a single entry in the lookup table contains thousands of sequence offsets, scattered throughout all the protein sequences in the RPS database. These sequence offsets have no locality at all in the database, and so while it is possible to collect seeds in the same manner as ordinary blastp, the amount of ‘hopping around’ in the wordfinder will cause huge numbers of cache misses. We can reduce the number of cache misses by

- collecting a very large number of seeds in a single ScanSubject call

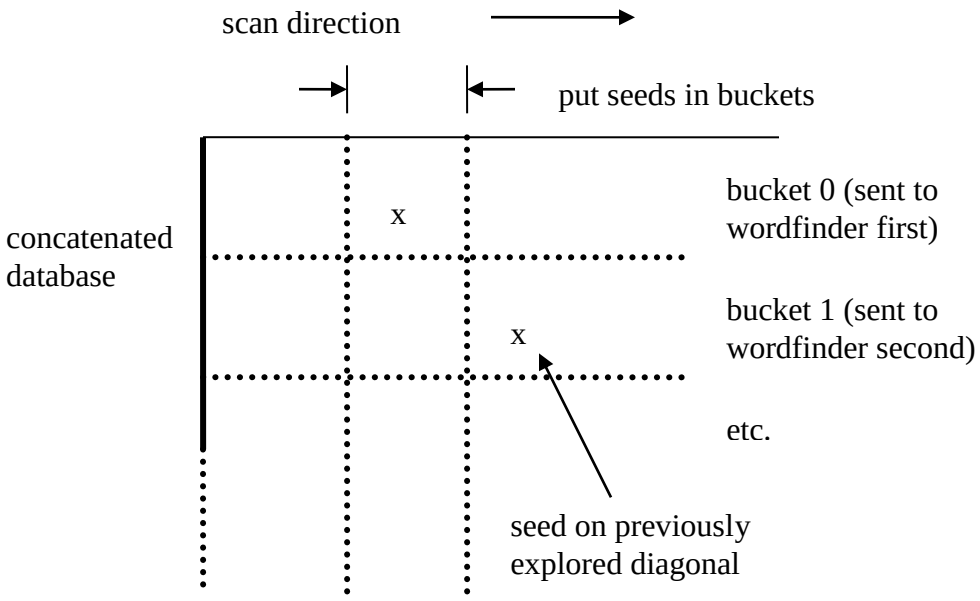
- organizing those seeds so that the protein wordfinder still works but the amount of locality is increased

Here ‘locality’ refers to the amount of clustering in the values of sequence offsets that appear in the list of offset pairs returned by BlastRPSScanSubject. The more locality in the list, the more often all of the relevant quantities (sequence data, score matrix columns, wordfinder data structures) will fit in cache and the higher the performance achieved. The number of seeds returned at once must be large so that when everything is loaded into cache the first time, it gets reused thoroughly before going on to the next batch of data.

Correct operation of the wordfinder requires that offsets into the sequence being scanned appear in increasing order. There is no restriction on the order in which the lookup table offsets appear. Optimal locality can be achieved by pulling a large number of seeds out of the lookup table and then sorting, first by query offset and then by subject offset. This is much too much work, since a call to qsort would involve dozens of passes through the list (when there are millions of seeds).

A better approach is to use one pass of a radix sort: we divide the RPS database into chunks of fixed power-of-two size, and for every seed that comes out of the lookup table we locate the chunk that corresponds to the database offset of the seed, and append the seed to a list (or ‘bucket’) of seeds located within the chunk. Later, once a large number of seeds have been collected, the wordfinder then operates on each chunk in turn, iterating linearly through the bucket for that chunk. This is implemented as an array of RPSBucket structures.

Changing the order of seed access in this way technically violates the requirement that scan offsets always are handled in increasing order. However, the real reason for the ordering restriction is that it guarantees that seeds falling on the same diagonal are processed in order of increasing distance down that diagonal. Proceeding a bucket at a time still does that: seeds on the same diagonal and inside one bucket are handled in the same order as before, while two seeds in different buckets but on the same diagonal still occur in order (since buckets are handled in order of increasing DB offset).



In real RPSBLAST searches, the concatenated DB is much larger than the sequence being scanned, so the graph above would be very ‘tall’ and ‘thin’. This means the number of diagonals (DB offset minus query offset) that are simultaneously active is limited. We want a large number of hits so that we accumulate a large number of scan offsets, and reuse the same range of DB offsets for as long as possible.

The performance difference is pretty dramatic: rpsblast is 25-50% faster than it used to be. Approximately half of the benefit comes from reusing data structures in the wordfinder (since the number of simultaneously active diagonals is limited), and the other half comes from needing only a small portion of the huge score matrix at a time, when calculating ungapped extensions.

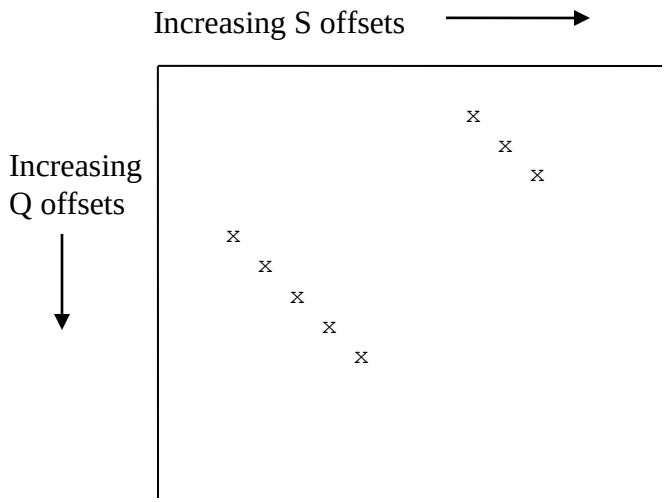
Currently a bucket is of size RPS_BUCKET_SIZE(=2048), and this produces enough locality to be fast without needing a huge number of buckets. As caches get smaller and RPS databases get larger, this granularity will become completely inappropriate. For a sense of scale, the largest RPS database has about 23000 sequences, and this is more than 200 times smaller than our largest protein database. Maintaining optimal performance while avoiding huge data structures will take further restructuring of the scanning code than is present now.

Blast Ungapped Alignment Phase

Once blast collects seed hits, the next phase is to convert a subset of these hits into ungapped alignments. Before the capability for gapped alignments was added, the majority of the work blast performs fell into this phase. The actual code has many more twists and turns than its description in the 1997 blast paper would indicate. The blast-program-specific subsystem that creates ungapped alignments is called the *wordfinder* for that blast program.

Terminology

It will be very helpful to visualize an ungapped alignment between a query sequence Q and a subject sequence S graphically. To that end, create a table with letters of Q labelling the rows and letters of S labeling the columns:



In the ‘dot plot’ above, ungapped alignments between Q and S are represented as diagonal strings of x’s. Such alignments are also called high-scoring pairs or high-scoring segment pairs (both are abbreviated ‘HSP’). The job of the wordfinder is to produce all of the HSPs between Q and S whose alignment score exceeds a cutoff value.

Any line in the dot plot where S[i] - Q[i] = c for all i is referred to as *diagonal* c. Diagonals extend from the top left of the plot to the bottom right. An ungapped alignment occurs on exactly one diagonal, and several ungapped alignments can share the same diagonal.

Ungapped Extensions

The input to all of the wordfinders is a stream of BlastOffsetPair structures from one of the various ScanSubject routines. The simplest protein ungapped wordfinder (not the default) is described in BlastAaWordFinder_OneHit and is called from the top-level routine BlastAaWordFinder (in aa_ungapped.c).

Each BlastOffsetPair gives the offsets into Q and S where a seed ungapped alignment (typically three letters long for protein searches) begins. For each such offset pair, BlastAaWordFinder_OneHit attempts to extend the seed alignment to the left and the right by some distance. The extension process uses an ‘X-drop’ algorithm: starting from the score of the seed, the running score of the alignment is kept. For extensions in one direction, if aligning the next letter of Q and S increases the running score above its previous best value, the alignment is enlarged to include that letter pair. If adding the next letter reduces the total alignment score by more than some constant X (an input to the algorithm) below the current best alignment score, the extension stops. If neither of these happens, extension continues but the alignment is not enlarged. The assumption is that once the running score dips below the X-drop threshold, it will never improve the alignment again. The larger X is, the longer the algorithm works before giving up.

More formally, the steps involved are as follows (see BlastAaExtendOneHit in aa_ungapped.c):

1. Given the initial seed hit, find the highest scoring subregion inside the seed. This process by itself is a tiny (the length of the seed) ungapped extension. The subregion so produced has 1, 2, or 3 contiguous letters for word size 3.

2. Set the running score to the score of the above subregion. Starting with the first letter pair to the left of the subregion, run the X-drop algorithm to extend to the left. Remember the maximum score encounterd and the offset of that maximum score.

3. Set the running score to the maximum from step 2. Starting with the first letter pair to the right of the subregion, run the X-drop algorithm to extend to the right. This time, extension stops if the X-drop test fails *or* if the running score ever drops to 0 or below. Remember the maximum score encountered and the offset of that maximum score.

Either or both of steps 2 and 3 can fail immediately, i.e. the extension fails on the first letter pair outside the region found in step 1. Step 1 is extremely important: if the left-hand letters of the seed are low-scoring, step 1 guarantees that the X-drop test will fail as soon as possible, while leaving a locally high running score to help the right extension in step 3.

The extra terminating condition in step 3 is a bit subtle. The left extension keeps going until the X-drop test fails, so the offset to the left really is the left endpoint of an optimal alignment. If the extension to the right reaches a running score of zero, but the optimal right-hand endpoint was beyond this, then you could have thrown away all of the alignment that lies to the left of the zero point (because the alignment left of the zero point would make no net contribution to the alignment right of the zero point). Left of the zero point could still be optimal, but the above shows that an optimal alignment will not include a zero point, and so step 3 lets you quit early (i.e. possibly before the X-drop test would actually have failed).

There is a fairly complicated asymmetry here. Extending left will give an optimal alignment that may or may not include the seed (if the score threshold $T \geq X$ and step 1 is performed then the seed is included in the left extension). If this extension turns out to be too long and a higher-scoring but shorter alignment is possible, later seeds that start farther to the right on the same diagonal will find that better alignment. However, extending too far to the right could prevent the wordfinder from ever considering the current portion of the diagonal again because of the optimization described next.

Reducing the Number of Extensions

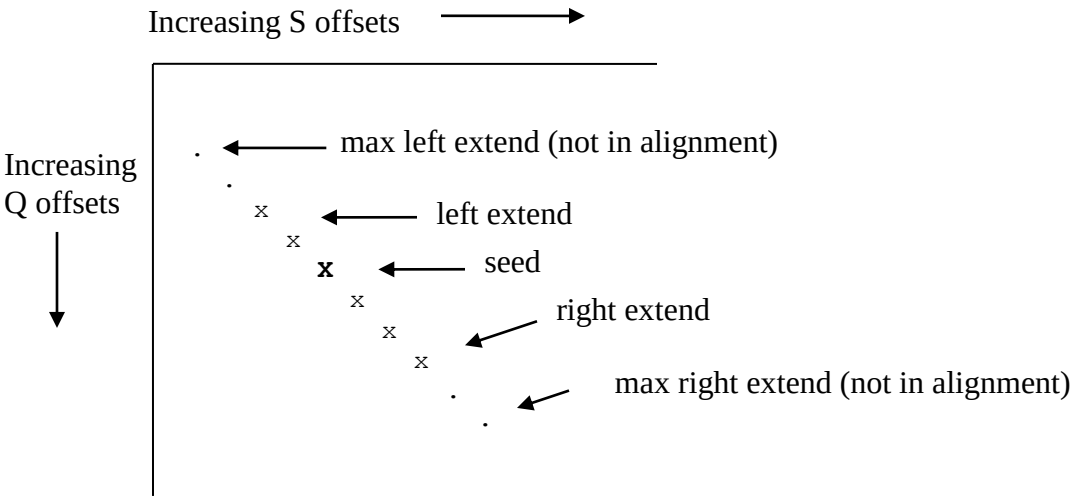
Once the range and the score of the ungapped extension is computed, if the score exceeds a cutoff value then the details of the extension are saved in a BlastInitHSP structure (blast_extend.h). The complete collection of ungapped alignments is kept in a BlastInitHitlist structure that is passed on for subsequent processing when all such ungapped alignments have been computed.

Even small ungapped alignments may contain clusters of seeds. For the BLOSUM62 score matrix, the following ungapped alignment:

```
Query: 6  ALWDYEPQNDDELPMKEGDCMTI 28
          AL+DY+ + ++++ +  GD +T+
Sbjct: 6  ALYDYKKEREEDIDLHLGDILTV 28
```

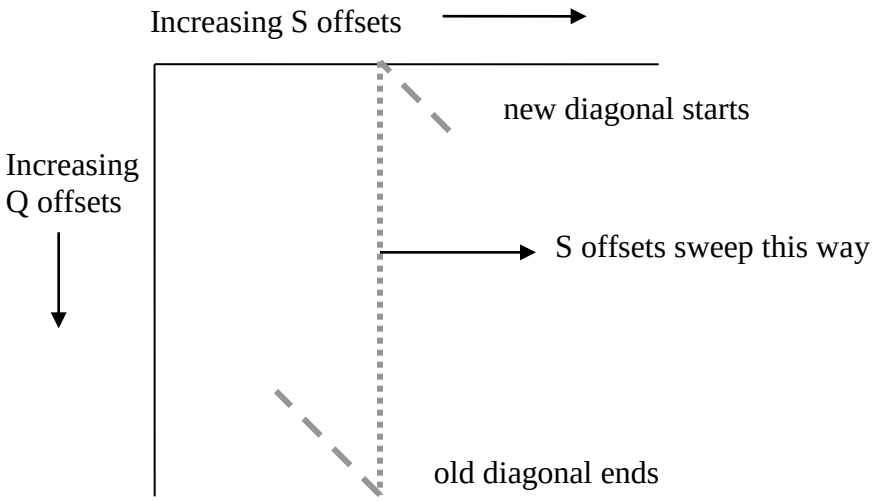
contains 4 triplets of letters whose combined score is 11 or more. If the extension procedure described above is performed on all 4 seeds, most likely all four extensions would produce the same ungapped alignment. This is wasteful, since three of the alignments would have to be thrown away later. To remove the potential for duplicate extensions when seeds are packed close together, a method is needed to remember when a previous alignment has already covered the neighborhood of Q and S that a given seed would explore if it was extended.

In general, when an ungapped extension happens the region of Q and S that is explored will exceed the size of the final alignment:



Since the entire extension uses the same diagonal, we can remember the largest ‘distance’ to the right that is traversed on this diagonal by this alignment, and if future seeds wind up on the same diagonal at a distance less than this number then we know they can be immediately discarded. Further, if seeds only occur on diagonals in order of increasing distance, then we don’t have to store the maximum extent explored by each alignment, only the maximum extent of the last alignment to hit this particular diagonal. In other words, last-hit information can occur on a per-diagonal rather than per-hit basis.

In a dot plot for Q and S, the number of distinct diagonals is (size of Q) + (size of S) - 1. However, if the list of (Q,S) offsets has the offsets into S arranged in ascending order (the ScanSubject routines work this way), then dealing with the distinct subject offsets is equivalent to sweeping a vertical line from left to right across the dot plot. When the vertical line moves to the right by one position, one diagonal becomes invalid (at the bottom of the plot) and another diagonal becomes active (at the top of the plot):



Hence the total number of diagonals active at one time is (size of Q).

Diagonal distance is tracked by an array of type DiagStruct. Given offset q_off into Q and offset s_off into S, the diagonal information for the pair is at offset (s_off - q_off) mod (size of Q) in the array. Within this entry, the DiagStruct::last_hit field contains the maximum subject offset of the last hit to this diagonal. If this number exceeds s_off then no ungapped extension is necessary. However, if s_off is larger, then an ungapped extension is performed with seed (q_off, s_off). The only difference is that the extension to the right saves the maximum S offset that was examined. Whether or not the resulting ungapped alignment scores high enough to be saved, last_hit is set to (max offset examined - (wordsize-1)). Any future seeds that lie on this diagonal, and whose subject offsets exceed this number, have at least one letter in the seed lying beyond the extent of the previous alignment on this diagonal.

A nice property of this method is that if the computation of diagonal entry puts the current seed on a diagonal that was just invalidated, the value of last_hit there is at most s_off-1 (because offset s_off is currently being processed). s_off will always exceed this number, so the ungapped extension will always be performed. This allows invalid diagonal structures to automatically be recycled for use by new diagonals.

Reducing overhead

One easy way to reduce the overhead of using diagonals is to make the total number of DiagStruct entries a power of two. More specifically, the number of entries should be the smallest power of two exceeding the size of the query sequence. This way, calculating the offset into the diag array only involves a mask operation rather than a full integer remainder.

Use of the diag array further requires that all last_hit fields be initialized to zero for each subject sequence. When the query is large (i.e. the diag array is large) and the database

contains many small sequences, the overhead of clearing the diag array becomes significant.

A trick allows avoiding this overhead almost all the time. When the last_hit field is updated, a bias is added to the new value. The bias is subtracted from last_hit before the comparison to s_off is made to determine whether an ungapped extension is needed. After the entire subject sequence has been processed, Blast_ExtendWordExit adds the length of the subject sequence to the bias. When the next sequence is being processed, the bias is larger than any of the leftover last_hit entries in the diag array, and so subtracting it from any last_hit will make new subject offsets automatically larger than all of the leftover last_hit entries from the previous sequence. In this way the recycling of diag entries happens not only within a subject sequence but also between different subject sequences. Clearing of the diag array is only necessary when the bias value threatens to cause integer overflow; when the bias exceeds 2^{30}, Blast_ExtendWordExit explicitly clears the diag array and sets the bias to zero.

All of the information concerning the diag array is packed into a BLAST_DiagTable structure (blast_extend.h). This includes the diag entries themselves, the array size, the mask for locating diagonals, and the bias (BLAST_DiagTable::offset). The BLAST_DiagTable is one of the options for remembering per-diagonal information, and is a member of the Blast_ExtendWord (blast_extend.h) structure.

2-Hit Ungapped Extensions

The ungapped extension algorithm previously described is sometimes called the '1-hit wordfinder', because a single (q_off, s_off) seed is all that is required to trigger ungapped extensions. Unfortunately, protein sequences can have many seed alignments embedded inside regions that are otherwise not similar at all. By 1997 blast was spending 90% of its time generating ungapped alignments that did not score high enough to exceed the cutoff score.

The 2-hit wordfinder only goes to the trouble of performing an ungapped extension if two seeds wind up on the same diagonal within a certain distance of each other. This is an attempt to only perform ungapped extensions that have a high likelihood of producing a good-scoring alignment, under the assumption that most good alignments will contain many seed hits packed close together on the same diagonal. The 1997 blast paper estimated that use of the 2-hit algorithm tripled the speed of the wordfinder. Recent internal studies on 'well behaved' biological sequences have shown that what is rare is not just two hits close together on the same diagonal, it is actually that there are two hits on the same diagonal at all!

The 2-hit algorithm for protein sequences is implemented in the high-level function BlastAaWordFinder_TwoHit, with the actual ungapped extension performed by BlastAaExtendTwoHit. Seeds that hit to the same diagonal and whose subject offsets differ by less than window_size (the default value of window_size depends on the score matrix used; it is 40 for BLOSUM62) will trigger an ungapped extension. The modifica-

tions to the 1-hit wordfinder are relatively straightforward, but there are corner cases that need special attention.

Ungapped extension code is passed the offsets of both hits. The assumption is that the hit with the lower subject offset occurred sometime in the past, and the second hit has the larger subject offset. This is guaranteed because seeds are processed in order of increasing subject offset. Extension to the left and the right happens from the second hit. In the description below, the first seed is the 'left hit' and the second seed is the 'right hit', since that's the order in which they appear on the diagonal.

The ungapped extension proceeds as follows.

1. Determine the highest-scoring prefix of the right hit, i.e. the highest-scoring group of letters that starts at the first letter of the hit

2. Starting from the first letter before the right hit, perform a left extension identical to that of the 1-hit algorithm

3. If the extension managed to reach the last letter of the left hit, then starting from the first letter after the right hit, perform a right extension identical to that of the 1-hit algorithm

Note that in the 2-hit case attempting the right extension is optional, whereas in the 1-hit wordfinder attempting it is mandatory. Step 1 therefore needs to build up the starting score as much as possible before the left extension begins, since if the left extension cannot go far enough then the right extension will not happen at all. Finally, the ungapped extension must remember whether extension to the right actually was attempted, and if so how far to the right it explored.

As with the 1-hit wordfinder, an array of DiagStruct structures tracks the maximum subject offset explored on each diagonal. As before, the last_hit field holds the subject offset of the last seed that hit on this diagonal. For the 2-hit wordfinder each DiagStruct also uses a 1-bit boolean that is set if the first of two hits to the diagonal has *not* been found, and is cleared if the second hit has not been found. last_hit has a bias added in, to avoid having to be reset for each subject sequence (see previous section).

The complete 2-hit algorithm is described below. The code is basically a state machine for each diagonal, with two states (first hit found / first hit not found) and some complicated conditions for transitioning between those two states. If an extension happens then the portion of the diagonal that it explored is remembered to avoid triggering future extensions; otherwise the algorithm relies on most seeds not having neighbors nearby on the same diagonal. This does mean that the 2-hit algorithm will miss some alignments that the 1-hit algorithm will find. To compensate for this reduced sensitivity, use of the 2-hit algorithm is accompanied by a slightly lower threshold for putting query offsets into the lookup table. This has the effect of increasing the number of seeds to deal with; the time that the 2-hit wordfinder saves in ungapped alignments avoided far outweighs the extra time spent throwing away an increased number of seed hits.

Some notes on this implementation:

- obviously this is much more complicated than the 1997 BLAST paper suggests, and is much more complex than the 1-hit wordfinder

- the old engine implementation of this algorithm is very different. Diag structures there have two words instead of one, and this lets the old engine find a small number of hits that this code does not find. The reason is that in rare cases, the old engine code allows an ungapped alignment to start when it mostly overlaps a previously found ungapped alignment. Having one diagonal offset instead of two does not provide enough information for the current implementation to mimic that behavior. Our tests have not shown any *gapped* alignments that have been missed as a result of this

- all of the booleans begin with arbitrary values when the first seeds arrive. This is okay; as long as (s_off + bias) always exceeds last_hit, the state machine initializes properly independent of the starting boolean value

Finally, the procedure for initializing and updating the diag array for a new subject sequence needs to change for the 2-hit wordfinder. Previously, the bias added to diag entries gets incremented by the length of the subject sequence, and when clearing the diag array (once per gigabyte of subject sequences) all the last_hit fields just need to be set to zero. The first item guarantees that diag entries get replaced when they are no longer valid. Both items together guarantee the first hit on a diagonal by seeds from a new subject sequence is automatically saved.

For the 2-hit wordfinder, last_hit does not get updated unless a hit is further down the diagonal than window_size from the previous hit. Therefore, if there is no previous hit then last_hit must be initialized to -window_size to guarantee that the first hit to a diagonal is automatically saved. If last_hit started as zero, then hits whose subject offset is between 0 and window_size-1 would never be saved, meaning that alignments may be erroneously skipped if they are near the beginning of the subject sequence. Similarly, the bias should start as window_size, so that it is (subject_length+window_size) when moving to the next subject sequence. Otherwise, if a previous hit has a subject offset near the end of the subject sequence and the next subject sequence hits that diag with an offset between 0 and window_size-1, last_hit will not get replaced like it should (i.e. stale information will remain in the diag entry). A single routine (Blast_ExtendWordExit) handles updating both wordfinders, since the 1-hit wordfinder is defined to have a window_size of zero.

```
for (each seed) {
   q_off = query offset of seed
   s_off = subject offset of seed
   diag = entry of diag array describing the diagonal on which the seed lies

   if (diag->flag == true) {
      /* first hit not found */
      if (s_off + bias < diag->last_hit) {
         /* a previous extension has already explored the neighborhood
            of the diagonal containing s_off; do nothing */
      } else {
         diag->last_hit = s_off + bias    /* first hit found */
         diag->flag = false;
      }

   } else {

      /* this could be the second hit */

      last_hit_d = s_off - diag->last_hit /* distance to last seed on diag */

      if (last_hit_d > window) {
         /* the distance to the last hit is too large; current
            hit becomes last hit without perfoming extension */
         diag->last_hit = s_off + bias
         continue;
      }

      if (last_hit_d < word_size) {
         /* current hit and first hit overlap; do nothing */
         continue;
      }

      /* current hit is inside the window; trigger an extension */

      perform ungapped extension from (q_off, s_off)

      if (alignment score >= cutoff_score) {
         /* save the alignment */
      }
      if (extension to right happened) {
         /* a big chunk of the diagonal was explored. Start looking
            for future first hits, and ignore seeds that do not extend
            at least one letter beyond the neighborhood already explored */
         diag->last_hit = max_right_extend - (word_size - 1) + bias
         diag->flag = true
      } else {
         diag->last_hit = s_off + bias    /* last hit = current hit */
      }
   }
}
```

Nucleotide Wordfinders

For all versions of nucleotide blast, the function responsible for converting seed hits into ungapped alignments is BlastNaWordFinder (na_ungapped.c). Many of the principles are the same compared to the protein wordfinder, but there are changes necessary because the sequences involved can be very large, the word size can be different from the number of

letters indexed in the lookup table, and the subject sequence is always in compressed form.

Nucleotide Ungapped Extensions

All of the different nucleotide routines rely on the same ungapped extension, performed by s_NuclUngappedExtendExact. It has a few differences from the protein ungapped extension algorithm:

- the subject sequence is assumed to be in compressed form, so that its letters must be extracted one at a time

- extension to the left begins one letter before the seed hit, but extension to the right begins at the first letter of the seed. Both extensions only use the X-drop test.

There is no shortcut for right extensions like protein ungapped code uses. This is mainly historical, but sometimes (i.e. in discontiguous megablast) the initial seed can contain mismatches, and so will have a low starting score that would make the extension risk terminating prematurely unless the full X-drop criterion was applied. For X values that are reasonable we can expect to at least achieve the highest-scoring prefix to the seed. Another reason to explore the seed itself during the right extension is that s_NuclUngappedExtendExact actually uses a nucleotide score matrix (rather than registering only matches and mismatches), and this is the first time that actual scores, that include ambiguities in the query sequence, are calculated for the alignment.

Approximate Nucleotide Ungapped Extensions

For blastn with default parameters, ungapped extensions do not take much of the total time. However, when the word size is small the time needed for ungapped extensions is a critical bottleneck. Such small searches involve far more seeds extracted from the lookup table, and the small size of these words means an extension is more likely to be triggered.

When ungapped nucleotide extensions are a bottleneck, the vast majority of them do not achieve a high-enough score to be saved. This suggests that an approximate ungapped extension that is faster could be used by default.If the approximate score achieved exceeds a more generous cutoff value then the full ungapped extension is computed.

The exact left extension schematically works as follows (the right extension is similar):

```
sum = 0;
while (letters left in both sequences) {
   subject_off -= 1
   query_off -= 1
   extract base 'subject_off' from the byte of the
                   compressed subject sequence containing it
   sum += matrix[q_base][s_base]

   if (sum > 0) {
      mark new alignment start point
      score += sum
      sum = 0;
   } else if (sum < X) {
      break;
   }
}
```

Note the multiple pointer decrements, use of a score matrix, and the pair of branches (the first of which is unpredictable) that happen for each letter in the extension. Reducing the time needed to compute the extension should involve removing as many of these operations as possible. Some simplifications can help:

- The approximate extension should only count matches and mismatches. This will overestimate the penalty associated with ambiguities, but allows precomputations that save time

- The extension may only start and end on a 4-base boundary of the subject sequence

These two simplifications allow an effective unrolling by four of the inner loop: one table lookup, one pair of comparisons, and one pair of base extractions for every four positions of the query and subject. The table lookup at iteration i really only needs to determine the score for the i^{th} group of four letter-letter alignments, which in turn only depends on how many of those 4 are matches and how many are mismatches. We can precompute this information with an 8-bit hashtable.

For example,

Query: in binary:	A A C G 00 00 01 10
Subject: in binary:	A C C A 00 01 01 00
Query XOR Subject	00 01 00 10 (index into hashtable)

In the above example, the table will contain the score for two matches and two mismatches at index 00010010. Since matches are interchangeable with each other and mismatches are similarly interchangeable, the modified extension code looks like:

```
sum = 0;
while (groups of 4 bases left in both sequences) {
   subject_off -= 4
   query_off -= 4
   pack next 4 query bases into q_byte
   sum += table[q_byte XOR s_byte]

   if (sum > 0) {
      mark new alignment start point
      score += sum
      sum = 0;
   } else if (sum < X) {
      break
   }
}
```

Especially when the match and mismatch penalties are close to each other in magnitude, the above will be much faster (up to four times faster) than the original extension.

What must the cutoff for the approximate extension be? It must be smaller than the ordinary ungapped cutoff, but making it too small will cause the exact ungapped alignment to run too often. Empirically, setting it to floor(60% of the ungapped cutoff) is a reasonable compromise: the vast majority of exact ungapped extensions are avoided, and I have yet to see an ungapped alignment that is missed. It's possible that the cutoff should instead be represented as the ordinary cutoff plus the score for a few mismatches, since most of the time the approximate ungapped alignment will look like the exact alignment with some spurious letters to either side. However, when the ordinary cutoff is somewhat small both these schemes lead to approximate cutoffs that are about the same.

The table for approximate ungapped extension is computed in BlastInitialWordParametersNew, and the extension itself happens in s_NuclUngappedExtend. This is the only entry point for nucleotide ungapped extensions.

For blastn with default parameters the runtime difference is essentially zero; however, as the word size goes down and the match/mismatch penalties converge to each other, the speedup is significant (30-50%). One side effect of using approximate extensions is that the generated alignment will not extend to the right as far as it used to on average, which limits the amount of pruning of future seed alignments that is possible. Hence the number of ungapped extensions performed increases slightly (~5-8%), but the slowdown from this is more than offset by the speedup from the extensions themselves being faster.

Extending Nucleotide Seeds

The complexity in the nucleotide wordfinders actually has more to do with the seeds returned from the lookup tables, or equivalently deciding when to call s_NuclUngappedExtend and whether preprocessing of the seed must happen first. Blast-

NaWordFinder runs the previously determined ScanSubjectroutine on the subject sequence as many times as is needed to scan the whole subject. For each of the seed hits returned (they come in batches), we must determine if the seed belongs to a run of exact matches equal to at least the blast wordsize in length. The routine that does that varies from search to search, and is chosen by BlastChooseNaExtend(na_ungapped.c) when the search begins, using information from the lookup table. The choice of 'exact match extension' routine is stored as an opaque pointer within the lookup table structure.

The extension process here is not very elaborate; the objective is to determine as quickly as possible if a seed alignment lies in a field of sufficiently many exact matches. Performance is more important here than with the complete ungapped extension, since if the lookup table width does not match the blast word size then every single seed alignment must undergo this exact match extension.

There are several exact match extension functions to choose from (all are in na_ungapped.c):

- s_BlastNaExtendDirect performs no exact match extension at all, and merely forwards every seed to the ordinary ungapped extension routines. This is used when the lookup table width matches the word size, and applies to any lookup table type

- s_BlastNaExtend is a generic routine that performs exact match extensions assuming the lookup table is of type BlastNaLookupTable or BlastMBLookupTable

- s_BlastNaExtendAligned works like s_BlastNaExtend, except that it assumes the number of bases in a lookup table word, and the start offset of each hit, is a multiple of 4

- s_BlastSmallNaExtend performs the exact match extension process assuming the lookup table is of type BlastSmallNaLookupTable. This routine and the next use special precomputations that make them much faster than their counterparts above

- s_BlastSmallNaExtendAlignedOneByte works like s_BlastSmallNaExtend, but assumes the number of bases in a lookup table word and the start offset of each hit is a multiple of 4, and also that the word size is within 4 bases of the lookup table width. This allows the routine to omit loop and function overhead that the previous routine requires, and makes it the fastest of all the routines that perform exact match extensions. Happily, in practice this routine is what blastn with default parameters will use for the vast majority of searches

Regular Exact Match Extensions

The exact match extension first occurs to the left of the seed and stops at the first mismatch between the query and subject sequences. Ambiguities in the query sequence are always treated as mismatches. When the left extension is complete, if more exact matches

are needed to achieve the blastn word size then a right extension is performed, starting from the end of the seed. The seed itself is always assumed to contain only exact matches. While this is not true for discontiguous megablast, in that case the lookup table width matches the word size of the search, so that exact match extensions are not necessary at all.

Because the subject sequence is compressed, it is technically possible to compare letters in batches of four when performing these extensions to the left and right, then on the first mismatch to go back and compare one letter at a time. However, this will slow down the common case: since the nucleotide alphabet has four letters, the first letter pair beyond the seed will be a mismatch ¾ of the time (assuming the sequences are random). Thus most of the time an extension will encounter a mismatch very quickly, and the speed can be improved if we always compare one letter at a time since this has much less overhead. Concentrating on reduction of overhead rather than processing letters in parallel makes the exact match phase run more than twice as fast on average. Especially for searches with small word size, where it is customary to find a huge number of seeds, fast exact match extensions are essential.

Small-Query Exact Match Extensions

Most queries are small, and if exact match extensions proceed one base at a time, then the previous routines would still require several loop iterations to complete the exact match extension. Typically the blast word size is 11 and the lookup table has width 8, so that 2-3 letters must be tested. If the subject sequence was compressed by four, then 1-2 table lookups would be sufficient to complete the extension, and would be much faster. This doesn't invalidate the conclusions of the previous section: compressing the query sequence on the fly adds unacceptable overhead, but if the query was compressed once, at the beginning of the search, then using table lookups to determine the number of additional exact matches removes setup and loop overhead, allowing a net savings. When the lookup table is of type BlastSmallNaLookupTable, we can be assured that the query is small, so it's feasible to compress the query.

The SmallNa exact match extension routines take advantage of this preprocessing (first published by Cameron and Williams). When the query is first created, a call to BlastCompressBlastnaSequence compresses it 4-to-1. The compressed array has one byte for every query position, plus three extra bytes at the beginning. The compressed array is set up so that offset i of the array gives the packed representation of bases i to i+3, and iterating through the query amounts to iterating through the compressed array with stride 4. This fourfold redundancy means that all groups of 4 bases within the query are aligned on a byte boundary, so that a seed hit can start at any query offset and pull groups of 4 bases from the compressed array without having to realign them. The handling of bases at the start or end of the query is special; for query length L, the compressed array is of length $C = L + 3$, and the start of the array is a pointer to offset 3 of C.

- offset -3 gives the first base, right-justified in a byte
- offset -2 gives the first 2 bases, right-justified in a byte
- offset -1 gives the first 3 bases, right-justified in a byte

- offset i, $0 \leq i < C - 4$, of the compressed array gives the 4 bases starting at offset i of the query, packed together
- offset $C - 3$ gives the last 3 bases, left-justified in a byte
- offset $C - 2$ gives the last 2 bases, left-justified in a byte
- offset $C - 1$ gives the last base, left-justified in a byte

Queries of length less than 4 bases have the same representation, except that nonexistent bases have a value of 0 in the first three compressed entries, and are not represented at all in the last entries of the compressed array.

With the compressed array in place, we need two 256-entry tables, for extensions to the left and right. Entry i of the left extension table contains floor(number of trailing zero bits in the binary representation of i) / 2 and entry i of the right extension table does the same but with leading zero bits. With these precomputed quantites, the number of exact matches to the left of a seed hit at offset (q,s) on the query and subject sequences is

```
min(q, s, bases_allowed,
    left_table[compressed_query[q-4] XOR subject[s-1]] )
```

assuming q and s are positive (note that the format of the compressed query puts valid data at offset q-4 when this is negative). A similar but more tedious 4-term min applies to the right extension, if it turns out to be necessary. The streamlined extension process allows an overall speedup approaching 15% for blastn with default parameters and a small query.

s_BlastSmallNaExtend reuses the compressed query to perform arbitrary-size exact match extensions. Unfortunately, this is only slightly faster than the base-at-a-time implementation, for reasons detailed in the previous section, and also because the extension process cannot skip over the entire seed (bytes of the subject sequence must remain aligned). One future item to do is to allow use of the compressed query for the full ungapped extension. This can provide a big speedup for blastn with a small query and word size 7, which is somewhat popular and where 30% of the time is spent performing ungapped extensions.

From Seeds To Alignments

Whether or not a seed is extended out to the full word size expected, at some point an attempt may be made to extend that seed into a true ungapped alignment that may contain mismatches and uses a conventional X-drop algorithm. There are two routines that perform this function, s_BlastnDiagHashExtendInitialHit and s_BlastnDiagExtendInitialHit (in na_ungapped.c). Each calls s_NuclUngappedExtend as necessary.

Like their protein sequence counterparts, the nucleotide wordfinders use auxiliary data structures to throw away seeds that will just repeat the work of previous ungapped alignments. s_BlastnDiagExtendInitialHit use an array of DiagStruct structures to track the furthest extent of the last hit on each diagonal. Each DiagStruct has a 31-bit field for the last offset on that diagonal and a 1-bit boolean that defines what the 31-bit field means. If the top bit in a DiagStruct is set, it means that the lower 31 bits is the offset of the first hit

(out of 2 that the 2-hit algorithm would need). If the top bit is clear then the lower 31 bits give the maximum extent of ungapped alignments on that diagonal, which helps in removing seed hits that occur in regions of the diagonal that have already been explored. The data structures needed for this are stored in a BLAST_DiagTable.

Even with only four bytes per diagonal, a diag array can consume excessive amounts of memory when the query is large. Further, if long alignments with high identity are expected then it is typical to get long runs of seeds that hit to the same diagonal. This implies that even with a full diag array available, only a few of the entries would ever be used at a given time. Under these circumstances we can save the memory of a diag array at the expense of a small increase in runtime by using a hashtable of diag entries rather than a random-access data structure. s_BlastnDiagHashExtendInitialHit implements this improvement.

The hashtable is implemented as an array of DIAGHASH_NUM_BUCKET(=512) integers, acting as a front end for an array of initially DIAGHASH_CHAIN_LENGTH (=256) DiagHashCell structures. These basically wrap an ordinary DiagStruct along with a next pointer and a word to tell which diagonal the struct is for. Each hash bucket stores the offset of a linked list of diagonals that hash to that bucket, and all the linked lists are packed together in the ‘chain’ field of a BLAST_DiagHash struct. This behavior is very similar to the way the megablast lookup table works.

Processing happens one seed at a time; for each seed, whether extended to the full word size or not, the hashtable entry that contains the corresponding diagonal is first located in s_BlastDiagHashRetrieve. If the entry is not found, a new entry is added to the list for that has bucket. If the entry is found, the 1-hit or 2-hit algorithm proceeds using that entry. This layout makes it easy to increase the number of diagonals stored. A final optimization is that like the protein wordfinder, the hashtable does not have to be cleared when switching from one database sequence to the next. Both the diag array and the hashtable implementations maintain a bias (the ‘offset’ field in those structures) that is added to the computed values of diagonal distance, and the bias can be used to tell when a diagonal has ‘gone stale’ and its corresponding diag struct can be recycled for use by another diagonal.

A final wrinkle in the use of ungapped extensions is that all of them are optional. In particular, discontiguous megablast is configured by default to skip ungapped extensions entirely, so that all of the diag machinery works but seeds that survive are simply saved in a BlastInitHSP structure directly. Discontiguous megablast in this configuration forwards lookup table hits directly to the subsystem for gapped extensions. This effectively turns off all of the diag machinery used for ungapped extensions, since lookup table hits will only be pruned if they lie a distance larger than (scanning stride) away from previous lookup table hits on the same diagonal. When there are clusters of seeds this still can amount to a lot of pruning, although this process only works well with 2-hit extensions, to reduce the number of seeds that are forwarded.

Blast Gapped Alignment Phase

Once a collection of ungapped alignments has been found, the next step is to turn a subset of those alignments into gapped alignments, then to keep the highest-scoring subset of those. Blast did not have gapped alignments in mind when it was originally written, and we've expended a significant amount of work to make the current engine handle gapped alignments in a clear and flexible way.

Gapped alignments are handled in two distinct phases. The first is a score-only phase, which only produces the endpoints and the score of each gapped alignment. The more expensive traceback phase actually recomputes all of the work performed in the score-only phase, with different parameters that cause the code to expend more work to produce a better alignment. This time, however, the alignment code generates enough bookkeeping information to allow later stages to reconstruct each alignment in full. blast_gapalign.c contains all of the low-level alignment routines and the top-level routines for the score-only phase, while blast_traceback.c contains the top-level routines for the traceback gapped alignment phase.

The end result of the gapped phase, both score-only and traceback, is a BlastHSPList (blast_hits.h) containing among other things a collection of BlastHSP structures. At the end of the score-only and traceback phases, the HSPs within the BlastHSPList are sorted in order of decreasing alignment score (i.e. in order of decreasing BlastHSP::score). A BlastHSPList contains all of the alignments between one query sequence and one database sequence.

Score-Only Gapped Alignment

The input to the score-only phase is a BlastInitHitList (blast_extend.h) containing a list of ungapped alignments. These may just be hits from the lookup table that are forwarded to the gapped phase. It's expected (though not necessary) that each BlastInitHSP structure within the BlastInitHitList contain a filled-in ungapped_data field, which gives the extent and score of the corresponding ungapped alignment. Any gapped alignments performed must begin from one of these ungapped alignments. The ungapped alignments in the list are sorted in order of decreasing score.

The main routine controlling gapped alignments for all blast searches except phiblast is BLAST_GetGappedScore (blast_gapalign.c), while the routine for computing gapped alignments with traceback is Blast_TracebackFromHSPList (blast_traceback.c). Because gapped alignments are expensive, these routines go to a lot of trouble to avoid computing them unless absolutely necessary. The procedure is as follows:

1. The main alignment procedure:

```
for (ungapped alignment i) {

   -  locate the one query context, in the concatenated set of queries,
      that contains alignment i; change the query offsets of alignment
      i to be local to this context. If any previously saved gapped
      alignments completely envelop alignment i, delete alignment i;
      the assumption is that the previous alignment was optimal, and
      alignment i would not add new information if extended

   -  call BlastGetStartForGappedAlignment, which finds the highest-
      scoring group of HSP_MAX_WINDOW (=11) contiguous letters in
      alignment i. The middle of this region is the start point for the
      gapped alignment. Note that this does not happen for nucleotide
      searches; in that case, the start point for the ungapped align-
      ment is assumed to be a group of at least four exact matches
      between query and subject sequences (the original seed hit).
      There is no 'better' or 'worse' start point within the seed hit;
      for greedy nucleotide alignment the seed is used directly, while
      for non-greedy gapped alignment, the start point is the first
      byte within the match region that contains only exact matches

   -  (traceback only) call AdjustSubjectRange. If the subject sequence
      is smaller than MAX_SUBJECT_OFFSET(=90000) letters, this does
      nothing. For larger subject sequences, this routine artificially
      limits the range of the subject sequence that will be seen by the
      gapped alignment code. The limitation is not that strict: exten-
      sion to the right is limited to the distance to the end of the
      query sequence from the query start point, plus MAX_TOTAL_GAPS
      (=3000) letters. Extension to the left is similarly limited. This
      basically assumes that no gapped alignment will have more than
      2*MAX_TOTAL_GAPS gaps in the query sequence. Of course if the
      query is also large the range becomes the whole subject sequence
      instead.

      It's not clear that this measure is needed anymore; we do not ac-
      tually retrieve partial subject sequences from blast databases,
      and about the only advantage to having it is that when the query
      is small it places an upper bound on the size of the auxiliary
      structures used in the gapped alignment. Because these structures
      now grow dynamically, a limit probably isn't necessary, but then
      again it's not performance critical and it's not much of a limit.

   -  convert alignment i into a gapped alignment. If the resulting
      score exceeds a cutoff, save the gapped alignment in a BlastHSP
      structure
}
```

2. Once all of the HSPs are computed, delete any HSPs what share a starting or ending (query,subject) pair with a higher-scoring HSP. The routine that does this is Blast_HSPListPurgeHSPsWithCommonEndpoints (blast_hits.c)

3. Sort the surviving HSPs by score before they are forwarded to the next phase

For score-only alignments, steps 2 and 3 happen in blast_engine.c

The second step in the loop, testing to see whether alignments are contained within other alignments, is performed using a binary-tree-like data structure called an interval tree, described in Appendix B. For protein searches, ungapped HSPs are discarded if the query range and subject range are both enveloped by the query and subject ranges of a previously found gapped alignment. For contiguous megablast, the containment tests are a little more subtle: ungapped alignments may lie inside the 'box' formed by a higher-scoring previous gapped alignment, but must lie more than MB_DIAG_CLOSE (=6) diagonals away from that gapped alignment (see the MB_HSP_CLOSE macro of blast_gapalign_priv.h).

There are two reasons for this difference; first, gapped alignments found by megablast can be very large, and the greedy nature of most megablast gapped alignments means that smaller gapped alignments can often cluster around interesting regions of the main, 'big' alignment. Second, since most gapped alignments have few actual gaps, older versions of this code implemented containment tests by sorting the list of gapped HSPs in order of increasing diagonal and limiting the search to HSPs up to MB_DIAG_CLOSE diagonals away. This was much faster than the alternative, testing every ungapped HSP against all previously computed gapped alignments. Using an interval tree for this application allows the two kinds of containment tests to be unified, and for very large lists of alignments is much faster still.

A note about GetStartForGappedAlignment: when this routine computes the start point for the gapped alignment, that start point is saved in the HSP 'gapped_start' field. The assumption is that the traceback version of the alignment is also justified in starting at this point. For protein blast this is safe to assume, but for nucleotide blast it's only an approximation. The problem is that during the score-only phase, the subject sequence is compressed for nucleotide blast, so we don't know where ambiguities may be in the alignment. These ambiguities may adversely affect the score at the point where the gapped alignment is chosen to start. Thus, the traceback phase first calls BLAST_CheckStartForGappedAlignment, which verifies that when ambiguities are present in the subject sequence, the previously saved start point for the alignment still has positive score in a region of size HSP_MAX_WINDOW centered at the start point.

If this is not the case, then we're stuck; only the boundaries of the alignment are known, so there's really no a priori good way to get a good alignment start point. Currently the code calls BlastGetStartForGappedAlignment from the left end of the gapped alignment. This isn't a bad idea, since a good local alignment will not have many gaps and will start and end with a long run of matches. A long run of matches will at least mean that the traceback alignment will start in a locally good region. It's unlikely that the region will have the optimal score, though, since gaps later in the alignment will move it to a different diagonal that could be better scoring. There may be better ideas to solve this problem, but the current procedure works well enough and blast has never really been obsessed with always finding the best possible alignments.

The low-level routines also fill in an input boolean called fence_hit. This represents a hook for the fetching of partial database sequences. The C++ toolkit contains an advanced library ('SeqDB') for accessing sequences from blast databases, and in certain applications (like binaries that generate traceback from previously computed blast results)

the extents of all the alignments going into the traceback phase of blast are known. In this case, memory is allocated for database sequences but only the portions that will be used in the traceback phase are actually initialized. The problem is that traceback alignments can become larger than when they started, and there's no way to know how large they will get.

We work around this problem by assuming that all alignments will become X bases larger at most, then putting 'fence' bytes at the boundaries where initialized letters end inside the database sequence. If the traceback gapped alignment routines encounter this fence byte, the boolean is set to true and the entire traceback process fails for this DB sequence. Calling code then retrieves the entire database sequence and reruns the traceback. When the database contains very large sequences, this makes the traceback generation process dramatically faster (5-10x), and since traceback ordinarily consumes a very small fraction of the total runtime, putting a test for the fence byte in the innermost loop does not noticeably reduce throughput.

Low-Level Gapped Routines

The following covers the low-level dynamic programming routines, located in blast_gapalign.c.

Gapped alignments may be computed from protein or nucleotide ungapped HSPs. For translated protein searches, routines are also available for computing out-of-frame gapped alignments. Gapped alignments for all protein searches call s_BlastProtGappedAlignment, and non-greedy nucleotide gapped alignments begin with s_BlastDynProgNtGappedAlignment. Both of these routines set up the sequence offsets that describe where the gapped alignment should start, and also handle collection of the ending offsets and scores. Each gapped alignment is composed of a left extension and a right extension; the alignment score is the sum of the scores for these two extensions. Obviously, if the start point is at the boundary of one sequence or the other then a gapped extension cannot begin in that direction.

All gapped extensions, whether to left or right, compute the offset into both sequences where the extension ends, and the score of that extension. They can optionally also produce the complete traceback for the alignment. Alignment extents and scores fill in the fields of a BlastGapAlignStruct (blast_gapalign.h), from which calling code disperses the information to other structures.

There are five main low-level gapped routines:

- ALIGN_EX performs gapped alignment of two protein sequences, or two nucleotide sequences in ncbina format. Computes the alignment traceback

- Blast_SemiGappedAlign performs a gapped alignment of two protein sequences
- s_OutOfFrameAlignWithTraceback performs a gapped alignment of a protein sequence with an out-of-frame protein sequence, and computes the alignment traceback

- s_OutOfFrameGappedAlign performs a gapped alignment of a protein sequence with an out-of-frame protein sequence

- s_BlastAlignPackedNucl performs a gapped alignment of a nucleotide sequence in blastna format with a packed nucleotide sequence in ncbi2na format.

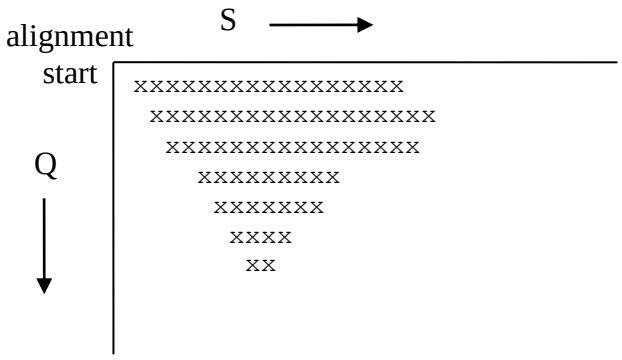
All of these routines perform extensions to either the right or to the left. blastn and discontiguous megablast searches use s_BlastAlignPackedNucl for the score-only gapped alignment and ALIGN_EX for the traceback alignment; by the time traceback happens, all sequences are in one-letter-per-byte form, and this is all that ALIGN_EX really needs (along with a suitable score matrix). The out-of-frame routines always assume that the second input sequence is the out-of-frame one. This means that for blastx searches calling code has to swap the inputs. There is no out-of-frame routine that allows frame shifts in both sequences; this would be much more computationally intensive, and it's not clear that the extra alignments found by 'OOF tblastx' would actually be useful. Finally, the first two routines are not static because the composition-based statistics code in blast_kappa.c uses them directly.

The low-level details of how these routines work is beyond the scope of this document. Some highlights of why they look the way they do can help understand them, though.

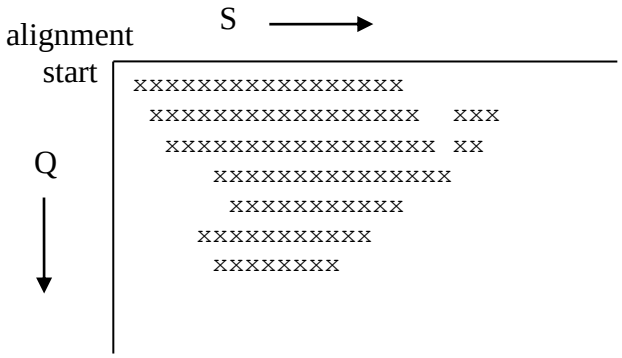
The in-frame routines are variants of ordinary Needleman-Wunsch dynamic programming. s_BlastAlignPackedNucl is essentially identical to Blast_SemiGappedAlign, except that the first input sequence is unpacked letter by letter as the extension takes place. There are several variations in these routines that are unusual:

- All the gapped routines incorporate an X-drop criterion. The dynamic programming lattice is filled in row by row, and if the best possible score in a given lattice cell is less than some number X below the best score seen thus far, the best score at the cell is set to -infinity (guaranteeing that the final alignment will not pass through it).

- This means that a given row can only explore a few cells beyond the extent of the previous row; it also means that if the first cells in the row fail the X-drop test, the next row will begin some distance past the current row. Hence the portion of the search space that the gapped extension (in one direction) examines usually looks like the following (this corresponds to a gapped extension to the right of the start point):



There is a very unintuitive side effect to the X-drop test when implemented this way. If the region that does not fail the X-drop test instead looks like



i.e. has ‘horns’ or concave regions, then the horn in the second and third row gets explored (because the number of cells to examine is determined before the row starts) but the horn in the last two rows is ignored because the starting cell of a row always increases monotonically. In general, this means that a gapped alignment of sequence A and B may not turn out to be equivalent (even in score) to a gapped alignment of sequence B and A, despite having the same start point.

- The factor that adds the most complexity to the code boils down to the fact that the input sequences can be very large. Most Needleman-Wunsch implementations assume that at least a linear array of per-letter structures is acceptable, but these routines cannot. If one sequence has 10 million nucleotides, then a traceback gapped alignment would need hundreds of megabytes for the auxiliary structures needed during the dynamic programming, even if the actual alignment was only 100 letters long and needed a few kilobytes for the actual traceback. Hence all of the auxiliary structures are allocated dynamically, and only increase in size if the alignment demands it. The array of auxiliary structures is reused from subject sequence to subject sequence. For traceback alignments, the space allocated for the auxiliary data needed to reconstruct the traceback uses a separate memory manager, since many small allocations and no deallocations are expected. The same considerations apply to protein sequences, even if most pro-

tein sequences are small: a translated search can turn a giant nucleotide sequence into several giant protein sequences.

Low-Level Greedy Routines

For contiguous megablast, alignments are expected to be very large but to contain few gaps. For that situation an ordinary dynamic programming gapped alignment will do too much work, since it makes no assumptions about how gap-filled the optimal alignment is. For this type of problem, blast uses specialized routines located in greedy_align.c that attempt to skip over large portions of the ordinary dynamic programming search space in an attempt to build an optimal gapped alignment greedily. These routines are called from BLAST_GreedyGappedAlignment (blast_gapalign.c) and only care about matches, mismatches and gaps; X-dropoff criteria are still used, and the methods are parametrized to perform a configurable amount of work looking for an optimal alignment.

A low-level understanding of how the greedy routines work would require a careful reading of the paper that originally described them, as well as of the code in the current implementation. Briefly, ordinary dynamic programming uses a table that corresponds to aligning a portion of both sequences; each cell of the table gives the score of an optimal path ending at that cell. Greedy methods instead measure an alignment in terms of a 'distance', i.e. the combined number of gaps and of mismatches that occur on that path. Basically the routines assume that an alignment score depends only on its length.

The table for a greedy algorithm assigns distances to each row and a *diagonal* to each column. Each cell in this table contains not a score, but the maximum offset into one of the sequences that lies on the diagonal and achieves the distance described by the cell. The larger the offset, the more matches occur in the alignment and the higher its score (i.e. for fixed distance and diagonal, how many matches can be achieved?). The greedy algorithm examines each distance in turn, and stops when any larger distances would fail an X-dropoff test, or at any rate when a maximum distance has been processed. For each distance, it then iterates through all of the diagonals that were valid in the previous distance, looking for the diagonal that can achieve the current distance with the largest number of matches. This is a 3-term recurrence relation that looks a lot like ordinary dynamic programming. The number of diagonals examined can increase or decrease when examining the next distance. For each distance, the score of the best path is saved separately; a traceback procedure can use the table of sequence offsets to create a list of edit operations that recover the optimal alignment.

The number of distances examined when aligning sequences A and B is either GREEDY_MAX_COST (=1000) or ((length of B) / 2 + 1), whichever is smaller. In the implementation the maximum number of diagonals to examine can only increase by at most 2 for each distance, so this also caps the maximum number of diagonals. Roughly speaking, no matter how large the input sequences are, the alignment produced will never venture more than a certain number of mismatches/gaps away from 'all matches'. Unlike a 'banded aligner', which has the same restriction, the greedy aligner does not need to examine every combination of sequence offsets that both lies in the band and passes the X-drop test.

The paper describing the algorithm claimed order-of-magnitude speedups as a result; my personal opinion is that the big advantage of the greedy routines is not performance (for ordinary alignments they do not appear to be materially faster than the ordinary gapped routines we now use) but memory efficiency. The central idea of the greedy alignment routines is that they implement a runlength encoding of the ordinary dynamic programming lattice; if your gapped alignment is a million letters long but only has three actual gaps, the greedy routines only have to store perhaps a few hundred states.

There are two species of greedy alignment supported, one with affine and the other with non-affine (linear) gap costs. BLAST_GreedyAlign implements the linear version; this requires gaps to have a constant per-letter penalty of exactly (mismatch_score - match_score / 2). This restriction lets the aligner get away with less than 1/3 the memory consumption and arithmetic of the affine aligner, but the alignments it produces are most suited to cases where the only reason to have a gap is because of sequencing errors, i.e. the alignments are not expected to be biologically plausible. The affine greedy code is in BLAST_AffineGreedyAlign, and here the match, mismatch, gap open and gap extension penalties are arbitrary. Zero gap penalties are a signal to the affine routine that the non-affine routine is to be used; the non-affine routine is never called directly. Note that the standard paper that describes greedy alignment only goes into detail about the case of linear gap penalties.

Unlike the ordinary dynamic programming alignment code, there is no separate version of the affine and non-affine greedy routines that calculates traceback. The traceback can optionally be computed right after the alignment bounds are determined. In the current engine, traceback is performed in a separate call to the greedy aligners, with a more generous X-drop criterion (and ambiguities in the subject sequence represented faithfully). This has the nice advantage of stitching together many small alignments into a single big one when the underlying neighborhoods of the query and subject sequence are very similar to each other. It also allows gapping across stretches of ambiguities that would have split apart a score-only version of the alignment.

Occaisionally, when ambiguities are restored they invalidate the start point that is computed during score-only greedy alignment. This usually means the traceback will start from a suboptimal point and generate spurious gaps to rejoin the otherwise optimal alignment. To make this less likely, the low-level greedy routines remember the longest run of exact matches encountered during the gapped extension, and return that information in an SGreedySeed (greedy_align.h) structure. When score-only alignment completes, the alignment start point is chosen as the middle of this region. Recovering the longest run information is easy because the greedy recurrence relations need the same information anyway.

Semi-Gapped Alignment

For nucleotide blast, the time needed for gapped alignment is usually not that significant. However, for protein blast experiments show that gapped alignment takes a fairly consistent 25-30% of the total runtime, and the vast majority of that time goes into the score-only gapped alignment phase. Cameron/Williams/Cannane from RMIT developed a clev-

er method for reducing the score-only gapped alignment time, and a modification of this scheme is implemented in the NCBI blast code.

Ordinary gapped alignment computes a 3-term recurrence relation for every cell of the dynamic programming lattice. The central idea with the modified scheme is that if gaps are only allowed to start at specific offsets in the sequences being aligned, then most cells in the lattice can be computed using a 1-term or (occasionally) 2-term recurrence. The drawback is that because gaps may not be allowed to start at the point needed to achieve an optimal alignment score, the score from this 'semi-gapped' alignment may be worse than the true optimal score one is entitled to. CWC's idea was to first perform a semi-gapped procedure on each ungapped alignment that needs extending. If the score achieved exceeds the gapped cutoff score, then the alignment bounds and the score are kept. If the score cannot exceed a reduced cutoff score (set to 68% of the gapped cutoff), the alignment is thrown away. If the alignment achieves a score between these two extremes, it is recomputed using the full optimal score-only alignment.

A very nice advantage to this scheme is that as the cutoff for the search increases, more and more gapped alignments are computed using the semi-gapped algotithm, i.e. the optimal score-only routine is needed less and less often and performance increases. Without semi-gapped alignment, performance increases a little but that's only because fewer *ungapped* alignments make the more stringent ungapped cutoff to begin with. Semi-gapped alignment gives a boost over and above this.

Semi-gapped alignment is performed by s_RestrictedGappedAlign. This routine is approximately five times faster than Blast_SemiGappedAlign, since its inner loops are much simpler most of the time. Implementing the full CWC scheme speeds up blastp by about 5% with an e-value cutoff of 10.0; as the cutoff becomes more stringent, the speedup climbs to about 15% asymptotically. With the semi-gapped routine being five times faster, the largest speedup we can expect is about 22%, so the performance behaves about as expected.

Unfortunately, implementing CWC's scheme as described causes huge differences in the output. If all the hits expected to be found are high-scoring, the differences are minor; but if some low-scoring hits are expected, many of them are missed. More importantly, some hits (including high-scoring hits) do appear in the output but end up scoring lower and looking worse, with big gapped regions that did not originally appear.

Each score-only gapped alignment has a start point, and the traceback phase reuses that start point. Starting the traceback alignment from a suboptimal region will produce a lower-scoring final alignment, and the differences come about because some alignments are allowed to graduate to the traceback phase that previously did not do so. These alignments carry their suboptimal start point with them; the reason is that CWC's scheme produces a list of HSPs that are computed by a mixture of approximate and exact gapped alignment. If the approximate alignment is good enough, it is not recomputed. Other alignments may not be good enough either if computed approximately, but sometimes an approximate alignment gets recomputed optimally. When this happens, a mediocre alignment computed optimally will sometimes trump a better alignment that was comput-

ed approximately. This causes the mediocre alignment to erroneously make it to the traceback phase.

This means that we cannot mix together approximate and exact gapped alignments. All alignments for a given HSP list should be computed approximately, and if the score from one of those alignments is ambiguous then all of the previously computed alignments should be thrown away and recomputed optimally. This technique removes 90% of the output differences, but the total number of gapped alignments increases, and wipes out the runtime savings of using semi-gapped alignment at the default e-value cutoff! We can recover the performance with one more trick: semi-gapped alignment is not attempted at all if the highest-scoring *ungapped* alignment exceeds the gapped cutoff. Experiments show that when this happens, there is usually another ungapped alignment in the list whose score will be ambiguous if we try to extend it using the semi-gapped code, in turn causing all the previously computed approximate alignments to be thrown away. Basically this change uses semi-gapped alignment only when all the ungapped alignments are mediocre, and this avoids computing something like 80% of the approximate alignments that would be thrown away later.

These changes mean that the output differences are minor, and we still get a 5-10% speedup of blastp. That doesn't sound like much, given all of the complexity of the new alignment method, but after years of development there really isn't all that much room for performance improvement in blastp, and we'll take any improvement we can get.

Smith-Waterman Gapped Alignment

The functions in blast_sw.[ch] replace all of the alignment algorithms used by the engine with a custom implementation of the Smith-Waterman sequence alignment algorithm. All of the non-sequence-alignment portions of the engine (sequence handling, translation, statistics, etc.) still run normally, but when the engine is configured to perform Smith-Waterman alignment:

- No ungapped wordfinder runs at all

- The score-only gapped routine produces at most one HSP per query context. The HSP contains the relevant context, the entire range of the query and subject sequence, and the highest local gapped alignment score found by the score-only Smith-Waterman code. Every context of every query sequence is individually aligned to each subject sequence

- The Smith-Waterman traceback routine takes each HSP from the score-only phase and converts it to a list of *all* the local alignments between query and subject sequence whose score exceed the gapped score cutoff, along with the traceback for each alignment. This is the only example in the blast engine where one input alignment generates multiple different output alignments. The Smith-Waterman code has to work this way because the score-only routine examines every query-subject offset pair but only produces one alignment score. The traceback routine does much more

bookkeeping, using a clever variation on the basic algorithm to recover all the local alignments one at a time.

Nobody realistically expects the Smith-Waterman code to be used at the NCBI in a production environment. It was added to the blast source for several reasons:

- The alignment algorithm is rigorous, and does not depend on the heuristics that make blast run fast. It thus can provide 'gold standard' alignment results to which ordinary blast can be compared. Users can also take advantage of the rigorous results when the input problems are known to be very small, and alignment accuracy is vital

- Smith-Waterman is very easy to implement in dedicated hardware, and adding an implementation to the blast engine should make it easy for hardware companies to insert their own super-optimized Smith-Waterman hardware into an otherwise unmodified code base.

- the other features in the blast engine, i.e. sophisticated statistical models, translated searches, standardized output formatting, etc. are hard to find in other Smith-Waterman programs

Users should be aware that the vast majority of the time the ordinary blast heuristic is more than sufficient for your local alignment needs. The Smith-Waterman implementation runs 50-100 times slower than ordinary blast, and the alignments found are identical unless the alignment scores are very low. This is hardly surprising; blast has been optimized and tuned for almost 20 years now, and has always been intended as a complete replacement for Smith-Waterman. The code is present more for test purposes than anything else.

Configuring the Blast Engine

The configuration infrastructure for the blast engine is designed so that configuration information is constructed in two passes: blast_options.h describes a collection of <XXX>Options structures that are assumed to be user-specified, and can be changed at any time before the blast search begins. Once the search happens, these Options structures are converted into <XXX>Parameters structures that contain derived information used in the search. Parameters structures are not under user control, and once formed are used for the entirety of the search. When the engine needs a given parameter, most of the time it looks in a Parameters structure for the value to use, though sometimes the underlying Options structure is accessed. In any event, the engine never modifies the contents of any Options structure.

Options Structures

The Options structures and the way they are used are as follows (see blast_options.[ch]):

- LookupTableOptions contains information used to construct a blast lookup table. One of the fields here is a code for the lookup table type, and for blastn searches this may be overwrriden during a call to LookupTableWrapInit(lookup_wrap.c)

- SDustOptions / SSegOptions / SRepeatFilterOptions / SBlastFilterOptions: values needed for configuring low-complexity filtering, repeat detection, and masking of query sequences. Numerical values used in the Dust and Seg structures are pretty much always set to default values.

- QuerySetUpOptions: contains information used for constructing the BLAST_SequenceBlk used to represent the query sequence

- BlastInitialWordOptions: contains values used when finding ungapped alignments between a query and subject sequence. This includes the window size for the 2-hit wordfinder and the X-drop value used when computing ungapped extensions

- BlastExtensionOptions: contains values used for gapped extensions. This structure provides information needed to determine the gapped X-drop values, as well as describing the type of gapped algorithm to use. This last can be dynamic programming, greedy dynamic programming (blastn only), or Smith-Waterman. The algorithm used in the score-only and the traceback phases can be chosen independently, but Smith-Waterman can only be used for both phases.

- BlastHitSavingOptions contains all the information used to determine when an alignment would be rejected in the course of a search. This includes cutoffs for the e-value, the alignment score, the alignment length and percent identity, the number of alignments, the number of sequences containing hits, etc. Most of these limits are applied after the traceback phase, but limits that can be expressed as an alignment score are applied as early as possible. Note that nothing in this structure makes a distinction between ungapped and gapped alignments, the limits are applied identically to either kind.

- BlastScoringOptions contains all factors used to calculate the score of an alignment (protein or nucleotide). This includes the name of a score matrix, match and mismatch scores (blastn only), and gap penalties.

- BlastEffectiveLengthOptions contains all the information needed to calculate the search space used when computing expect values for alignments. When none of these fields are set, the engine will calculate all search spaces using the real values from the database being searched. Otherwise, the number of database sequences, the length of the database and the final search space can all be individually overridden. This structure is also flexible enough to hold a specified search space for each context of a collection of query sequences, and this capability is important when large query sequences are split.

- PSIBlastOptions contains information needed to compute position-specific score matrices.

- BlastDatabaseOptions currently only contains the genetic code needed to translate sequences in a (assumed nucleotide) database

The option structure for XXX has a XXXNew, XXXFree and FillXXX function associated with it. The first allocates memory for the structure and fills in default values, and second frees the allocated memory, and the third sets the fields in the structure to specified values. BLAST_InitDefaultOptions is a shorthand function that calls all the XXXNew functions, and BLAST_ValidateOptions is called by the C or C++ API layer to verify that option values are consistent with each other. The FillXXX functions are only called from C toolkit binaries; in addition to these functions, there is a C options API that slightly automates the setting of options. ncbi/algo/blast/api/blast_options_api.[ch] within the C toolkit implements the functions in this API. The handling of options in the C++ toolkit is much more complex; in C++, the CBlastOptions class contains an instance of each of the options structures, and there is a get- or set-value member for each field of all structures. The classes derived from CBlastOptions provide the intelligence needed to set option values.

Parameters Structures

Once the engine has started the search, the XXXOptions structures are converted to XXXParameters structures. Each such structure has an internal pointer to the Options structure on which it is based, as well as fields that the engine will use during the search. Initializing these structures is typically quite complex, with many interdependencies. The Parameters structures are as follows:

- BlastInitialWordsParameters is derived from BlastInitialWordOptions and contains information for finding ungapped alignments. The main field used in this structure is an array of BlastUngappedCutoffs structures, one for each context in the query. Each such structure contains the X-drop value and the cutoff score for ungapped alignments that occur on that context. BlastInitialWordsParameters also contains fields for the most conservative X-drop and cutoff score values, in case parts of the engine just need one value for these numbers. Finally, there is a field that specifies whether the wordfinder uses a Diag array or hashtable for triggering ungapped alignments, and a lookup table for speeding up nucleotide ungapped extensions.

- BlastLinkHSPParameters is not derived from a single Options structure, but contains the cutoff scores and overlap information used by the HSP linkin routines.

- BlastExtensionParameters is derived from BlastExtensionOptions and contains the X-drop values used for gapped alignments. Note that there is no per-context

data here, because gapped searches use the same statistical information for all contexts. If this ever changes, it would require modifying the next structure

- BlastHitSavingParameters is derived from BlastHitSavingOptions and contains the information needed to prune alignments. Most of the means of doing so will only depend on the fields of BlastHitSavingOptions, but there is also a per-context array of BlastGappedCutoffs structures, each of which contains the cutoff score for gapped alignments (score-only or traceback) occurring in one context.

- BlastScoringParameters is derived from BlastScoringOptions and contains many of the same fields. The difference is that the fields in BlastScoringParameters may be scaled up or down compared to their counter-parts in BlastScoringOptions, and when the engine needs gap penalties or some such it gets them from the BlastScoringParameters. Scaling of scoring information happens just once in the engine; although subsystems like composition-based statistics use their own scaling, they perform that scaling in their own internal structures

- BlastEffectiveLengthsParameters is derived from BlastEffectiveLengthsOptions. It currently contains the same information.

All of the fields in the Parameters structures are set by the engine, with no user intervention allowed. Filling the Parameters structures requires that a query and corresponding QueryInfo structure be available; hence, most of the initialization of these structures occurs just as the search is about to start. All Parameters structures are freed before the search completes; they are not visible outside the engine.

Overview of the Blast Engine Operation

This is intended to give a high-level guide for understanding the operation of the blast engine, how the engine is configured and the flow of control that performs a blast search. This last can be somewhat confusing, since there is a single codebase that performs all the different flavors of blast and parts of that codebase are turned on or off on a per-blast-flavor basis.

At the most abstract level, a blast search proceeds as follows:

```
prepare the blast options
prepare the query
prepare the database
build the lookup table
for each of N threads {
   while database has unsearched sequences {
      reserve a group of sequences from the database
      find alignments between the sequences and the query
      save alignments found
   }
}
```

```
for each database sequence containing alignments, perform traceback
display the alignments somehow
```

Within the C toolkit, applications must perform the first step, half of the second step, and the last step by themselves (though the blast API and the rest of the C toolkit each contain a lot of code to automate those steps). All the other steps are encapsulated in functions from algo/blast/api/blast_api.c, as follows:

- Blast_TwoSeqLocSetsAdvanced performs a blast search to align two sequences embodied as ASN.1 Seq_loc objects

- PHIBlastRunSearch performs regular expression searches between a query sequence and a database

- Blast_DatabaseSearch aligns one or more query sequences to the sequences in a database

These functions do a little initialization (usually limited to initializing a database) and then call Blast_RunSearch to perform all other steps of the search and return any alignments found. Details can be found in the C blast binaries, in ncbi/demo (blastall.c, megablast.c, rpsblast.c, blast_driver.c)

For the C++ toolkit, the query and database are initialized by the application, and the search itself is performed using either CLocalBlast (which uses the blast engine library directly) or CRemoteBlast (which sends the search to the blast queueing system at NCBI). Example applications that use the blast engine are available in the C++ toolkit (src/app/blast/*).

The next sections will cover each of these processes in turn.

Preparing the Search Options

This phase involves allocating and filling in the C Options structures described in options.txt; the method of doing so is toolkit-dependent. For the C++ toolkit, all of the C options structures are wrapped collected into the CBlastOptions class (primarily defined in include/algo/blast/api/blast_options.hpp). The CBlastOptionsHandle class (blast_options_handle.[ch]pp) derives from this, adding an interface for constructing different Handle objects with various combinations of options filled in. Finally, CBlast<program>OptionsHandle creates the same types of objects directly, if you know the kind of blast search you want. The C++ classes that run the blast engine can take either a CBlastOptions or a CBlastOptionsHandle as a delivery vehicle for the configuration of a blast search. Much more on configuring the blast engine options can be found in the blast chapter of the NCBI C++ Toolkit book (http://www.ncbi.nlm.nih.gov/books/bv.fcgi?rid=toolkit.chapter.ch_blast). For the C toolkit, SBlastOptions (algo/blast/api/blast_options_api.[ch]) is a structure that wraps the C structures used by the engine, and there are a few non-blast-program-specific functions for filling individual fields within the C structures. C toolkit applications also make use

of the FillXXXOptions structures defined in blast_options.c; the initializing of C structure fields happens in a C-toolkit-binary-specific way, see for example s_FillOptions in blastall.c

Preparing the Query

By the time the blast engine is called, the set of queries to be searched must all be initialized into a BLAST_SequenceBlk and its corresponding BlastQueryInfo structure. This process is again toolkit-dependent, since query sequences can come from a variety of different sources and be encoded in a variety of different formats (for example, as text in a file, or as a C or C++ Bioseq ASN.1 construct).

When there are many query sequences, the engine can be called multiple times, with each instance searching a subset of the complete collection of queries. This is much more efficient than searching each query individually, since the database is scanned only once per batch. Another advantage of batching is the extra efficiency possible when many queries are small; in this case, the internal data structures used by the blast engine have room to fit several queries at once and still remain in cache. This makes searching multiple query sequences almost free if those sequences are small. The optimal batch size is program-dependent, since different blast programs have different performance characteristics. For example, blastn can adaptively use very large lookup tables when given many query sequences, so its preferred batch size is larger than that of blastp.

In the C toolkit, BLAST_GetQuerySeqLoc (algo/blast/api/blast_input.c) is the function responsible for reading a batch of sequences from file and converting to a C toolkit ASN.1 Seq_loc structure. This function can also be configured to save masking locations corresponding to lower-case sequence letters in the text of the file. Once a batch of sequences has been loaded in this way, a call to Blast_FindRepeatFilterSeqLoc produces additional masking locations corresponding to (organism-specific) repetitive regions in the query sequence. This is optional, but if repeats are to be found the engine must perform an entire separate blast search to recover the repeat locations. Finally, the query Seq_loc and its list of mask locations are both passed to Blast_RunSearch, which uses BLAST_SetUpQuery to create a BlastQueryInfo and BLAST_SequenceBlk that the engine core will ultimately use (via s_QueryInfoSetUp and s_GetSequence, respectively).

In the C++ toolkit, the ‘blastinput’ library (algo/blast/blastinput) converts a collection of query sequences (expressed as a C++ stream) into a collection of SSeqLoc structures (i.e. a TSeqLocVector). These structures will also contain the masking locations to be used later. CBlastFastaInput is the class used to perform the conversion. C structures are populated by calling SetupQueryInfo_OMF and then SetupQueries_OMF (both are located in blast_setup_cxx.cpp). The rest of the C++ code accesses pointers to these structures via members of ILocalQueryData (query_data.[ch]pp). There is also a IRemoteQueryData, used to store the collection of Bioseqs (presumably retrieved from a remote server) that will eventually be used to create the C structures. The handling of ILocalQueryData and IRemoteQueryData is unified inside the IQueryFactory class (query_data.[ch]pp), and this is the object that the rest of the C++ blast API uses to access C structures corresponding to query sequences.

Most applications will initialize their query sequences by instantiating a CObjMgr_QueryFactory; this class uses the C++ Object Manager to retrieve sequence data from arbitrary sources. Lightweight applications can also use CObjMgrFree_QueryFactory, which assumes the sequences are directly available as CBioseq objects and hence do not need to rely on the Object Manager. Both kinds of QueryFactory derive from IQueryFactory.

C++ applications that use the Object Manager have the option of detecting repeat regions in the query sequences, using one of the functions in algo/blast/api/repeats_filter.hpp; repeats are detected at the point where the query sequences are converted to SequenceBlk structures.

By the time the core of blast runs, all mask locations must already have been computed for the collection of queries. Lower-case mask locations are computed when the query sequences are first read, repeat locations are computed before the engine runs, and low-complexity filtering is a handled specially. Nucleotide low-complexity filtering is applied to all sequences before calling the engine, using Blast_FindDustSeqLoc (C toolkit) or Blast_FindDustFilterLoc (C++ toolkit). These implementations of the DUST filtering algorithm are completely different from each other; the C++ version is slightly more effective and about twice as fast as the C version, because it was developed later (see the Symmetric DUST paper in the bibliography). We have not ported the improved DUST to the C toolkit because it's not performance-critical for the vast majority of searches. Protein low-complexity filtering using the SEG algorithm occurs in BLAST_MainSetup (algo/blast/core/blast_setup.c), using BlastSetUp_Filter (algo/blast/core/blast_seg.c). This asymmetry is needed because composition-based statistics require the SEG algorithm to be run on database sequences, and this must take place inside the engine so the SEG implementation is included there.

Before the core engine runs, masking locations on the query sequences are represented by lists of Seq_loc structures (C toolkit) or CSeq_loc objects (C++ toolkit). To remove the toolkit dependencies, all such locations are converted to C BlastMaskLoc structures (algo/blast/core/blast_def.h) using BlastMaskLocFromSeqLoc (C toolkit, algo/blast/api/blast_seq.c) or s_AddMask (C++ toolkit, algo/blast/api/blast_setup_cxx.cpp) before the core is called. Since SEG filtering is in the core, the resulting locations are in MaskLoc form already. None of these mask locations are actually applied to the query sequence data until BLAST_MainSetup explicitly does so by calling Blast_MaskTheResidues (algo/blast/core/blast_filter.c). It's possible that the query sequence data doesn't even need to be touched, if 'mask at hash' filtering is selected.

Preparing the Database

Toolkit independence also requires that the C engine core use an abstract representation of the blast database, from which sequences can be retrieved in BLAST_SequenceBlk form (in the appropriate sequence encoding). The core represents a source of biological sequences with a BlastSeqSrc (algo/blast/core/blast_seqsrc.[ch]). API code must fill in all the details of how the sequence source counts sequences, retrieves sequences, finds se-

quence sizes, etc. Applications that must iterate through multiple sequences can use a BlastSeqSrcIterator to do so, and the API will define how the iterator retrieves blocks of sequences to search and how it synchronizes with other threads to avoid retrieving the same sequence multiple times.

There are currently six implementations for the abstract BlastSeqSrc:

- a SeqSrc for readdb, the same blast database abstraction layer used by the old blast engine (algo/blast/api/seqsrc_readdb.[ch]).

- a SeqSrc for C applications to represent multiple read-in sequences (algo/blast/api/seqsrc_multiseq.[ch]). The core data object here is an array of SequenceBlk structures that are created by repeated calls to BLAST_SetUpSubject (algo/blast/api/blast_seq.c). Having this SeqSrc available means that the blast engine core can treat 2-sequence searches in exactly the same way as database searches. There are a few places where the core must detect that it is performing a 2-sequences search, and this SeqSrc will signal that by claiming that its 'database size' is zero. Nucleotide sequences are stored in blastna format and manually compressed into ncbi2na format when searched by the engine. This is the opposite way that a blast database would work.

- a SeqSrc for the C++ CSeqDB class (algo/blast/api/seqsrc_seqdb.cpp), the C++ based abstraction layer for blast databases.

- a SeqSrc for C++ applications to represent multiple read-in sequences (algo/blast/api/seqsrc_multiseq.cpp) described by a TSeqLocVector.

- a SeqSrc for C++ applications to represent multiple read-in sequences (algo/blast/api/seqsrc_query_factory.cpp) described by a IQueryFactory.

- a SeqSrc for indexed nucleotide databases (algo/blast/api/blast_dbindex.cpp)This is a work in progress.

In the C toolkit, the SeqSrc is initialized by functions that will call Blast_RunSearch, and passed to the core directly. In the C++ toolkit, the SeqSrc for a database is created by the CreateBlastSeqSrc method of CSetupFactory (algo/blast/api/setup_factory.[ch]pp). The input to this function can be either a previously created SeqDB object, or a CDataBaseSearch object. This latter (algo/blast/api/uniform_search.[ch]pp) is an abstraction of a blast database, and includes private data to remember the database name, GI lists, Entrez queries, etc. For 2-sequences searches the SeqSrc is initialized directly, while for database searches CSetupFactory does the initialization from the classes described next.

Running the Engine

With the initialization described above complete, control is transferred to routines that perform final initialization and then run the blast engine. The search is split between the preliminary stage that recovers alignment scores and boundaries, and the traceback stage

that reruns the alignment phase with traceback added. In the C toolkit, Blast_RunSearch performs both of these stages in order, while in the C++ toolkit each stage can be run separately via instantiating a CBlastPrelimSearch (prelim_stage.[ch]pp) or CBlastTracebackSearch (traceback_stage.[ch]pp) object. For convenience, preliminary and traceback stages are both performed by the Run method of CLocalBlast (local_blast.[ch]pp).

Regardless of which stage and which toolkit is involved, there is some final initialization that must happen. BLAST_MainSetUp performs this initialization (algo/blast/core/blast_setup.[ch]) as follows:

- performs protein low-complexity filtering for protein searches using the SEG algorithm
- for out-of-frame protein searches, computes the mixed-frame representation of the query

- given all the mask locations, possibly including all of the locations just computed, mask the query (if needed) and 'complement' the locations, producing another set of BlastMaskLoc structures that describe the regions of the query that are *not* masked.

- Initialize the score matrix and statistical information by calling BlastSetup_ScoreBlkInit (blast_setup.[ch]). All of the results are packed inside a BlastScoreBlk structure; statistical information is embodied in BlastKarlinBlk structures, one per query context, and there is a separate set of Karlin blocks for ungapped alignments, gapped alignments and psiblast alignments

- verify the the generated BlastScoreBlk has no null pointers

For the preliminary stage, a call to LookupTableWrapInit (lookup_wrap.[ch]) is also required to create the blast lookup table. This function is the only consumer of the complemented mask locations previously computed in BLAST_MainSetUp. With these calls complete, the C and C++ toolkit APIs both call the engine core to commence searching.

Blast Preliminary Stage

The preliminary stage uses a previously generated lookup table and SeqSrc to conduct the search. When run multithreaded, all threads reuse these quantities. There are two types of output from the preliminary stage:

- statistics (counts of lookup table hits, alignments, cutoff scores, etc.) from the search that are encapsulated within a BlastDiagnostics structure (algo/blast/core/blast_diagnostics.[ch]). Each thread updates a local BlastDiagnostics structure while each database sequence is searched, and when each thread is about to finish all of the per-thread structures are accumulated into a global BlastDiagnostics structure (which uses a mutex to synchronize access). For the C toolkit, the global structure is then accumulated into another BlastDiagnostics, so that

when the search is performed a batch of queries at a time, the statistics generated are for all query sequences combined. Since these statistics are just needed to reproduce the summary in traditional blast output, and the C++ toolkit binaries that run blast do not care about this summary, this last per-query-batch accumulation is not performed by the C++ API.

- the alignments themselves, encapsulated within a BlastHSPStream abstract data type (algo/blast/core/blast_hspstream.[ch]). We have implemented this abstraction of a collection of blast alignments because the preliminary stage should not have to know how traceback on those alignments will be computed. For example, if traceback is generated 'on-the-fly', as soon as preliminary alignments are computed, then hits to database sequences should be placed into a queue; however, if the preliminary stage is supposed to completely finish before traceback is computed, then we can use a more complex data structure that allows sorting of database sequences by their best hit. Having a BlastHSPStream hide the method of storage of alignments allows a simple interface to the results of the preliminary stage, where the traceback only needs a method to grab the next database sequence containing hits. There is a default implementation in the core that allows the latter method of traceback (algo/blast/core/hspstream_collector.[ch]), and an API-specific implementation of a queue of database sequences for on-the-fly traceback (algo/blast/api/hspstream_queue.[ch] in the C toolkit, algo/blast/api/hspstream_queue.[ch]pp in the C++ toolkit, the latter unused). The vast majority of blast searches will use the default HSPStreamCollector implementation.

Both toolkits perform the preliminary search by having each thread call Blast_RunPreliminarySearch (algo/blast/core/blast_engine.[ch]). This routine calls BLAST_GapAlignSetUp (algo/blast/core/blast_setup.[ch]) to allocate and fill the various Parameters structures derived from the input Options structures, then calls BLAST_PreliminarySearchEngine. In pseudocode, this routine does the following:

- call s_BlastSetUpAuxStructures to fill in a BlastCoreAuxStruct with pointers to the ungapped and gapped extension routines that the core will use (these depend on the blast program and the search options)

- initialize an iterator for the input SeqSrc. The iterator is responsible for reserving a group of sequences for this thread to search, one at a time. The size of the group must be decided at the time the iterator is initialized, and is (number of sequences in SeqSrc)/100 (or 1, if this is too small). We basically want a batch that is large enough to avoid too much synchronization between threads but small enough to make a load imbalance between threads unlikely. We really cannot predict ahead of time what an optimal batch size will be, since it is time-consuming to determine how many letters are in each sequence batch, and even if we could the runtime needed by blast depends only weakly on the sequence sizes. The current choice seems to work well enough. Later, when a batch has been searched, the next batch is reserved automatically

- Run the main loop:

```
while there are still sequences to search {

   retrieve the next subject sequence from the SeqSrc.

   For blastn searches the subject sequence will be in packed (ncbi2na)
   format if this is not a database search, then the search space to
   use for e-values depends on every query context and the particular
   subject sequence. Thus, recalculate the search space and the cutoffs
   for ungapped alignments (which depends on the subject length) for
   each query context by calling BLAST_OneSubjectUpdateParameters
   (blast_setup.[ch]). Database searches set the search space just
   once, in the API layer

   for tblastn, find the genetic code to use for the subject
   sequence. This can in principle be different for each subject
   sequence, but in practice all subject sequences use the same
   genetic code most of the time. The genetic code will ultimately
   be represented by a 64-entry lookup table, and the GenCodeSingleton
   structure (algo/blast/core/gencode_singleton.[ch]) caches all
   such previously encountered lookup tables

   call s_BlastSearchEngineCore (blast_engine.c) to collect the align-
   ments between all query contexts and the current subject in a local
   BlastHSPList structure (algo/blast/core/blast_hits.[ch]). Only the
   alignment boundaries and scores are saved

   for ungapped searches {
      for nucleotide subject sequences {
        call Blast_HSPListReevaluateWithAmbiguitiesUngapped to refetch
        the subject sequence in blastna (for blastn) or ncbi4na
        (tblast[nx]) format so that ambiguities are present, and then
        recalculate the scores of alignments accounting for ambiguities

        since adding ambiguities could have changed the alignment
        scores, recalculate the e-values of such alignments here, ei-
        ther using sum statistics (BLAST_LinkHsps, linkhsps.c) or
        ordinary blast statistics (Blast_HSPListGetEvalues,
        blast_hits.c)

        call Blast_HSPListReapByEvalue (blast_hits.[ch]) to prune
        alignments whose e-value is too high
      }

      compute the bit scores of alignments by calling
      Blast_HSPListGetBitScores (blast_hits.[ch]). This is
      ordinarily handled in the traceback phase (because traceback
      alignments can have different scores), but ungapped searches
      never generate traceback
   }

   save the BlastHSPList into the HSPStream (note that there is only
   one HSPStream per search, so this requires synchronization across
   multiple threads; in most cases, the save operation happens infre-
   quently and at unpredictable times, so contention should not be an
   issue)
```

```
}
```

s_BlastSearchEngineCore assumes a collection of query sequences and one subject sequence. The breakdown of operations is roughly as follows:

```
for tblast[nx], translate the subject sequence into
its six frames. For out-of-frame gapping, also compute
the mixed-frame representation of the sequence

for each subject context {

   form a SequenceBlk for the current subject context

   call s_BlastSearchEngineOneContext to align all query
   contexts with one subject context {

      for each subject sequence chunk of size MAX_DBSEQ_LEN(=5000000) {

           call the ungapped wordfinder, to produce ungapped alignments
           saved in a local BlastInitHitList structure; the alignments
           are sorted in order of decreasing score by the wordfinder via
           Blast_InitHitListSortByScore (blast_extend.c)

           for gapped searches {

              call the gapped wordfinder to convert the ungapped align-
              ments into gapped alignments, filling up a BlastHSPList

              prune alignments that share a common endpoint via
              Blast_HSPListPurgeHSPsWithCommonEndpoints
              (blast_hits.[ch]) and sort by score using
              Blast_HSPListSortByScore

           } else {
              convert the BlastInitHitList to a BlastHSPList
           }

           convert the subject offsets of any hits found so that offset
           zero is the beginning of the subject sequence, not the begin-
           ning of the current chunk, via a call to
           Blast_HSPListAdjustOffsets (blast_hits.[ch])

           call Blast_HSPListsMerge (blast_hits.[ch]) to possibly merge
           together pairs of HSPs. To be eligible for merging, one of
           the HSPs must have been found in a previous subject chunk and
           must end within DBSEQ_CHUNK_OVERLAP(=100) letters of the end
           of the chunk. The second HSP must be present in the current
           chunk and must begin in the first DBSEQ_CHUNK_OVERLAP letters
           of that chunk. The handling of chunks earlier assures that
           this overlap region is actually the same between the two
           chunks
        }
      }

      append any alignments found to the HSPList for the subject se-
      quence
```

```
    }
}

calculate the e-values of alignments, either using sum statistics (via
BLAST_LinkHsps, linkhsps.c) or ordinary blast statistics
(Blast_HSPListGetEvalues, blast_hits.c)

call Blast_HSPListReapByEvalue (blast_hits.[ch]) to prune
alignments whose e-value is too high
```

Note the asymmetry in the handling of query sequences and subject sequences in the pseudocode above; the core always assumes that all query contexts are handled at once, but that only one subject context is handled at a time. This turns out to be important because databases can contain very large sequences but typically the query sequence is quite small. Being able to handle large subject sequences a chunk at a time limits the size of data structures used in gapped alignment. The C++ API contains additional code that can split up a query sequence in a similar fashion, and for blastx this leads to large speedups.

Blast Traceback Phase

The traceback phase of blast is conceptually simpler than the preliminary phase; we assume that only one thread performs the traceback (though the code contains no global state and could in principle work in a multithreaded fashion), and the procedure for doing so is fairly straightforward.

The inputs to the traceback phase as the same as those of the preliminary phase, except that they also include the BlastDiagnostics structure and the HSPStream that were the output of the preliminary phase. The output from the traceback phase is an updated collection of Diagnostics, and a BlastHSPResults structure that contains the alignments with traceback included (ungapped searches generate the results in the same format but the alignments do not contain traceback). The C and C++ toolkits call Blast_RunTracebackSearch (blast_traceback.c) when performing traceback, which in turn calls BLAST_ComputeTraceback to perform the majority if the work involved.

BlastHSPResults is a sparse representation of the alignment output from a blast search. It contains a per-query array of BlastHitList (blast_hits.[ch]) structures, each of which contains a per-subject-sequence array of BlastHSPList structures. Each BlastHSPList contains a list of BlastHSP structures that describe individual alignments between one query and one subject sequence. Query sequences that do not contain any hits are assigned a NULL BlastHitList pointer. The use of BlastHSPResults is specific to the HSPStreamCollector flavor of BlastHSPStream; this means unsorted or on-the-fly blast traceback requires a different way of calling BLAST_ComputeTraceback that does not depend on the engine core, as well as a custom HSPStream implementation.

When the traceback has finished, each BlastHitList has its list of HSPLists sorted in order of worsening e-value of the best hit to the subject sequence. Note that the preliminary stage generates alignments that remember the query *context*, not the query *sequence*, containing the alignment. When the traceback asks the HSPStream for the pile of hits

corresponding to the next query-subject sequence pair, it does not care about the query context of each hit. Thus, when an HSPList is written to the HSPStream during the preliminary phase, the HSPStream maps the context of each hit back to the query sequence that contains that context. In the case of HSPStreamCollector-type HSPStreams, this could mean that the stream converts one input HSPList into several internal HSPLists, if one subject sequence contains hits to multiple query sequences. The functions that perform this remapping are Blast_HSPResultsSaveRPSHSPList (blast_hits.[ch]) for rpsblast and Blast_HSPResultsSaveHSPList (blast_hits.[ch]) for all other blast programs.

The procedure for BLAST_ComputeTraceback is as follows:

```
for rpsblast and rpstblastn {

   call s_RPSComputeTraceback

} else for composition-based statistics {

   call Blast_RedoAlignmentCore (blast_kappa.c); this function goes
   through each query-subject pair and recomputes gapped alignments af-
   ter sophisticated modifications to the score matrix

} else {
   while the HSPStream contains lists of hits {

      retrieve all of the HSPs that hit to the subject sequence that is
      next in the database. These can be segregated into multiple
      HSPLists, one per query sequence; all HSPLists are returned in a
      BlastHSPStreamResultBatch (blast_hits.[ch]) structure

      for a gapped search {
         retrieve the subject sequence

         for tblast[nx], find the genetic code for the subject sequence

         for a non-database search, recalculate the search space for
         each query sequence to account for the length of this particu-
         lar subject sequence
      }

      for each HSPList in the batch {
         if (gapped search) {
            call s_PHITracebackFromHSPList (for phiblast) or
            Blast_TracebackFromHSPList (for all other blast programs).
            These functions recompute every gapped alignment using a
            more conservative X-dropoff value, perform pruning on the
            results, and calculate e-values and bit scores.
         } else {
            compute bit scores for the HSPList
         }

         save the list of hits in the BlastHSPResults
      }
   }
```

More notes on Blast_TracebackFromHSPList:

We fetch database sequences only once and reuse the same sequence as often as possible. The alternative is to fetch the database sequence for every individual HSP list; while this would make the traceback code a lot more straightforward, the latency of fetching and converting large database sequences makes blastn searches against genomic databases run much slower. The database sequences are fetched in order of increasing ordinal number in the database; if there are hits to many sequences, this reduces the amount of thrashing as database volumes are opened and closed.

For subject sequences that must be translated, we do not form all 6 reading frames unless the subject sequence is small. For larger subject sequences the traceback uses 'partial translation', computed by calling Blast_HSPGetPartialSubjectTranslation (blast_hits.[ch]). This routine translates a region of the subject sequence that starts MAX_FULL_TRANSLATION nucleotides behind and ends MAX_FULL_TRANSLATION (=2100) beyond the subject range of each HSP, and limits the translation to just the frame in which the HSP occurs. The limit for giving up and translating all six frames is a nucleotide sequences of length below MAX_FULL_TRANSLATION, and I think this is a bit small. Partial translation is needed to avoid translating very large subject sequences if there are only a few small hits to those large sequences; while the translation process is fairly quick, converting millions of bases into six reading frames can easily exceed the runtime needed to perform a small gapped alignment. The maximum extent of the gapped alignment is limited to the region of the subject sequence for which a translation is available.

In addition to this mechanism, the C++ API allows fetching only a portion of a subject sequence during the traceback phase, and adding sentinel letters with value FENCE_SENTRY before and after properly read letters of the subject sequence. All of the gapped alignment routines, and the routines that perform translation of nucleotide sequences, look for the occurrence of a FENCE_SENTRY letter and invalidate any computed alignments if one is encountered. We try to compute traceback alignments with sentries in place, and if even one alignment encounters a sentry the entire BlastHSPList is thrown away and recomputed with the full subject sequence. This allows the amount of sequence data fetched around alignments to be kept fairly small, while still guaranteeing that correct answers are computed. The partial sequence fetching mechanism is implemented in the CSeqDB class and dramatically speeds up traceback generation involving large sequences like chromosomes.

The updating of scores from traceback alignments comes at the end of Blast_TracebackFromHSPList, with a call to s_HSPListPostTracebackUpdate. This includes calculation of e-values, bit scores and sum statistics. The pruning of gapped alignments that have insufficient percent identity happens in Blast_HSPTestIdentityAndLength, though ungapped alignments are not affected by this.

Whether in the preliminary or the traceback stage, BlastHitList (BlastHSPList) structures allow limiting the number of HSPLists (HSPs) that are saved. If the number of items that need saving is lower than the limit, then the list is left unsorted and new entries are appended to the list. When the number ot save equals the limit, the list is heapified and

turned into a priority queue. BlastHitList structures key on the best e-value among all the HSPs in each BlastHSPList, while BlastHSPLists key on the e-value of each HSP added. By default, in the preliminary stage there is no limit to the number of hits a BlastHSPList can store, except for ungapped searches where the limit defaults to kUngappedHSPNumMax(=400). The number of HSPLists that can be saved for a particular query defaults to somewhat larger than the number required in the traceback, to allow for alignments that drastically change score between the preliminary and traceback phases. Composition-based statistics also prune more alignments, so when composition-based corrections are selected the number of HSPLists in the preliminary phase is doubled. Finally, there is a minimum number of HSPLists to save, in order to deal with cases where users run blast with a very low number of hits to save, expecting to get only the best hit from the list that a full blast search would have produced. It usually takes a significant number of sequences in the preliminary stage to find the one that will produce the best alignment in the traceback stage. The logic implementing these choices is in SBlastHitsParametersNew (blast_hits.[ch])

A Blast Bibliography

Introductory papers (plus a good book):

Ian Korf, Mark Yandell, Joseph Bedell. "BLAST". O'Reilly, 2003

S. F. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. "Basic local alignment search tool. Journal of Molecular Biology, 215(3):403--410, 1990

Altschul SF, Madden TL, Schaffer AA, Zhang J, Zhang Z, Miller W, Lipman DJ. "Gapped BLAST and PSI-BLAST: a new generation of protein database search programs". Nucleic Acids Res. 1997 Sep 1;25(17):3389-402

Tatusova TA, Madden TL. "BLAST 2 Sequences, a new tool for comparing protein and nucleotide sequences". FEMS Microbiol Lett. 1999 May 15;174(2):247-50.

Alexander Pertsemlidis and John W Fondon III "Having a BLAST with bioinformatics (and avoiding BLASTphemy)"
http://www.lacim.uqam.ca/~anne/BIF7000/Pertsemlidis.pdf

Optimized lookup tables:

Michael Cameron, Hugh E. Williams, Adam Cannane, "A Deterministic Finite Automaton for Faster Hit Detection in BLAST"

Michael Cameron, Hugh E. Williams, "Comparing Compressed Sequences for Faster Nucleotide BLAST Searches"

Gapped alignment, background references:

Gotoh, O. (1982) “An improved algorithm for matching biological sequences”. J. Mol. Biol. 162, 705-708.

Needleman, S. B. and C. D. Wunsch (1970) “A general method applicable to the search for similarities in the amino acid sequences of two proteins”. J. Mol. Biol. 48, 443-453.

Smith, T. F. and M. S. Waterman (1981) “Identification of common molecular sequences”. J. Mol. Biol. 197, 723-728.

Geoffrey J. Barton “An Efficient Algorithm to Locate All Locally Optimal Alignments Between Two Sequences Allowing for Gaps”. Computer Applications in the Biosciences, (1993),9,729-734

Gapped alignment, more blast-specific:

Zheng Zhang, Scott Schwartz, Lukas Wagner, and Webb Miller “A Greedy Algorithm for Aligning DNA Sequences” Journal of Computational Biology, Volume 7, Numbers ½, 2000 Pp. 203-214

Zheng Zhang, William R Pearson, Webb Miller. “Aligning a DNA Sequence With a Protein Sequence”. Journal of Computational Biology, Volume 4 Number 3 (1997) pp 339-349

Michael Cameron, Hugh E. Williams, Adam Cannane, “Improved Gapped Alignment in BLAST”. IEEE/ACM Transactions on Computational Biology and Bioinformatics, vol 1 #3 (2004) pp 116-129

Phi blast:

Zheng Zhang, Alejandro A. Schaffer, Webb Miller, Thomas L. Madden, David J. Lipman, Eugene V. Koonin and Stephen F. Altschul, “Protein sequence similarity searches using patterns as seeds” Nucleic Acids Research, 1998, Vol. 26, No. 17 pp 3986-3990

Baeza-Yates & Gonnet, “A New Approach to Text Searching”, Comm. ACM 35(1992) pp 74-82

Wu & Manber, “Fast Text Searching Allowing Errors”, Comm. ACM 35(1992) pp 83-91

Sequence filtering:

Wootton, J.C. & Federhen, S. (1996) “Analysis of compositionally biased regions in sequence databases.” Meth. Enzymol. 266:554-571

Wootton, J.C. & Federhen, S. (1993) “Statistics of local complexity in amino acid sequences and sequence databases.” Comput. Chem. 17:149-163

Morgulis A, Gertz EM, Schaffer AA, Agarwala R. “A Fast and Symmetric DUST Implementation to Mask Low-Complexity DNA Sequences” J. Comput. Biol. 2006 Jun; 13(5):1028-40

Alignment scoring systems:

Dayhoff, M.O., Schwartz, R.M. & Orcutt, B.C. (1978) “A model of evolutionary change in proteins.” In “Atlas of Protein Sequence and Structure, vol. 5, suppl. 3.” M.O. Dayhoff (ed.), pp. 345-352, Natl. Biomed. Res. Found., Washington, DC.

Schwartz, R.M. & Dayhoff, M.O. (1978) “Matrices for detecting distant relationships.” In “Atlas of Protein Sequence and Structure, vol. 5, suppl. 3.” M.O. Dayhoff (ed.), pp. 353-358, Natl. Biomed. Res. Found., Washington, DC.

Altschul, S.F. (1991) “Amino acid substitution matrices from an information theoretic perspective.” J. Mol. Biol. 219:555-565.

States, D.J., Gish, W., Altschul, S.F. (1991) “Improved sensitivity of nucleic acid database searches using application-specific scoring matrices.” Methods 3:66-70.

Henikoff, J.G. (1992) “Amino acid substitution matrices from protein blocks.” Proc. Natl. Acad. Sci. USA 89:10915-10919.

Altschul, S.F. (1993) “A protein alignment scoring system sensitive at all evolutionary distances.” J. Mol. Evol. 36:290-300.

E Michael Gertz, “BLAST Scoring Parameters” 3/16/2005
ftp://ftp.ncbi.nlm.nih.gov/blast/documents/developer/scoring.pdf

Alignment statistics:

Karlin, S. & Altschul, S.F. (1990) “Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes.” Proc. Natl. Acad. Sci. USA 87:2264-2268.

Karlin, S. & Altschul, S.F. (1993) “Applications and statistics for multiple high-scoring segments in molecular sequences.” Proc. Natl. Acad. Sci. USA 90:5873-5877.

Dembo, A., Karlin, S. & Zeitouni, O. (1994) “Limit distribution of maximal non-aligned two-sequence segmental score.” Ann. Prob. 22:2022-2039.

Altschul, S.F. (1997) “Evaluating the statistical significance of multiple distinct local alignments.” In “Theoretical and Computational Methods in Genome Research.” (S. Suhai, ed.), pp. 1-14, Plenum, New York.

Seteven A. Benner, Mark A. Choen, “Empirical and Structural Models for Insertions and Deleteione in the Divergent Evolution of Proteins” J. Mol. Biol. (1993) 229: 1065-1082

PSI blast:

Schaffer AA, Aravind L, Madden TL, Shavirin S, Spouge JL, Wolf YI, Koonin EV, Altschul SF. (2001) “Improving the accuracy of PSI-BLAST protein database searches with composition-based statistics and other refinements.” Nucleic Acids Res. 2001 Jul 15;29(14):2994-3005

Henikoff S, Henikoff JG. “Position-based sequence weights”. J Mol Biol. 1994 Nov 4;243(4):574-8.

Composition-based Statistics:

Altschul SF, Wooton JC, Gertz EM, Agarwala R, Morgulis A, Schaffer AA, Yu YK. “Protein Database Searches using Compositionally Adjusted Substitution Matrices”. FEBS Jour. 2005 Oct; 272(20):5101-9

Yu YK, Gertz EM, Agarwala R, Schaffer AA, Alschul SF. “Retrieval Accuracy, Statistical Significance and Compositional Similarity in Protein Sequence Database Searches”. Nucleic Acids Res. 2006; 34(20):5966-73

Gertz EM, Yu YK, Agarwala R, Schaffer AA, Altschul SF. “Composition-Based Statistics and Translated Nucleotide Searches: Improving the TBLASTN module of BLAST”. BMC Biol. 2006 Dec 7;4:41

Discontiguous seeds for BLAST:

Ma, B., Tromp, J., Li, M., “PatternHunter: faster and more sensitive homology search”, Bioinformatics 2002 Mar;18(3):440-5

Ming Li, Bin Ma, “PatternHunter II: Highly Sensitive and Fast Homology Search” Genome Informatics 14: 164-175(2003)

Jeremy Buhler, Uri Keich, Yanni Sun, “Designing Seeds for Similarity Search in Genomic DNA” RECOMB 2003

Yanni Sun, Jeremy Buhler, “Desinging Multiple Simultaneous Seeds for DNA Similarity Search” RECOMB 2004

Miscellaneous topics:

Piotr Berman, Zheng Zhang, Yuri I. Wolf, Eugene V. Koonin, and Webb Miller “Winnowing Sequences from a Database Search”. Journal of Computational Biology

Volume 7, Numbers 1 / 2, 2000 pp. 293-302

Valer Gotea, Vamsi Veeramachaneni, and Wojciech Makaowski “Mastering seeds for genomic size nucleotide BLAST searches” Nucleic Acids Research, 2003, Vol 31, No. 23, pp 6935-6941

Appendix A: Optimization of MEGABLAST

NOTE: The work described in this appendix was performed around 2004 and parts of the results have been superseded by further development. It is included for historical interest, and to give a flavor of the kind of low-level details that developers optimizing megablast should bear in mind.

Introduction

This document is the result of a performance analysis of MEGABLAST. Recent measurements have shown that this subprogram of the BLAST family does not scale well with more recent high-performance pro-cessors. The sections that follow will show that correcting this situation is not a simple matter, but a few small changes to the existing NCBI toolkit will allow better MEGABLAST performance on the machines in common use today. The conclusion will offer additional suggestions for more involved changes that can increase the performance improvements.

These performance optimizations are important only in the case when megablast spends most of its time accessing its lookup tables. While this is definitely the case when performing cross-species searches, if the sequences involved contain e.g. human repeats then the vast majority of runtime will be spent elsewhere, and the optimizations described below become irrelevant.

A Case Study

To illustrate MEGABLAST’s behavior, we will begin with an example problem to submit to the NCBI toolkit. The query is a sequence composed of the first 300kbp from the HTGS database, and the database to search will be patnt (which is 180MB in size at the time of this writing, containing about 720Mbp of sequence data).

MEGABLAST will first break the query up into contiguous 12-base chunks; for the sample query, there are about 600,000 of these “seed hits”. Each seed hit forms a 24-bit index into a hashtable with 2^{24} entries. The hashtable at this index stores a pointer to a linked list of offsets into the query which contain this 24-bit index.

Because a given seed hit can be placed in one of 16.7 million locations, it’s expected that the hashtable will be mostly empty unless the query sequence is very large. For the ex-

ample, 530,000 hashtable entries contain a linked list of offsets, and the rest are empty. Hence the hashtable in this case is 3% full.

70-80% of the total runtime for MEGABLAST goes into scanning the subject database. For each sequence in the database, 12-base words are pulled out with some stride, and each word becomes a 24-bit index into the hashtable. A linked list at that position in the hashtable indicates an exact 12-base match with one or more words from the query; the offsets stored at this hashtable position are collected and submitted for hit extension elsewhere in the NCBI toolkit.

The hashtable is 64 megabytes in size, and checking it for linked lists amounts to accessing the hashtable randomly, since the access pattern depends on the database and so can be considered approximately random. A table this large will not fit in cache on almost all modern processors, and so to reduce the average latency of accessing the hashtable an auxiliary bitfield is first checked to see if the corresponding entry in the hashtable is empty. Because this presence vector (PV) array needs only one bit where the hash table needs a full 32-bit word, the PV array is 2 megabytes in size.

Unfortunately, using a PV array this size does not improve the performance at all. The reason is that most modern processors (Pentium 3, Pentium 4, Apple G4) have caches much smaller than 2 megabytes (most are .5 megabytes or even smaller). Thus, getting the PV array to fit into cache requires "compressing" the array, so that one bit in the PV array is used for multiple hashtable entries. The advantage is that with enough compression the PV array can be made to fit in any size cache, but the drawback is that as the compression increases there are more and more spurious accesses to the hashtable, each of which cause a cache miss. Put another way, the smaller the PV array gets, the less effective it is at stopping cache misses.

To use concrete numbers, a model for the average latency for each word in our example subject sequence is:

$$\text{Latency} = (\text{L2 latency}) + C * P * (\text{Memory latency} + \text{TLB latency})$$

where "L2 latency" is the latency to the processor's external cache, C is the factor by which the PV array is compressed, P is the percentage of hashtable entries that are nonempty (3% in the case of our example computation) and "TLB Latency" is the time needed to reload a translation lookaside buffer entry into the processor. For a 2.4GHz Pentium 4 system, the currenttoolkit, and the example problem,

L2 latency = 14 clock cycles
C = 32
P = .03
Memory latency = 350 clock cycles
TLB latency = 20 clock cycles

This assumes simplistically that compressing the PV array by 32 would cause 32 times the number of cache misses. Counting the number of spurious hashtable accesses in our real example query shows that C=32 causes only 10 times as many spurious hashtable

accesses and 20 times the cache misses. The extra factor of two is probably because a hashtable access kicks a small part of the PV array out of cache, so that a second cache miss is incurred to access the PV array later. All told, real MEGABLAST on real Pentium 4 hardware in this case has a per-subject-word latency of 240 clocks.

Analysis

These numbers mean that on average every word of the database incurs 0.75 cache misses. MEGABLAST with these parameters takes 9.7 seconds to run. While this is not an overly long runtime, both the query and the database in this example are much smaller than is typical for MEGABLAST runs. Further, with the minimal number of cache misses (i.e. using a PV array that fits in cache with no compression) the expected runtime for our example should be in the neighborhood of 3 seconds. MEGABLAST's problems only get worse with increasing query length, since in that case more of the hashtable (and hence the PV array) is filled, and main memory accesses occur more often.

While there is potential for MEGABLAST to run much faster, any alternate method for collecting seed hits from the main hashtable must incur less than 0.75 cache misses per word on average; otherwise it would be slower than the present toolkit. Thus, collecting all the seed hits into an array and performing binary search will never be an improvement unless the query is quite small. For our example, storing the seed hits contiguously will require about 4 megabytes, and bsearch() into this array would incur multiple cache misses per word. Likewise, loading and then storing the subject sequence would probably be acceptable, but the present code does not produce much write traffic, and writes would add more uncertainty to the cache behavior of MEGABLAST.

The easiest way to improve performance without moving any data would involve better utilizing the PV array. Reducing the amount of compression in the PV array would greatly cut the number of spurious cache misses, but reducing it too much would cause cache misses accessing the PV array.

Here are the MEGABLAST runtimes (in seconds) for various sequence sizes, with the PV array compression factor varied. A compression factor of C means that one bit of the PV array counts for C hashtable entries.

Query size	Runtime (in seconds) by compression factor (current default is C = 32)				
	C = 2	C = 4	C = 8	C = 16	C = 32
50k	5.52	4.42	3.71	3.87	4.93
80k	5.91	4.97	4.50	4.71	5.63
100k	6.27	5.51	4.93	5.22	6.24
120k	6.68	5.88	5.54	5.84	6.73

140k	6.84	6.13	5.80	6.22	7.12
160k	7.17	6.70	6.42	6.57	7.61
180k	7.47	6.80	6.78	7.25	8.00
200k	7.68	7.30	7.38	7.56	8.21
240k	8.35	7.78	7.86	8.21	9.14
280k	8.79	8.51	8.34	8.96	9.23
300k	9.09	8.80	8.78	9.29	9.66

Note that these and all other timings presented were achieved on a Dell 2650 server with dual 2.4GHz Pentium 4 processors, only one of which was used.

The optimal choice of compression factor (C = 8) yields a speedup of 5-20% over the default. A compression factor of 8 here means that the PV array is 256 kbytes in size, or half the size of the external cache on the Pentium 4 test machine. In general, the larger the external cache on the target machine, the less the PV array should be compressed, and the more effective it will then be in reducing the number of cache misses.

This brings up another issue: the entire NCBI toolkit is designed to be as machine-independent as possible, and at present there is no way to pass to the toolkit fundamental machine parameters like cache sizes. It would be ideal for the application to determine these parameters on its own, either by accessing operating system services that can do this or by direct experi-mentation. Fixing a single value at compile time is the next best thing, but the above shows that a single fixed value carries a performance penalty on hardware for which the fixed value is suboptimal. Yet another approach would involve maintaining a database of hardware-specific parameters, which a Configure script could use at toolkit build time.

As another data point, repeating the above test on Sparc hardware (with a 4 megabyte external cache) shows an optimal compression of 4 instead of 8. The time difference is quite small between the two, although it's 20-25% better than the default of 32.

A similar optimization is possible with one variety of discontiguous MEGA-BLAST. Discontiguous MEGABLAST with wordsize 11 uses a hashtable that is 16 megabytes instead of 64. In this case, the PV array can be compressed less (one fourth as much) and still fit in cache. For moderate size queries, accounting for the smaller hashtable size allows for speedups in excess of 30%.

Finally, experiments show that when the query is very small (under 10kbp), more PV array compression is better. This is probably because the hashtable is very sparsely filled in this case, and a large amount of compression will leave most of cache free (since the PV array is small) and still allow the PV array to stop many cache misses.

Results

For sequence sizes out to about 400 kbp the optimal PV array compression is a win performance-wise. For larger sequences it seems to be a liability and the compression should be increased.

Conclusions and Future Work

Significant speedups are possible in the MEGABLAST program if the size of the cache(s) on the local machine are accounted for. These optimizations, however, have an upper limit on the query size that can benefit; for queries larger than this, more radical surgery can theoretically improve performance.

For queries that are very large, the main hashtable is mostly full and must be repeatedly accessed; the PV array cannot prevent this, even if not com-pressed at all. The only way to gain more reuse out of the hashtable is to change the order in which it is accessed.

A promising optimization is to take a large subject sequence and proceed through it in two passes. The first takes each word in the subject and drops it into one of a large number of hash buckets (512 or 1024), where the hash bucket index is determined from the top few bits of the 24-bit subject word. The second pass goes through each bucket one at a time and uses the words in that bucket as indices into the main hashtable. The first pass corresponds to one pass of a large-radix sort of the subject sequence, and the second pass collects the seed hits from the main hashtable. Preliminary experiments suggest that for queries and subjects of size above 1 Mbp this two-pass method is up to 30% faster than the present 1-pass algorithm.

However, the code that calls the subject scanner expects seed hits that are returned to be in numerical order, and the two-pass scheme does not do this. Attempting to sort the seed hits after they are collected negates any time saved using the two-pass method. It remains to be seen if the hit extension code can be modified to make this optimization possible.

Another modification that the two-pass scheme would need, but which would be useful on its own, involves batching subject sequences together. This is already performed on the query input to MEGABLAST, but subject sequences are still processed one at a time. In the case of databases like EST, where many subject sequences are small, this could lead to worse performance. The two-pass scheme requires that sequences be aggregated together in order to have many subject sequence words fill up each hash bucket, providing the reuse of the hashtable that is needed for a cache to work. Even without a two-pass scheme, aggregating subject sequences could make MEGABLAST performance less database-dependent, since even in the present code it would allow the PV array to stay in cache longer. Preliminary measurements show that large numbers of small sequences do not degrade performance greatly, so the aggregation may not be necessary. However, more work is needed to ensure this.

Appendix B: Blast Interval Tree

There are many occaisions in the gapped alignment phase of blast - both the score-only and the score-with-traceback flavors - where it is important to know if one alignment is completely contained within another. The old engine made these ‘containment decisions’ by brute force, comparing each of a set of alignments to the entire set of alignments that could envelop it. The quadratic complexity of this approach becomes a serious bottleneck when handling very large search problems.

The new engine makes containment decisions using a data structure called an interval tree. The interval tree implementation in the new engine (blast_itree.[ch]) handles only a small subset of what a real interval tree can do, and is specialized for the needs of blast.

For protein searches, ungapped HSPs are discarded if the query range and subject range are both enveloped by the query and subject ranges of a previously found gapped alignment. For contiguous megablast, the containment tests are a little more subtle: ungapped alignments may start inside the ‘box’ formed by a higher-scoring previous gapped alignment, but the start point of the ungapped alignment must lie more than 6 diagonals away from the start point of the gapped alignment.

There are two reasons for this difference; first, gapped alignments found by megablast can be very large, and the greedy nature of most megablast gapped alignments means that smaller gapped alignments can often cluster around interesting regions of the main, ‘big’ alignment. These smaller alignments are sometimes of independent interest and should not be automatically pruned. The second reason is that since most gapped alignments have few actual gaps, older versions of this code implemented containment tests by sorting the list of gapped HSPs in order of increasing diagonal and limiting the search to HSPs up to 6 diagonals away. This was much faster than the alternative, testing every ungapped HSP against all previously computed gapped alignments. Using an interval tree for this application allows the two kinds of containment tests to be unified, and for very large lists of alignments is much faster still. Note that there is nothing special about fixing the number of diagonals at 6; the exact value is controlled by BlastHitSavingOptions::min_diag_separation.

Another application of the interval tree is culling. This feature, which was added to the old engine around 2000, attempts to highlight ‘interesting’ alignments found in a blast search. The culling code considers an alignment interesting if it includes a region of the query sequence that not too many other alignments contain. The containment decision considers all hits to all database sequences simultaneously, and is an attempt to elevate the priority of alignments that may be low-scoring but are also somewhat unique. More formally, the culling phase removes any alignment whose query range is enveloped by the query range of more than X higher-scoring alignments (given by BlastHitSavingOptions::culling_limit). The original culling implementation was very complex; it used a variant of an interval tree and attempted to use culling in both the score-only and traceback phase of the search. In the new engine, culling is only applied to the set of alignments remaining after the traceback phase, and the implementation (in

Blast_HSPResultsPerformCulling, located in blast_hits.c) is very simple, reusing the interval tree implementation with some slight extensions.

What is an Interval Tree?

Interval trees are fundamental data structures in many fields; they are used in geographic information systems, computational geometry, and sometimes in database management systems. The description will begin with the 1-dimensional case.

In one dimension, an interval tree is a method for indexing a collection of (start,stop) pairs. Each pair represents the bounds of a line segment. Once the tree is built, it can be used to return the collection of line segments that overlap a 'query' line segment. This is how it is typically used. The requirements of blast are a tiny subset of this: return true if at least one segment in the tree envelops the query segment.

Each internal node of the interval tree has three pointers: a left subtree, a right subtree and a midpoint list. Each internal node also has a 'center', a location used for indexing. Line segments that lie completely to the left of node.center go in the left subtree; segments strictly to the right of node.center go in the right subtree; and segments that straddle node.center go in the midpoint list. If an interval tree node only has to index a single line segment, the node points to the segment directly and becomes a leaf of the tree. Thus, except for the midpoint list, an interval tree is a pretty standard binary tree. A fully indexed interval tree will have every line segment either in a midpoint list or in a tree leaf.

To solve the general 'query intersection' problem, each endpoint of the query segment is treated separately. The left and right endpoints are each used for a 'stabbing query'. For each endpoint, the leaf of the tree that envelops the endpoint is calculated. This locates the tree leaves that contain the start- and endpoints of the query. The complete set of line segments that intersect the query segment can just be read from the tree nodes between the left leaf and the right leaf. Just walk through all the leaves and internal nodes in the tree between the leaf containing the query start and the leaf containing the query end, and read off each line segment that those nodes contain. If the tree is balanced and contains N segments, the worst-case time for a stabbing query is O(log(N)), and this is the complexity of the segment retrieval problem. Tree construction takes O(N*log(N)) time.

Blast alignments have a query range and a subject range, so an interval tree suitable for blast use should index a set of rectangles rather than a set of line segments. More generally, an interval tree of dimension d indexes a set of d-dimensional 'hyper-rectangles'. Within the d-dimensional tree, every internal node has a (d-1)-dimensional interval tree instead of a midpoint list. Tree construction has a worst-case time of O(N*d*log(N)) and a stabbing query takes time O(d*log(N)). As d increases, worst-case behavior becomes less and less likely since the set of hyper-rectangles must be packed tighter and tighter to cause a traversal of all d dimensions of the interval tree.

A Blast Interval Tree

As previously stated, the containment decision problem that blast needs solved is easier than the general segment retrieval problem. For one thing, no segments need to be retrieved! Only a boolean is required, set to true if one of the alignments (rectangles) indexed by the tree completely envelops the query alignment. We can think of the tree taking a callback that implements the containment test, and then traversing the tree down to one of the leaves, applying the containment test to alignments encountered along the way. No stabbing queries are needed, since we just use the tree as a filter to reduce the number of times the callback is needed. Traversal stops when the tree runs out or when an enveloping alignment is found.

Let the first dimension of the interval tree be the query range and the second dimension be the subject range of an alignment. Then the main function that solves the containment decision problem looks conceptually like:

```
Boolean BlastIntervalTreeContainsHSP(node, HSP) {

while (node is not a leaf) {

   /* for query range, traverse the interval
      tree of subject ranges. For the subject range,
      traverse the list of HSPs */

   if (BlastIntervalTreeContainsHSP(node->midpoint_list, HSP) == true)
      return true
   if (HSP->end < node->center)
      node = node->left;
   else if (HSP->start > node->center)
      node = node->right;
   else
      return false;   /* midpoint list was already tested */
   }

   /* compare to HSP at leaf */
   return containment_test(node->HSP, HSP);
}
```

The nature of the tree also deserves some explanation, since the nature of the data being indexed lends itself to a simplified tree implementation. The interval tree is a structure of type BlastIntervalTree, and an internal node of the tree is of type SIntervalNode. The implementation includes a batch memory allocator, and pointers to other tree nodes are actually integer offsets into the array of allocated nodes. This lets us avoid using malloc() to allocate new nodes, and also does not invalidate pointers when the entire array of allocated nodes is reallocated.

Unlike the general case, the endpoints of ranges being indexed are always integers (i.e. offsets into the query and subject sequences), and the maximum values these ranges can take is known at tree construction time. This suggests that rather than continually rebalancing the tree to keep the number of HSPs in left and right subtrees approximately the same, we can instead hierarchically divide the space that the HSPs occupy. For example, to add an HSP to a tree that indexes only the query offsets, BlastIntervalTreeAddHSP will do something like the following:

```
while(1) {
   if (HSP->end < node->center) {
      if (node has no left subtree) {
         node->left = leaf with HSP in it
         return;
      }
      if (left subtree is not a leaf) {
         node = node->left;
         continue;            /* descend the tree */
      }
   } else if (HSP->start > node->center) {
      if (node has no right subtree) {
         node->right = leaf with HSP in it
         return;
      }
      if (right subtree is not a leaf) {
         node = node->right;
         continue;            /* descend the tree */
      }
   } else {
      add HSP to node->midpoint_list
      return;
   }

   /* reached a leaf of the tree. We cannot put HSP here;
      instead, we have to try inserting new internal nodes
      until the leaf is in a subtree by itself (or gets added
      to a midpoint list somewhere) */

   create new internal node, link to previous
   node in place of the leaf that used to be there

   add old leaf to new internal node

   node = new internal node
}
```

The root of the tree has a center at (min_query_offset + max_query_offset)/2. When a new internal node is created, its center is the midpoint of the left or right query range of the parent.

This technique is very common in fields like computational astrophysics, where for example a collection of masses orbit in a fixed volume of space. The tree never needs rebalancing, the heuristics blast uses mean that HSPs will never be densely stacked together, and sequence offsets being integers mean that the tree will have depth at most $\log_2$(max_query_size)

There are some subtle implementation issues to consider.

- because the interval tree indexes HSPs and not just line segments, sometimes the midpoint list of an internal node will just be a linked list of HSPs and sometimes it will be the root of an entire interval tree. An internal node only has one set of pointers for subtrees and the midpoint list; the use to which these pointers are put

depends on context. While this is potentially confusing, it saves memory and lets a single loop build the entire tree

- It is surprisingly tricky to determine the maximum sequence offsets that an interval tree will encounter. The maximum subject offset is usually straightforward, but the maximum query offset must reflect all query sequences concatenated together. HSPs to be indexed by the tree must have all sequence offsets in local coordinates; the tree code will then translate these coordinates to be unique among all concatenated query sequences. The tree implementation contains many assertions, to verify that the ranges of an input HSP are well-formed and are bracketed at all times by the range of a tree node. Being rigorous about this has caught several bugs in other parts of the blast engine

- The actual containment test is completely separate from the tree traversal.

- As mentioned, the tree just reduces the number of times this test is performed. The current containment test is in s_HSPIsContained, and combines the tests for megablast containment (whether two HSPs are within a few diagonals of each other) and ordinary containment (whether an HSP lies within a box formed by another HSP)

- Containment tests for HSPs on different frames of the same query sequence are also tricky to compare. It is possible for two HSPs to be in different frames but for one to still contain the other. The solution is to place the (local) query range of an input HSP onto the strand in which the alignment occurs, and not the exact frame in which it occurs. This has the added advantage of transparently doing the right thing with out-of-frame query sequences, as well as HSPs from a blastn search. In these two cases, there are only the two strands anyway.

Performance and Future Directions

Despite the complexity of the interval tree implementation (blast_itree.c is currently 1036 lines) the performance boost from an interval tree is rarely needed. However, when it is needed the speedup can be very large. We've seen blastn searches of giant contigs where the whole search runs four times faster when an interval tree is used.

There are two potential issues with the current implementation. The first is that it assumes that only the query can contain multiple sequences/frames, while the subject sequence may not; this may change in the future. Also, in situations where the interval tree is used on lists of HSPs, it's expected that afterwards the list will go through a pruning phase where HSPs that start or end at the same offsets as higher-scoring HSPs are pruned. The interval tree knows this, and does not add an HSP that shares a start- or end point with a higher-scoring HSP already in the tree. This keeps the HSP from enveloping other HSPs later, but also means that the tiebreakers used by the tree must match those of the pruning routine that runs later. Note that the process also works in reverse: if an HSP added to the tree shares start or end points with and has a higher score than HSPs already

in the tree, those HSPs are removed from the tree. Since the tree is never rebalanced, removal of an HSP just means redirecting the pointers leading to it, with no attempt made to recycle the memory needed to index the HSP initially. This cycle of insertions and removals could potentially do the wrong thing, where an HSP kills off a lower-scoring HSP already in the tree, and then is itself killed off by a future HSP. Fortunately, since the tree only provides containment decisions and does not actually delete the underlying HSPs, the worst that can happen is that a rare extra HSP will make it into the list that survives the gapped alignment phase, and this extra will get pruned by the end of the traceback stage.

Appendix C: String Matching with Short Patterns

PHI-blast uses several different but related algorithms to scan sequences for a pattern match; the choice of algorithm depends on the length of the pattern and on how much of the pattern is variable-size.

Any pattern under PHI_BITS_PACKED_PER_WORD (=30) in size that has no variable-size regions, and small patterns that have small variable-size regions, use this algorithm. It's easiest to think of sequences in ncbistdaa format throughout, and for integers to be represented in big-endian **bit** order.

Scratch space equal to BLASTAA_SIZE 32-bit words is needed. Word i has bit j set if position j in the pattern ($0 \leq j <$ pattern_length) matches residue i ($0 \leq i <$ BLASTAA_SIZE).

A finite automaton is used to hold the state of the scanning process.

Scanning proceeds as follows:

```
mask = 1 << (pattern_length - 1);
state = 0;
for (letter = 0; letter < seq_size; letter++) {
   state = ((state << 1) | 1) & scratch[seq[letter]];

   if (state & mask) {
      /* pattern has matched the sequence starting at position
        (letter - pattern_length) */
   }
}
```

This is a blazing fast way to replace functions like strstr(). 'state' contains as many bits as are in the pattern, so patterns longer than 30 letters can be matched if multiple-precision versions of '&', '|' and '<<1' are available (as well as making scratch[] an array of multiple-precision integers). Finally, matching a phiblast pattern like [ABC] is easy: just modify how the scratch[] array is set up.

Initializing PHI-blast

Given a phi-blast pattern, the first task is to perform the precomputations needed for database scanning. It is here that PHI blast decides how scanning will be performed. There are three choices:

1. Use the algorithm described above to determine pattern hits, with 'state' being an Int4. The precomputations take place in s_PackPattern and the bottom of SPHIPatternSearchBlkNew

2. Use the algorithm described above to determine pattern hits, but using multiple-precision quantities throughout. This is for patterns that are 'big' but not 'too big'. All precomputation takes place in s_PackLongPattern

3. If the precomputations in 1. or 2. will be hideously slow or memory-intensive, the pattern is considered 'too big' and is prepared using s_PackVeryLongPattern

Before deciding which of the above to use, the pattern is first converted from a string to an 'initial array' in SPHIPatternSearchBlkNew. The initial array contains one word for each position in the pattern that must definitely appear, and one word that represents each variable-size region. For example, the pattern

W-x(7,15)-C-x(3)-C-x(3)-H-x(2)-M

contains 1 word for the W, 7 words for the portion of the x(7,15) that must appear, 1 word for the up-to-8-letter variable region, 1 word for the C, three words for the x(3), etc. Each word is a bitfield with up to BLASTAA_SIZE bits set; a set bit means amino acid (bit number) matches the pattern at that position. Variable-size regions assign -(size of region) to the word reserved for them; later code treats the negative number as a flag. The above example will create an initial array of size 1+7+1+1+3+1+3+1+2+1=21.

The maximum possible initial array size is 330 (=PHI_MAX_PATTERN_LENGTH); note that the largest possible *pattern* can be bigger than this, since variable-size regions are stored in compressed form at this point.

Once the initial array is created, the next step is to figure out how variable-size regions will be treated. This is the point where the code decides how to scan the DB. The initial array is passed to s_ExpandPattern in phi_lookup.c.

This function is a mess, but it appears to do the following:

1. If the initial array does not contain variable-size regions, nothing happens

2. If variable size regions are present, the initial array is converted to a 'final array'. s_ExpandPattern will attempt to concatenate all versions of the initial array, one version for each choice of size for any variable-size region. In the above example, s_ExpandPattern creates a final array of size 20+21+22+23+24+25+26+27+28=216.

If the size of the expanded array exceeds 330 words, s_ExpandPattern will give up and the pattern is declared ‘too big’.

Note that if s_ExpandPattern succeeds, the size of the pattern can become much larger; in the above example, scanning the DB can theoretically take up to 8 times longer, but in practice I doubt it’s this bad. The advantage to using the expanded version is that variable-size regions don’t matter; you can scan all combinations of region sizes and automatically pick whichever version works.

So the complete logic is:

```
parse the pattern

create the initial array

call s_ExpandPattern

if (s_ExpandPattern failed) {

   s_PackVeryLongPattern  /* too big */

} else if (final array > 30 letters) {

   s_PackLongPattern /* big */

} else {

   s_PackPattern /* small */
}
```

This logic will automatically pick the fastest valid DB scanner, irrespective of how big the pattern is and how complicated the variable-size regions are.

It can mean surprises, though:

W-x(7,15)-C-x(3)-C-x(3)-H-x(2)-M

is a ‘big’ pattern, even though its maximum length is 28, i.e. is less than PHI_BITS_PACKED_PER_WORD. Also,

V-x-H-x(33,40)-C-x(3)-C-x(3)-H-x(2)-M

is considered ‘too big’, even though the maximum possible length is 56, far shorter than the limit of 330.

Scanning the Database for PHI patterns

Whichever scanner is used, the end result of the scanning phase is a list of offset pairs, that specify the start and stop point for a match between the pattern and a database sequence.

The scanner for small patterns uses an internal state of 30 bits or less, and a scratch array of BLASTAA_SIZE words (actually PHI_ASCII_SIZE=256 words, but I think that all the code can work with PHI_ASCII_SIZE=BLASTAA_SIZE). Matches from the pattern scanner in _PHIBlastFindHitsShort are to the entire pattern.

For big patterns, the scanner is in s_FindHitsLong; its scratch array is BLASTAA_SIZE entries x PHI_MAX_WORD_SIZE (=11) words (SLongPatternItems::bitPatternByLetter). Otherwise it behaves the same as _PHIBlastFindHitsShort.

Either of these scanners can search through multiple concatentated patterns at the same time; any matches found are narrowed down to the one pattern that matches. Recall that if there are variable-size regions then either of these scanners implicitly is searching using multiple concatentated patterns. The only change to the low-level scanning code is that the variable 'mask' in the first section has multiple bits set, not just one. The set bits occur at the pattern positions that mark the boundary between sub-patterns. The only other change is that

```
if (state & mask) {

   /* pattern has matched the sequence starting at position
     (letter - pattern_length) */
}
```

must be changed to

```
if (state & mask) {

   /* pattern has matched the sequence starting at position
     (letter - (distance from matching bit in mask to
     previous bit set in mask)) */
}
```

The different way of calculating pattern length is what the functions s_LenOf and s_LenOfL do, for the small and big DB scanners respectively.

When the pattern is too big, it is broken up into fixed-size regions that are separated from each other by variable-size regions (possibly of size 0). If a fixed size region is long than 30 letters, it is further divided into as many fixed-size regions of size 30 as are necessary, separated from each other by a variable region of size 0. Each fixed-size region is treated separately, each has its own mask used by the scanner, and each gets its own row of SLongPatternItems::SLL, a table of size 330 x BLASTAA_SIZE Int4's. Hence,

V-x-H-x(33,40)-C-x(3)-C-x(3)-H-x(2)-M

becomes

- one fixed size region of size 30 (V-x-H-x(27))
- one fixed size region of size 6 (x(6))
- one variable size region of size 7
- one fixed size region of size 12 (C-x(3)-C-x(3)-H-x(2)-M)

s_PackVeryLongPattern figures out which of the fixed size patterns are ‘least likely’ to occur, i.e. have the fewest bits set in their group of ‘final array’ words from section 2. This least likely word becomes the ‘head word’ for the pattern. The above example has its last word as least likely.

The process of scanning a sequence using s_FindHitsVeryLong uses the 1-word scanner iteratively. _PHIBlastFindHitsShort is first used to match the least likely word in the pattern. For each of these matches, an attempt is made to extend the match using words to the right and then to the left of the head word. In the above example, the 1-word scanner is used on the last word. For each such pattern hit, the 1-word scanner is used on the stretch of the sequence from 0 to 7+6 words behind the hit. In this particular case, since x(6) will match anything, this region of the sequence will yield 7 more hits. Each of these 7 hits is combined with the hit to the head word. In the next round, each of the 7 combined hits will attempt to use the 1-word scanner to match word 0 immediately behind the hit. Any hits that manage to do so indicate a complete pattern match, and the range of these hits are saved.

I suppose that there’s a tradeoff in runtime between doing things this way and using s_FindHitsLong. The latter will find complete matches without having to iterate, at the cost of needing to track huge amounts of state when matching patterns with variable-size regions (especially *multiple* variable-size regions).

PHI-Blast Preliminary Gapped Alignment

The result of the database scanning is a list of offset pairs for each pattern match. The query sequence is also scanned, and the list of pattern matches to the query is stored in BlastQueryInfo::pattern_info along with the length of the match (to the query sequence).

Using this information, the preliminary gapped alignment phase creates HSPs. For each pattern hit in the query, PHIGetGappedScore will iterate through all the pattern hits to a subject sequence; the combination of query and subject pattern hits is used as the start point for a score-only gapped alignment. The region actually covered by the pattern hits does not contribute to the resulting score. Gapped extensions to the left happen at the pattern start point for query and subject, while gapped extension to the right happen at the end point of the pattern hit. The combined score is compared to the cutoff score for preliminary gapped alignment. Note that there are no containment tests that throw away HSPs.

PHI-Blast Traceback

As with other blast programs, the result of the traceback phase for phi blast is a list of HSPs, each containing an edit script that describes the underlying alignment. The process is controlled by PHIGappedAlignmentWithTraceback.

As with the preliminary phase, gapped extensions happen to the left and to the right, beginning at the boundaries of pattern hits. Unlike the preliminary phase, the gapped extensions produce trace back and use the final X-dropoff scores. The other major difference is that the pattern hit itself must generate an edit script. For each HSP,

PHIGappedAlignmentWithTraceback generates three edit scripts, one for the left extension, one for the pattern hit and one for the right extension. The concatenation of these three edit scripts forms the complete traceback for the given HSP.

s_PHIBlastAlignPatterns generates the traceback from a pattern hit. Given a query sequence and a subject sequence, each containing a pattern hit of a certain size, the objective is to produce an edit script that makes both pattern hits the same length, and does not touch the fixed-size regions of each pattern hit. For example, given a pattern of

V-x-H-x(33,40)-C-x(3)-C-x(3)-H-x(2)-M

there is a hit to gi 6686393:

```
VFHAFGIPEFKNKIDAIPGILNSMWIKTPDEPGKVYNAYCYELCGIGHSLM
V H.................................C   C   H  M
```

and a hit to gi 46199974:

```
 VIHSFWVPGLSGKRDAIPGQTTRISFEPKEPGLYYGFCAELCGASHARM
 V H.............................C   C   H  M
```

Note that these hits are of different size. The rule for aligning pattern hits is that gaps may only occur inside regions that the pattern allows to be of variable size (the dotted regions in the above). Any regions of fixed size may not contain gaps. It is the job of s_PHIBlastAlignPatterns to determine exactly where in the dotted regions that gaps should be placed. Once the patterns are the same length, it is straightforward to convert the sequence of substitutions and gaps to an edit script for the pattern hit.

The process of placing gaps depends on whether the small, big, or ‘too big’ database scanner is in use. For the first two cases, the ‘final array’ created by s_ExpandPattern contains all possible configurations of patterns that can occur, concatenated together. In these cases, locating the pattern that corresponds to the actual pattern hit is just a matter of running _PHIBlastFindHitsShort or s_FindHitsLong on a stretch of the sequence starting at the (previously determined) start point of the hit. However, these functions return the endpoints of any pattern hits found, when what we really want is the offset into the ‘final array’ that describes the pattern letters (stored in SLongPatternItems::inputPatternMasked). Hence, s_PHIGetShortPattern and s_PHIGetLongPattern perform the same function as their DB-scanning counterparts, except that their output shows where to look in the final array.

s_PHIGet{Long|Short}Pattern is run on both the query and the subject sequence hits, yielding two (possibly different) regions in the final array. Given the start and end point in the final array, the next step is to isolate the regions of the pattern hit that are variable size. For patterns that are not classed as ‘too big’, this occurs at the bottom of s_PHIBlastAlignPatterns. Each element of the final array is a bitfield describing which amino acids would match the pattern at that position. If an array element is kMaskAaAlphabetBits, then that

position in the pattern is a don't-care. If a group of don't-care elements in the query sequence has a size different from the corresponding region in the subject sequence, then gaps are needed. In pseudo-code, the process is as follows:

```
[qs]_pat_off = offset into final array that
describes the pattern for {query|subject} sequence

[qs]_off = offset into {query|subject} sequence where
the pattern hit begins

while (pattern letters still remaining) {
   if (final_array[q_pat_off] == kMaskAaAlphabetBits ||
       final_array[s_pat_off] == kMaskAaAlphabetBits) {

      q_size = size of query don't-care region
      s_size = size of subject don't-care region
      if (q_size == s_size) {

         add q_size matches to the traceback

      } else {

         align region [q_off,q_off+q_size] with
                     region [s_off,s_off+s_size]
         add the resulting traceback
      }
      q_off += q_size;
      s_off += s_size;
      q_pat_off += q_size;
      s_pat_off += s_size;

   } else {
      add a match to the traceback
      q_off++;
      s_off++;
      q_pat_off++;
      s_pat_off++;
   }
}
```

The alignment above is performed by s_BandedAlign; this is a banded global alignment (using dynamic programming) that uses the same score matrix and gap open/extend penalties as the rest of the blast search. The aligner only explores a region ±5 letters from the diagonal; there's an implicit assumption that it's not worth a lot of effort to get a rigorously optimal alignment. In the example mentioned previously the result is pretty good:

```
VFHAFGIPEFKNKIDAIPGILNSMWIKTPDEPGKVYNAYCYELCGIGHSLM
V H+F +P    K DAIPG    +  + P EPG +Y  +C ELCG  H+ M
VIHSFWVPGLSGKRDAIPGQTTRISFE-PKEPG-LYYGFCAELCGASHARM
V H.................................C   C   H  M
                           ^     ^ gaps
```

The score from this banded alignment does not contribute to the final score of the HSP; it's more to make the alignment 'look good'.

When the pattern is ‘too big’, the same principles apply but the details are very different. In this case we only have a compressed representation of the pattern available, i.e. a collection of fixed-width pieces separated by variable-width segments. In the previous example, the final array for the pattern contains

- one fixed size region of size 30 (V-x-H-x(27))
- one fixed size region of size 6 (x(6))
- one variable size region of size 7
- one fixed size region of size 12 (C-x(3)-C-x(3)-H-x(2)-M)

As in the DB scanning phase, all fixed-size regions contain at most PHI_BITS_PACKED_PER_WORD (=30) letters.

Given a start point and length for a pattern hit to a sequence, s_PHIGetExtraLongPattern is responsible for recovering the offsets of each of these pieces. After running s_PHIGetExtraLongPattern on the query and subject halves of the pattern hit, a procedure similar to the pseudocode above can isolate the variable-size regions and align them using s_BandedAlign, eventually generating the traceback for the pattern hit (this code is in phi_gapalign.c:573).

Let num_words be the number of fixed-size regions to locate. The output of s_PHIGetExtraLongPattern is a list of num_words offsets into the sequence, where each offset marks the position of the end of the corresponding fixed-size region. In the case of the query sequence above, where the variable-size region has length 36, the three offsets returned are 30, 36, and 51 (the last offset is the size of the final region plus 3 more letters out of the preceding variable-size region). If there are several pattern hits that start at the same point in the sequence, s_PHIGetExtraLongPattern generates them all and selects the first available whose total length matches the desired length of the hit (this length is an input).

s_PHIGetExtraLongPattern is somewhat obscure, but its logic is described below. The output is the first list out of pattern_hits whose total length equals the expected length of the pattern.

Especially when there are large variable-size regions, this code may have to do a lot of work. Further, pattern regions are matched left to right, rather than least-likely-first.

```
total_hits = 1     /* already know where the 0th region goes */

pattern_hits = {length of 0th region}

for (i = 1; i < num_words; i++) {

   new_hits = {}

   for (j = 0; j < total_hits; j++) {

      max_off = highest offset found so far for pattern hit j
      var_size = size of variable region behind region i (may be 0)

      call the 1-word pattern matcher, to find all matches
                     of region i in [seq+max_off, seq+max_off+var_size]

      for (k = 0; k < matches_found; k++) {
         append concatenation of list j and (highest
                        offset of match k) to new_hits
      }
   }
   pattern_hits = new_hits
   total_hits = number of lists in pattern_hits
}
```