BLAST-ing in Parallel: Enabling an Essential Computational Tool to Keep Pace with the Explosive Growth in Biological Sequence Data

By Jeff Krause, Shodor, Durham, North Carolina
and Michael Ly, University of Illinois,
Urbana-Champaign, Illinois
with contributions from Aaron Weeden
and Jennifer Houchins, Shodor, Durham, North Carolina

Introduction

The computational analysis of biological sequence data has become an essential enabling tool across the life sciences. The "genomic revolution" of the past decade or more would not have been possible, or at least would have been too slow to qualify as a "revolution" without a range of bioinformatics and computational biology tools. Arguably the most broadly and heavily used of all bioinformatics tools is the Basic Local Alignment Search Tool (BLAST, Altschul et. al. 1990 and 1997). This tool serves as the de facto first step in the analysis of any newly completed sequence.

The sequence similarity search implemented by BLAST enables researchers to quickly discover structure, function and evolutionary information about newly produced sequence data. Using a tool to query a new, or poorly understood, sequence for similarity against the database of known sequences identifies biomolecules similar to the unknown sequence, many of which have been annotated with information about how they function in living systems. If the similarity is weak, the scientist at least has some leads to direct an experimental exploration of the function of the sequence or its gene product. If the similarity is strong, the scientist may have a very complete picture of the nature and origins of the molecule, without doing any time-consuming experiments.

The dramatic advances in the speed and reliability of automated sequencing machines would have been for naught if our ability to interpret and understand these sequence data couldn't keep pace. As more reference data becomes available in increasingly larger datasets, researchers are discovering additional ways to leverage the information acquired in comparing poorly understood sequences to those that are better understood. This means that the performance of BLAST and other computational tools must continue to improve in order to sustain and advance biomedical science.

The National Center for Biotechnology Information (NCBI, www.ncbi.nlm.nih.gov) has been maintaining repositories of biological data and developing and hosting computational tools to help scientists to utilize and discover the biological meaning in these data since its establishment in 1988. The NCBI BLAST portal (blast.ncbi.nlm.nih.gov) has been the primary interface to BLAST for most

researchers. However, with advances in analytical approaches, proprietary interests and datasets, and growth in data volume, more users are opting to build their own BLAST installations using NCBI's standalone BLAST or a variety of third party versions or variations on BLAST and similarity search, some of which will be mentioned in this module.

As part of its support mission NCBI hosts many useful instructional materials, documentation and supporting tools. For example, the NCBI Handbook is available through the NCBI Bookshelf and provides an overview of the databases and search tools hosted by NCBI. The BLAST Help Manual and other documentation are available through the Help tab of the NCBI BLAST web page. The NCBI C^{++} Toolkit provides a number of libraries to facilitate the construction of custom applications using NCBI software and databases (the documentation manual is here).

The goals of this module are to:

1. Provide some background on the importance of biological sequence similarity
2. Describe the basic BLAST algorithm
3. Study the performance characteristics of the NCBI server-based and standalone BLAST software
4. Consider and test a parallelization approach
5. Study the performance of the mpiBLAST software as a function of the number of compute cores

The Biology of Sequence Similarity

Similarity between biological sequences implies functional and evolutionary relatedness. This stems from the fact that DNA is the molecular repository for hereditary information in living cells. This information is stored in the sequence of four distinct chemical building blocks linked together end-to-end to form long strands of DNA. Various molecular components of the cell are able to interact with portions of the DNA and use the sequence of chemicals in the DNA to direct the production of other molecules that carry out the multitude of physical and chemical processes necessary to sustain life.

While it makes sense that the ability to duplicate and disseminate the molecular instructions within the DNA with high-fidelity would be of paramount importance in perpetuating life, it turns out that a number of mechanisms for producing changes to the DNA have evolved as well. Apparently, the ability to adapt to changes in the natural world by altering the heritable molecular instructions for life is also important.

These basic biological facts have made similarity search among the most widely used computational biology methods, and with sequence data accumulation currently outpacing Moore's law (Nucleic acid in GenBank and Europe, genomic, and protein), this situation isn't likely to change any time soon.

One consequence of the discrete nature of biological sequence space is that these data are directly amenable to computational analysis. A number of metrics and algorithms based in statistics and probability have proven very successful in quantifying similarities and patterns among sequences. These methods effectively filter massive quantities of data down to a small number of sequences most likely to possess biologically meaningful similarity to the sequences being queried. As a result, scientists are able to focus their attention on the most informative comparisons among sequences.

The first step in the analysis of virtually every newly assembled sequence is to carry out a similarity search against databases of known sequences to see what the new sequence looks like. The types of sequences that are found to be similar, and the degree of similarity can provide conclusive information as to the function and origins of the new sequence. Most significantly, this information can be obtained without committing the time and expenses required to characterize each sequence experimentally.

Alignment Scoring

In order to make an assessment of the degree of similarity between two sequences, it is necessary to first align the sequences so as to bring into register those sequence characters that are most likely to share **homology**, or common ancestry. The challenge is that there are a large number of alignments possible for most biologically important sequences. To distinguish between these possible alignments biologists have adopted scoring methods that reflect the molecular processes that cause an organism's DNA to change across generations.

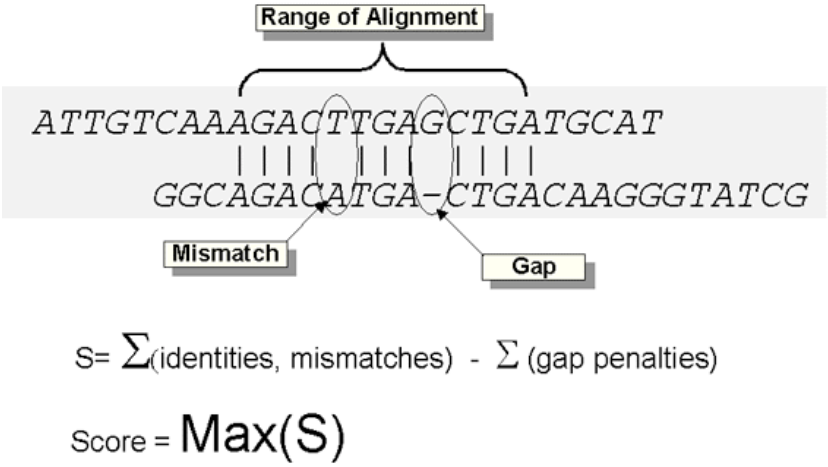


Figure 1 – The score of an alignment is obtained by summing the scores of each column in the alignment. The score for each column is assigned based on whether the characters are identical, mismatched, or a gap has been placed in one of the sequences. (from http://www.ncbi.nlm.nih.gov/Education/BLASTinfo/Alignment_Scores2.html)

Alignment scoring schemes assign more positive scores to aligned characters that reflect more likely evolutionary events. For example, maintaining the character **identity** at a sequence position is the most likely evolutionary event, so a column in an alignment that includes the same character in each sequence would receive a positive score. Less

positive, or more negative scores are assigned to aligned characters that reflect less common evolutionary events. For example, the **substitution** of one character with another is less likely than maintaining the identity of a character, so a column with **mismatched** characters receives a less positive, or more negative score than an identity.

Even more negative scores are given to columns in the alignment reflecting more uncommon events. If it is necessary to introduce a **gap** into one of the aligned sequences in order to maintain a good alignment at neighboring positions, this represents a hypothesis that a character was either deleted in one sequence or inserted in the other. Unfortunately, we can't know which event occurred without seeing the corresponding sequence from the most recent common ancestor of the aligned sequences so we refer to this event as an **indel**. These events are less common than identities and substitutions and are typically scored with a negative valued **gap penalty**.

A more realistic approach to scoring gaps is to impose a large penalty when a gap is introduced. However, a smaller penalty is contributed when another gapped position is appended to an existing gap. This reflects the biology that an insertion or deletion event may be rare, however there is little difference in the likelihood of insertions or deletions involving multiple characters. Including this biological detail in the scoring scheme doubles the number of calculations involved in scoring alignments, since the score at an alignment position will depend on the preceding position.

Problem # 1 – Scoring Alignments

Find the best scoring alignment between the following sequence pairs using the scoring scheme below. To score an alignment, at each position look up the score in the scoring matrix corresponding to the two aligned characters (assume the matrix is symmetric, so the value at the intersection of column G with row C would be the same as the value at the intersection of column C with row G). If a gap is introduced, subtract the quantity (20 + (3 * length of indel)).

As an example, consider the sequences "CAT" and "CAAT". Clearly these sequences are of different length, so an indel will have to be introduced at some point. Keeping the Cs aligned in the first position seems reasonable, and contributes 10 to the alignment score. Likewise, keeping As aligned in the second position contributes another 9. Now comes a decision point: if we align the T with an A this will subtract 12 from the alignment score, and we will then have to include an indel position because of the mismatch in lengths of the two sequences with a cost of (20 + (3*1) = 43. This would give an alignment score of 10+9–12–23 = -16. If instead we put the indel first so that the Ts line-up in the last position the score is 10+9–23+9 = 5. So CA-T over CAAT is the higher scoring alignment.

	A	C	G	T
A	9			
C	-11	10		
G	-3	-12	10	
T	-12	-3	-11	9

Find the best scoring alignments for the sequence pairs:

a) GATGAG and GTGTG
b) CTCGAATTCCG and GTCCTCC
c) AAGATC and ATAAAGCCGTC

Gap penalty = 20 + (3*length)

In general adding realistic details to the scoring scheme will add calculations to the scoring algorithm. However, this will be required to produce biologically meaningful results. For example, a number of distinct molecular processes can result in duplications, inversions, translocations, and recombinations of genomic DNA sequences. As sequences separated by greater stretches of evolutionary time are compared, scoring metrics incorporating these slower (i.e., less frequent = less likely) evolutionary processes need to be utilized in order to recognize these events when they occur.

There are important differences between nucleotide and protein sequence searches that need to be taken into account when preparing to search. Since there are four nucleotides in DNA, and twenty amino acids in protein, an identical match to a **word** four characters in length (or any length, n) is much less likely to occur by chance in an amino acid sequence than in a nucleotide sequence($1/20^n$ vs $1/4^n$). This means that matches of a given length within a protein sequence are more statistically significant, and therefore more likely to be biologically meaningful. As a general rule, for sequences that are known to code for a protein, the amino acid sequence should be used to search for similar sequences in a protein database.

Furthermore, there is more chemical and structural variation among amino acids. As a result some amino acids are able to replace others with little impact on the structure and function of the protein they are a part of. These **conservative** substitutions tend to be observed more frequently than those that disrupt structure or function. As we will see in the next section, BLAST looks for matching words between query and database sequence pairs. However, BLAST also looks for words that include conservative substitutions for characters in the query sequence.

As with any data analysis procedure, the objective is to find a good balance between sensitivity and selectivity so that true hits are found with few false positives. As the biological data sets being analyzed become larger, it becomes essential to maintain the desired analytical performance while increasing the rate at which data can be analyzed.

Sequence data has been accumulating at an exponential rate ever since sequencing was developed in the 1970s, with the result that new sequences are queried against increasingly larger databases. High-throughput sequencing of genomic DNA is accelerating the accumulation of sequence data even further. And this technology is enabling a range of new approaches to biological study that are advancing our understanding at unprecedented rates, scales and levels of detail. For example, in environmental metagenomics, all of the microbial creatures from an environmental sample are sequenced in a single experiment, and similarity search and other computational pattern finding methods are then used to sort and annotate the hereditary material of multiple species. In expressed sequence tag (EST) screening the complete set of expressed mRNAs for a cell or tissue can be sequenced in a single experiment. The identity of each of these transcripts is then determined by similarity search against the

organism's genome to allow scientists to understand, anticipate and intervene in changes in the functional state of the cell, tissue or organism as it responds to changes in the environment, disease or medical treatment.

As the size of the data sets being analyzed, and the number of comparisons being made continue to increase, the rate of progress in contemporary biology is becoming more closely associated with the rate at which high quality similarity search results can be obtained. The potential to reduce the time to complete these calculations through parallel computing is significant. The data intensive nature of this computation would suggest that near linear speed-ups might be possible. The rest of this module will consider the BLAST algorithm and the performance gains that have been observed through one particular parallelization strategy.

The BLAST Algorithm

Algorithms that are assured of finding an optimal scoring alignment between pairs of sequences exist, however their time and memory requirements increase in a quadratic manner, as the product of the lengths of the sequences being compared (order n x m, or $O(n^2)$). As described above, similarity search seeks to identify sequences homologous to a query sequence from within a potentially large sequence database. Because a large number of sequence comparisons are typically required, algorithms for optimal sequence alignment are not practical for similarity search applications.

Instead of exhaustively finding the best alignment by scoring every possible alignment, as the optimal alignment algorithms do, BLAST uses a **heuristic** approach. Heuristic methods reduce the total number of calculations required to find a solution by using insights from theory or practical experience to eliminate some of the possible solutions, and the calculations required to evaluate them.

The insight that BLAST (and other similarity search tools) starts with is this: Sequences with biologically meaningful similarity (homology) to the query sequence will posses some stretches of perfect identity with the query. So, instead of determining the best possible alignment score at every position in the sequence, BLAST takes the approach of finding **words** of length *w* that are common to both sequences, and then extending these matching words until the alignment falls below some user-defined **threshold** value. It turns out that the compute time for this approach grows linearly with the length of the sequences being compared, O(n), rather than quadratically.

Let's take a closer look at the algorithm that BLAST uses to find sequences similar to a query sequence(s) within a database. To make things clearer, we will conduct a search using the server-based BLAST at the NCBI. This link should bring you to a page resembling figure 2, on the following page. From tabs at the top of this page you have access to "Recent Results" that you have produced using BLAST within the last 36 hours, "Saved Strategies" for conducting BLAST searches according to conditions that

you have specified, and the "Help" page, which provides links to extensive documentation and tutorials, along with links to the standalone version of BLAST and downloadable BLAST database files, which we will use later.

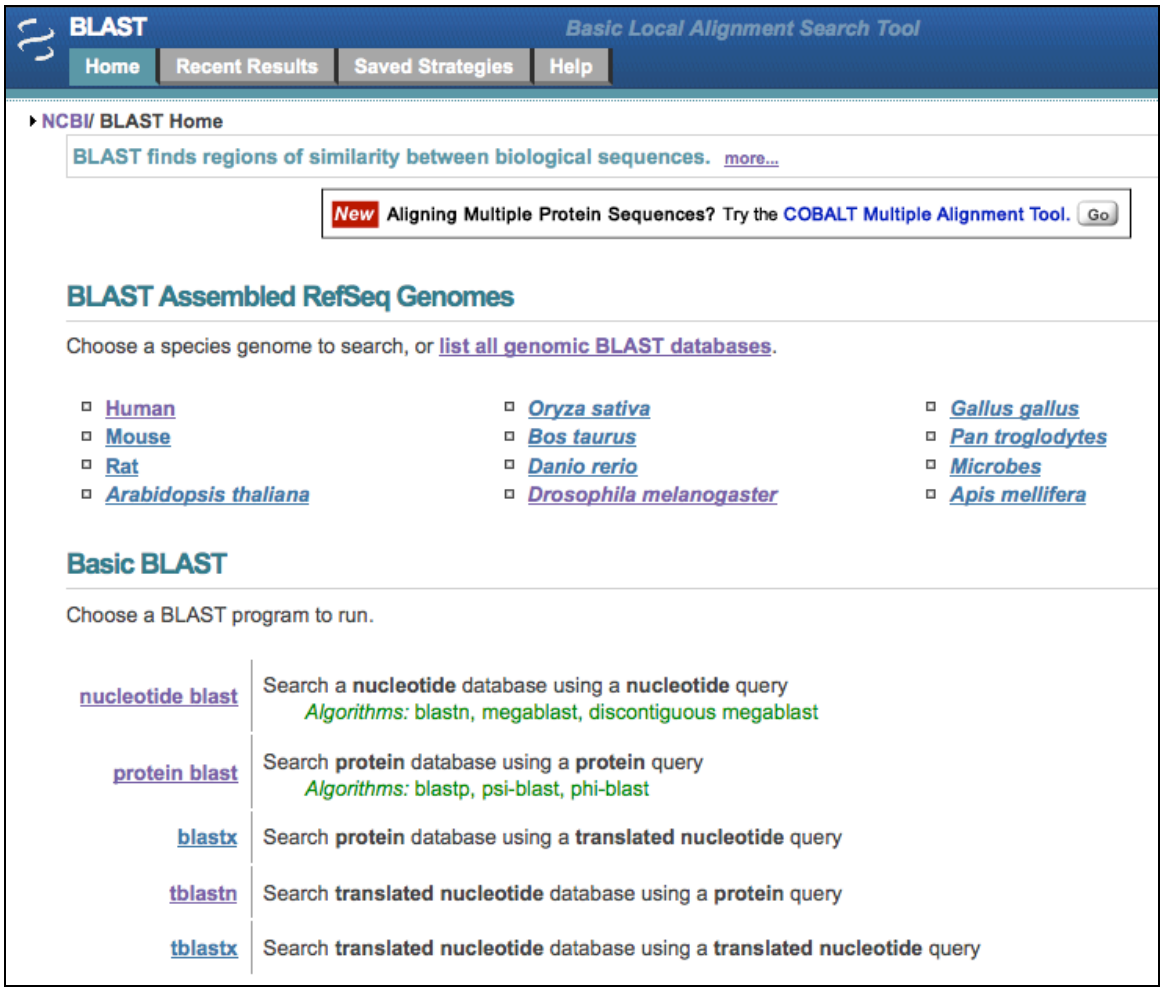


Figure 2 – The NCBI BLAST home page

For this example we will search a protein database using a protein query sequence. In order to do this, click on the "protein blast" link beneath the heading that says "Basic BLAST … Choose a BLAST program to run." The protein BLAST (BLASTp) page will open and is pictured in figure 3, on the next page. This page allows us to specify inputs and parameters for the BLAST search. Information about each of the inputs can be found by clicking on the question mark near the input on the web page. In order to run a search BLAST needs the following inputs:

1. A query sequence or batch – acceptable inputs include identifiers that uniquely identify catalogued sequences, raw nucleotide or amino acid sequences, or sequence files in FASTA or some other formats.
2. A sequence database – Server-based BLAST provides access to all of the major sequence data repositories. The need to search against custom databases is one of the motivations for downloading and running standalone BLAST.

3. Which BLAST algorithm – PSI and PHI BLAST will identify sequences with similarities to patterns in the query sequence, instead of the characters in the sequence.

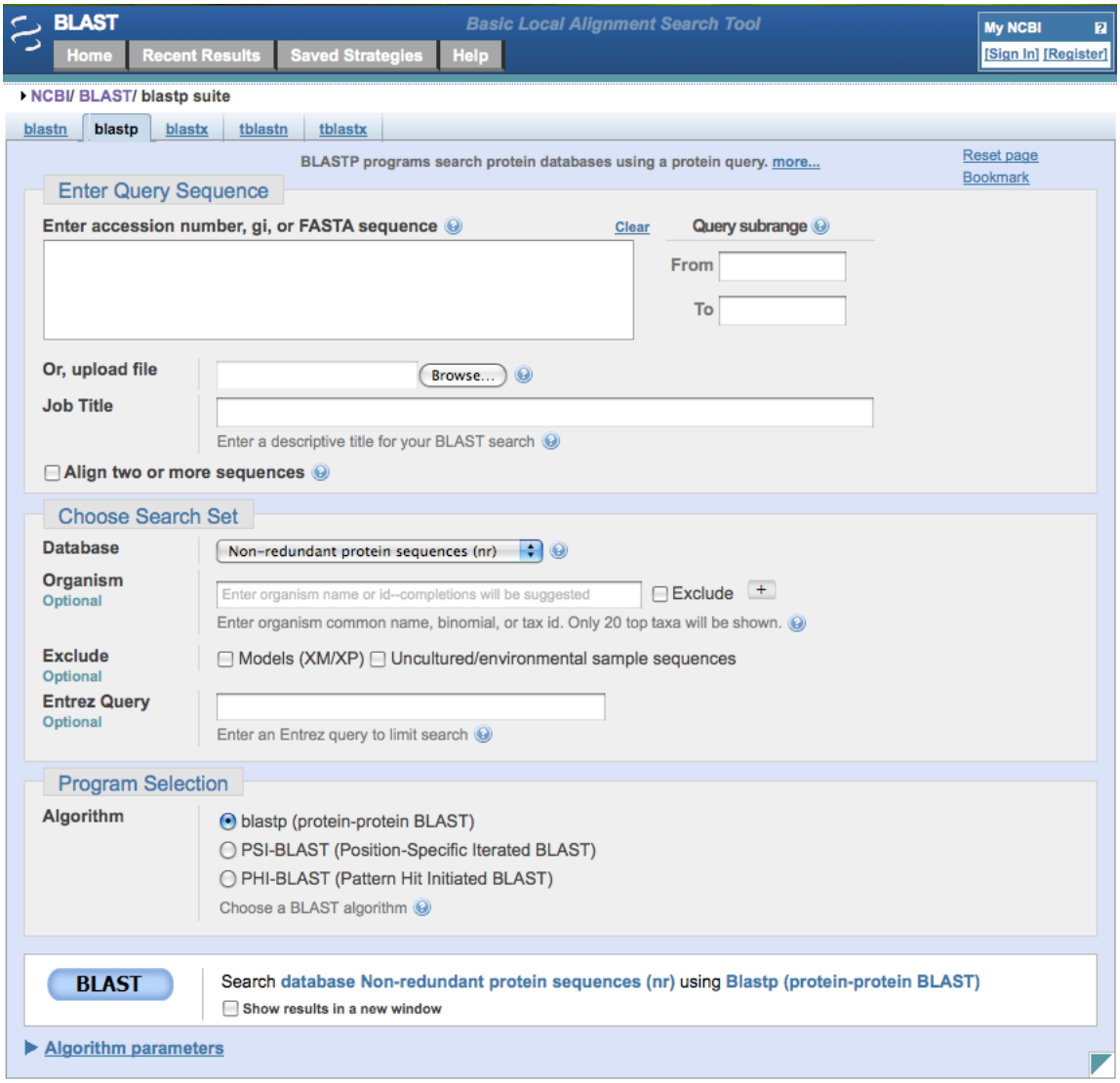


Figure 3 – The BLASTp setup page

In order to run a search, we will need a query sequence. Below is the amino acid sequence for an enzyme called tryptophan synthase from the corn plant. This enzyme catalyzes the last step in the set of biochemical reactions that assemble the amino acid tryptophan, which is represented by a "W" in amino acid sequences (there are only two of them in this sequence, one near the middle of the third line of the sequence, and one at the seventh position of the fourth line of the sequence).

>gi|1174778|sp|P43283.1|TRPB1_MAIZE RecName: Full=Tryptophan synthase beta chain 1; AltName: Full=Orange pericarp 1
GRFGGKYVPETLMHALTELENAFHALATDDEFQKELDGILKDYVGRESPLYFAERLTEHYKRADGTGPLI
YLKREDLNHRGAHKINNAVAQALLAKRLGKQRIIAETGAGQHGVATATVCARFGLQCIIYMGAQDMERQA
LNVFRMKLLGAEVRAVHSGTATLKDATSEAIRDWVTNVETTHYILGSVAGPHPYPMMVREFHKVIGKETR
RQAMHKWGGKPDVLVACVGGGSNAMGLFHEFVEDQDVRLIGVEAAGHGVDTDKHAATLTKGQVGVLHGSM
SYLLQDDDGQVIEPHSISAGLDYPGVGPEHSFLKDIGRAEYDSVTDQEALDAFKRVSRLEGIIPALETSH

ALAYLEKLCPTLPDGVRVVLNCSGRGDKDVHTASKYLDV

This sequence is in a format called FASTA, which is a holdover from a tool for similarity search that preceded BLAST. You will notice that this sequence is comprised of two portions. On the first line is a greater than symbol ">" followed by some numbers and information that describe this sequence. This is the header line. Everything that follows the header line is interpreted as sequence information, until the end of the file. The large text box on the BLASTp setup page is for the query sequence. You can copy and paste the entire sequence (including header line), a portion of the sequence, or one of the sequence identifiers that NCBI can use to find the sequence ("gi|1174778" on the header line).

BLAST provides default values for everything except a query sequence. Having entered a query sequence (or sequence identifier), the search can be started by clicking on the large "BLAST" button on the bottom of the page.

You may also notice a link below the BLAST button that says "Algorithm parameters". While all of these parameters provide the user with greater control of the search process and the results obtained, the two parameters that will always impact search time and results are the "**Expect threshold**" and the "**Word size**". To understand the effects of these parameters, lets consider what happens when BLAST search is initiated by clicking the big "BLAST" button.

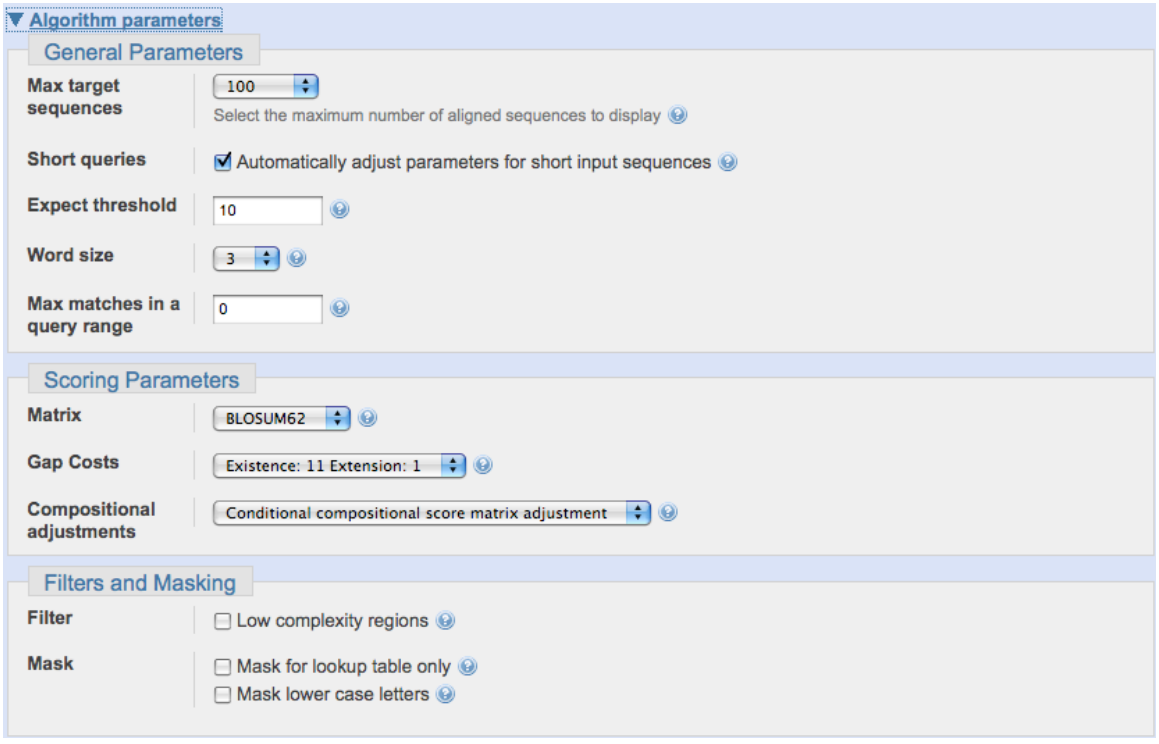


Figure 4 – Algorithm parameters for BLASTp

Build word lists

Instead of aligning sequence pairs, the BLAST algorithm looks for words of a user specified length (*w*), which are present in both the query and database sequence. In order to accomplish this, BLAST first carries out a **hashing** of the sequence data, converting the sequences to word lists (see figure 5 below). All of the possible words of the specified length with the appropriate alphabet (amino acids, or nucleotides) are first enumerated as a **look-up table**. Then, for each sequence, starting at the first position and proceeding to the end of the sequence (-*w*+1), the sequence ID and starting position of each word in the sequence are recorded with the corresponding word in the look-up table. The result is a list of words associated with the sequence ID and starting positions of all of the words that occurred within the sequence recorded.

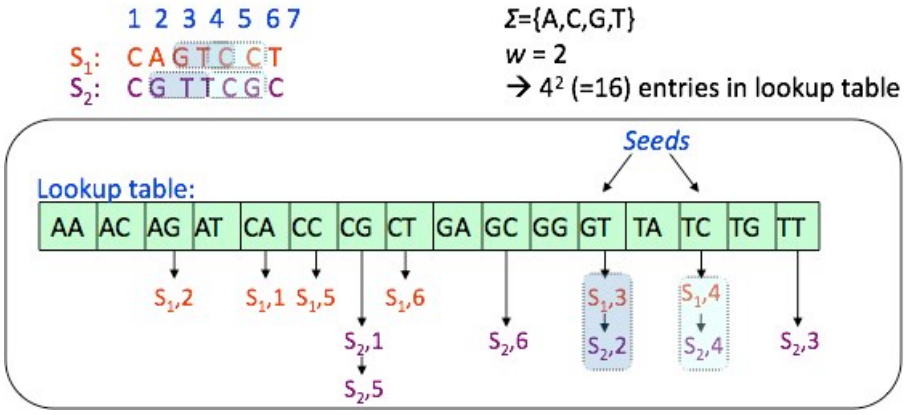


Figure 5 – Hashing sequences to word lists to identify word matches. Two nucleotide sequences, S_1 and S_2 are hashed into a list of words of length (w) = 2. The sequence identity and position of each word is recorded. Word look-ups with occurrences in both sequences are then identified.

Traditionally, the database comprises much more sequence data than the query (though this is less true now than in the past). Construction of the look-up table for the database is a pre-processing step, which will only be done once for a given database. BLAST saves the look-ups for each database it uses so that they will not have to be reconstructed for each search. At NCBI these database look-ups are constructed off-line and the BLAST servers have access to the latest builds for searching.

Identify seeds

Once word lists for all of the sequences have been constructed, finding matches involves going down the list of words present in the query sequences and looking to see if there are any occurrences in the list for the database sequence.

Extend seeds

Having identified all word matches between a query/database sequence pair, the next step is to extend an alignment in both directions from the matching words. This step resembles the optimal pairwise alignment algorithms that were described as too slow for

similarity search. They are still the best way of producing high-quality alignments in regions deemed of high enough potential value to dedicate the necessary compute cycles. The speed gains in the BLAST algorithm result from filtering out low value regions, not worth extending, based on the absence of matching word pairs.

The original BLAST algorithm would extend on every word match, and even allowed for conservative substitutions in the case of amino acid sequences (browse the BLAST documentation here). However, the gapped BLAST algorithm adopted the "two-hit" method, which requires that two non-overlapping word pairs must occur within a specified distance of one another along the same diagonal of an **alignment matrix** before extension is initiated (see figure 6 below). An alignment matrix is a representation of the alignment of two sequences where one sequence is arrayed across the top of a grid, and the other down the left side, so that areas of correspondence between the two sequences appear as diagonals from top left to bottom right.

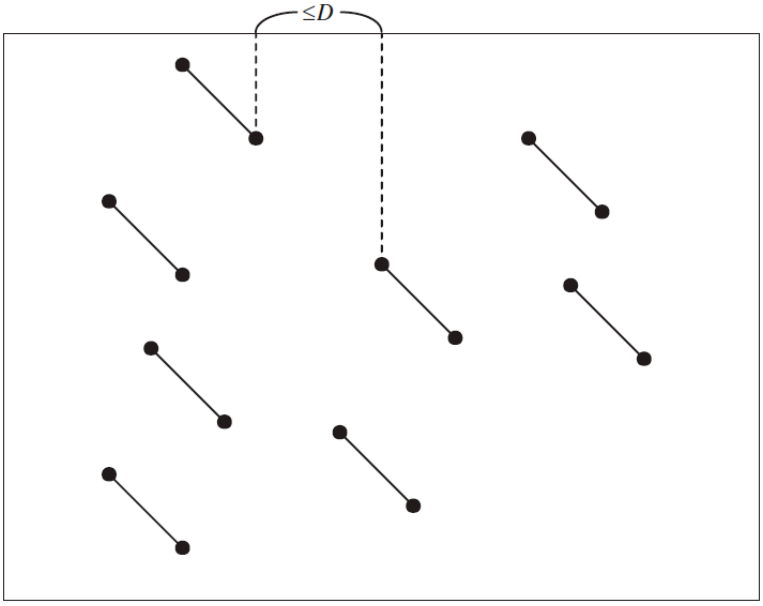


Figure 6 – The two-hit criterion in gapped BLAST requires that at least two matched word pairs must occur along a diagonal of an alignment matrix in order for extension to be carried out. (from Chao and Zhang, 2008)

As its name suggests, gapped BLAST also differed from the original BLAST in that it was able to include gaps in the alignment during the extension phase, although the two-hit criteria selects in favor of regions that will not require gaps between the word pairs. In both versions of the algorithm, the extension is continued until the alignment falls below a scoring threshold as measured with the scoring scheme specified as an input to the search. The aligned positions are scored by adding up the score across all of the aligned positions, and each position's score is determined by a value in the scoring matrix, when characters are aligned, or by a gap penalty.

Record all high-scoring segment pairs

The only results that are saved by BLAST are those that meet the criteria that their expect value (e-value) is less than the expect threshold parameter (from the alignment parameters page, figure 4). The expect value represents the number of alignments as good as the one observed that would be expected by chance. Reducing the expect threshold would reduce the number of random hits stored in the results, but it would also reduce the number of distantly related sequences that we would find.

By the time the expect threshold comes into play, the majority of the computation for the query/database sequence pair has already been carried out. As a result, this parameter has less of an effect on the run time for the search, but it has a large effect on the size of the results list, and the degree of similarity to the query within the results. While it may seem that one would always want to opt for a very small expect threshold, one valuable feature of conducting similarity searches is often the identification of less-clearly-related sequences in the database.

Rank and report results

Once the results have been saved, BLAST is able to report them in various formats, depending, among other things, on whether they are intended for human or machine consumption. The NCBI Handbook chapter on BLAST provides an excellent description of the standard BLAST results report as well as the alternative formats available.

Standalone BLAST Exercise

Installing Standalone BLAST on a linux x64 cluster (cluster.earlham.edu)
refer to ftp://ftp.ncbi.nlm.nih.gov/blast/executables/blast+/2.2.23/user_manual.pdf *for full documentation*

1. **Create a directory in your home directory to install BLAST locally:**

```
cd ~

mkdir Standalone_Blast

cd Standalone_Blast
```

2. **Get the VMD source file**

```
wget
ftp://ftp.ncbi.nlm.nih.gov/blast/executables/blast+/2.2.23/
ncbi-blast-2.2.23+-x64-linux.tar.gz
```

3. **'Untar' the file and create a directory for databases**

```
tar xzf ncbi-blast-2.2.23+-x64-linux.tar.gz

mkdir db
```

4. **Get a database file from** ftp://ftp.ncbi.nlm.nih.gov/blast/db

```
cd db

wget [url]

tar xf [file_name.tar.gz] or gunzip [file_name.gz]

**eg. wget ftp://ftp.ncbi.nlm.nih.gov/blast/db/nt.00.tar.gz
**eg. tar xf nt.00.tar.gz
```

5. **Look for a gene you want to search for at** http://www.ncbi.nlm.nih.gov/genbank/ **and then copy and paste the FASTA sequence into a file**

```
cd ..

mkdir query

mkdir results

cd query

vim test

copy and paste your fasta sequence in the test file you
just created

**eg. Drosoph Homeobox
```

6. **If the cluster you are using requires you to use a PBS scheduler, create a PBS submission script**

```
cd ..

vim blast.qsub
```

Add these lines to the file:

```
#PBS -l nodes=1
#PBS -o OUTPUT
#PBS -e ERROR

cd $PBS_O_WORKDIR

./ncbi-blast-2.2.23+/bin/blastn -db
~/Standalone_Blast/db/[db_file_name] -query
~/Standalone_Blast/query/test

**eg. ./ncbi-blast-2.2.23+/bin/blastn -db
~/Standalone_Blast/db/nt.00 -query
~/Standalone_Blast/query/test
```

7. Submit your job to the scheduler

```
qsub blast.qsub
```

8. Check the results

```
cat results/results.txt
```

Inside the results file you should find information on the score of the match, the expect, the number of identities, and the gaps. You should also see a graphic representation of the alignment. An example of the result file is shown below.

```
Score =  200 bits (108),  Expect = 5e-48
 Identities = 184/220 (83%), Gaps = 8/220 (3%)
 Strand=Plus/Minus

Query  1998   AGACAGGCAT-CCAAAAAACGACGGAAGTCCCGCACAGCCTTCACCAACCACCAAATCTA  2056
              ||||| || | |||| || ||||| || || ||||| ||||||||||||||||||||||
Sbjct  30696  AGACA-GCCTCCCAAGAAGCGACGAAAATCACGCACCGCCTTCACCAACCACCAAATCTA  30638

Query  2057   CGAGCTGGAAAAGAGGTTTTTGTACCAGAAATACCTCTCACCGGCTGACAGAGACCAGAT  2116
              ||||||||| ||| | ||| |||||||||| |||||||||||||| ||||| ||||||||
Sbjct  30637  CGAGCTGGAGAAGCGCTTTCTGTACCAGAAGTACCTCTCACCGGCCGACAGGGACCAGAT  30578

Query  2117   AGCACAGCAGC-TGGGGCTGACCAATGCGCAGGTCATCACCTGGTTCCAGAACCG-ACGG  2174
               || ||| ||| | ||  ||||||| || |||||||||| ||||| |||||| | | |
Sbjct  30577  CGCCCAG-AGCCTAGGTTTGACCAACGCACAGGTCATCACATGGTTTCAGAACAGGA-GA  30520

Query  2175   GCCAAGCTCAAGAGAGATCTGGAGGAGA-TGAAGGCGGAC  2213
              |||||||||||||||| || || || | ||||||||||
Sbjct  30519  GCCAAGCTCAAGAGAGACCTAGACGA-ACTGAAGGCGGAC  30481
```

Parallelizing BLAST

Introduction

The sequence databases have continuously grown over the years, to the point that some no longer fit within the memory of a single computer. At the same time, the number of queries that may be needed for accurate comparisons of sequences has also increased. To overcome these hurdles, researchers have leveraged **high performance computing (HPC)** to provide a way for multiple computers, each with their own memory, to work in tandem.

HPC utilizes the concept of **parallelism**, in which multiple computers work concurrently to solve a problem faster or to solve a bigger problem. In order for multiple computers to work in tandem, they must have a way of **synchronizing**, such that they are able to communicate the data that results from the calculations they perform. One method for doing this is to use a technique called **message passing**, in which computers are connected via a network over which they can pass messages that contain data used for the execution of the problem. A standard for message passing is provided by the **Message Passing Interface (MPI)**. MPI provides tools for splitting a program among multiple **processes**, which are software entities that execute tasks in parallel. In this lesson we will consider the MPI version of BLAST, **mpiBLAST**.

mpiBLAST works by **fragmenting, or partitioning,** the BLAST database, splitting it up into smaller chunks. One MPI process known as the **master** sends the fragments of the database to the rest of the MPI processes, known as **workers**. Each worker performs a query to its fragment of the database and sends the result back to the master. If there are more fragments still to process, the master sends them to the workers as the workers become available. This paradigm of workload distribution is commonly called the **Master/Worker Model** and while it also has many applications to HPC outside of mpiBLAST it is certainly not the only workload distribution paradigm for MPI programs.

mpiBLAST Exercise

Using mpiBLAST on a linux x64 cluster (cluster.earlham.edu)
refer to www.mpiblast.org/Docs/Install for full documentation

1. **Create a .ncbirc file in your home directory**

```
cd ~

vim .ncbirc
```

2. **Copy the following lines into the .ncbirc file, but replace "mly" with your username**

```
[mpiBLAST]
Shared=/cluster/home/mly/shared/
Local=/tmp/mpiblast/
```

3. **This following step is only needed if mpiBLAST is not loaded by default on the cluster**

```
module load mpiblast
```

4. **Get a database file at** ftp://ftp.ncbi.nlm.nih.gov/blast/db/FASTA/

```
mkdir mpiblast

cd mpiblast

mkdir db

cd db

wget [url]

tar xf [file_name.tar.gz] or gunzip [file_name.gz]

**eg. wget
ftp://ftp.ncbi.nlm.nih.gov/blast/db/FASTA/drosoph.nt.gz
**eg. gunzip drosoph.nt.gz
```

5. **Format the database file, specifying the number of fragments to use**

```
mpiformatdb -i [filepathofdb] --nfrags=[Number of Frags]

**e.g. mpiformatdb -i ~/mpiblast/db/drosoph.nt --nfrags=3
```

6. **Look for a gene you want to search for at** http://www.ncbi.nlm.nih.gov/genbank/ **and then copy and paste the FASTA sequence into a file**

```
cd ..

mkdir query

mkdir results
```

```
cd query

vim test

copy and paste your FASTA sequence in the test file you
just created

**e.g. Drosoph Homeobox
```

7. **Create a PBS Batch submission script**

```
cd ..

vim mpiblast.qsub
```

Add these lines to the file – the values for "nodes=" and "-np" should be the same as the value you picked for –nfrags above

```
#PBS -l nodes=3
#PBS -o OUTPUT
#PBS -e ERROR

module load mpiblast
cd $PBS_O_WORKDIR

mpirun -bynode -hostfile $PBS_NODEFILE -np 3 mpiblast -p
blastn -d [dbname] -i ~/mpiblast/query/test -o
~/mpiblast/results/results.txt

**e.g. mpirun -bynode -hostfile $PBS_NODEFILE -np 3
mpiblast -p blastn -d drosoph.nt -i ~/mpiblast/query/test -
o ~/mpiblast/results/results.txt
```

8. **Submit your job to the scheduler**

```
qsub mpiblast.qsub
```

9. **Check the results**

```
cat results/results.txt
```

The results file should be similar to the one obtained in the Standalone BLAST exercise.

10. See how long it took to run

```
cat ERROR
```

This should show something like the following:

```
Total Execution Time: 0.773756
```

Helpful Links
http://www.ncbi.nlm.nih.gov/staff/tao/URLAPI/unix_setup.html
http://openwetware.org/wiki/Wikiomics:BLAST_tutorial
http://telliott99.blogspot.com/2009/12/blast-ncbirc-file.html
http://debianclusters.org/index.php/Using_mpiBLAST
http://www.ncbi.nlm.nih.gov/genbank/
ftp://ftp.ncbi.nlm.nih.gov/blast/db/

Scaling mpiBLAST

Introduction

As **supercomputers** (the biggest, fastest computers in the world) become larger and faster, the total number of **cores** (processing units that comprise computer chips) available to an HPC application is increasing. In order to take advantage of a greater number of cores, an application must be able to **scale** well, to not only run correctly on a larger supercomputer, but also to better utilize the additional cores that supercomputer provides. Scaling an algorithm usually comes in two forms: **strong scaling**, in which the same sized problem is run over more cores, and **weak scaling**, in which the amount of work per core is kept the same while the problem size and the number of cores is increased. In general, strong scaling means we can solve a problem faster by using more cores, while weak scaling means we can solve a bigger problem with more cores in the same amount of time it would take to solve a smaller problem with fewer cores. In the context of sequence alignment, strong scaling means we can receive faster results from our database queries by utilizing more cores, while weak scaling means we can perform larger queries over bigger databases with more cores in the same amount of time it would take us to perform smaller queries over smaller databases with fewer cores. Since each processor has a decreasing amount of work to perform as the number of processors

increases in strong scaling, there is usually a threshold number of processors beyond which performance decreases due to the predominance of communication costs over computation costs. By way of contrast, the goal in weak scaling is to increase the problem size as the number of processors increases in order that each processor needs to perform approximately the same amount of computation. In the exercise below we will examine how well mpiBLAST helps us achieve strong scaling.

Strong Scaling Exercise

Strong Scaling of mpiBLAST on a linux x64 cluster (cluster.earlham.edu)
refer to www.mpiblast.org/Docs/Install for full documentation

1. **Complete the mpiBLAST Exercise above**

2. **Format the database file to use the number of fragments that corresponds to the maximum number of cores over which we will scale**

```
mpiformatdb -i [filepathofdb] --nfrags=[Number of Frags]

**e.g. mpiformatdb -i ~/mpiblast/db/drosoph.nt --nfrags=12
```

3. **Edit the PBS Batch submission script to start with 1 node (4 cores)**

```
cd ~/mpiblast

vim mpiblast.qsub
```

Change the file to contain these lines – note that we changed the value of "nodes=", added "ppn=", and changed the value of "-np". This will cause a job to run with 4 MPI processes on 4 cores.

```
#PBS -l nodes=1:ppn=4
#PBS -o OUTPUT
#PBS -e ERROR

module load mpiblast
cd $PBS_O_WORKDIR

mpirun -bynode -hostfile $PBS_NODEFILE -np 4 mpiblast -p
blastn -d [dbname] -i ~/mpiblast/query/test -o
~/mpiblast/results/results.txt
```

```
**e.g. mpirun -bynode -hostfile $PBS_NODEFILE -np 4
mpiblast -p blastn -d drosoph.nt -i ~/mpiblast/query/test -
o ~/mpiblast/results/results.txt
```

4. **Submit your job to the scheduler**

```
qsub mpiblast.qsub
```

5. **Check how long it took to run with 4 cores**

```
cat ERROR
```

This should show something like the following:

```
Total Execution Time: 1.96301
```

6. **Record the runtime in a table for later**

4 cores	1.96301 seconds

7. **Edit the PBS Batch submission script to run with 2 nodes (8 cores)**

```
vim mpiblast.qsub
```

Change the values of "nodes and "-np" to match the following:

```
#PBS -l nodes=2:ppn=4
...

mpirun -bynode -hostfile $PBS_NODEFILE -np 8 ...
```

8. **Submit your job to the scheduler**

```
qsub mpiblast.qsub
```

9. **Check how long it took to run with 8 cores**

```
cat ERROR
```

10. **Record the runtime in the table**

4 cores	1.96301 seconds
8 cores	1.1743 seconds

11. Edit the PBS Batch submission script to run with 3 nodes (12 cores)

```
vim mpiblast.qsub
```

Change the values of "nodes and "-np" to match the following:

```
#PBS -l nodes=3:ppn=4
...

mpirun -bynode -hostfile $PBS_NODEFILE -np 12 ...
```

12. Submit your job to the scheduler

```
qsub mpiblast.qsub
```

13. Check how long it took to run with 12 cores

```
cat ERROR
```

14. Record the runtime in the table

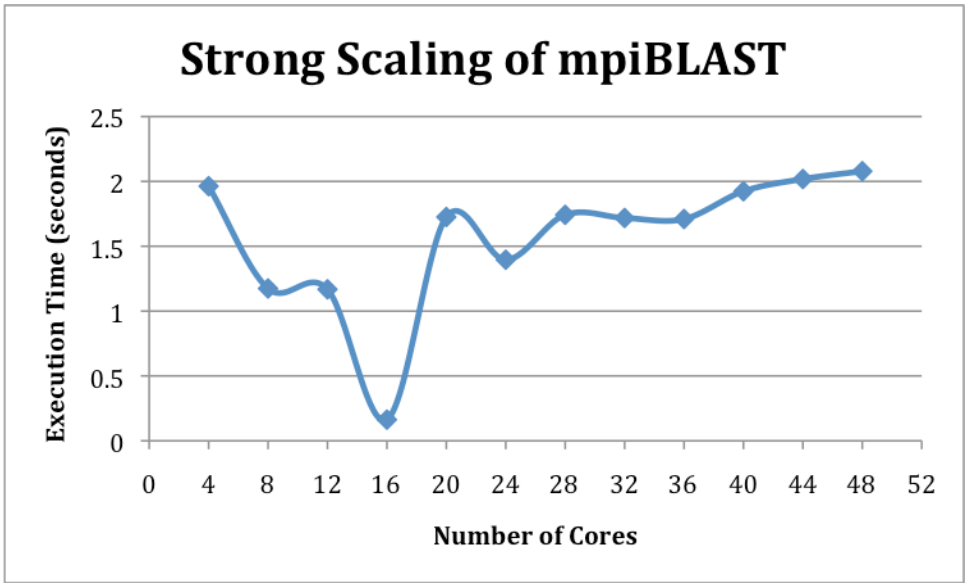
4 cores	1.96301 seconds
8 cores	1.1743 seconds
12 cores	1.16777 seconds

15. Repeat for values of 16, 20, 24, 28, 32, 36, 40, 44, and 48 cores. Fill in these values in the table

4 cores	1.96301 seconds
8 cores	1.1743 seconds
12 cores	1.16777 seconds
16 cores	0.162758 seconds

20 cores	1.72543 seconds
24 cores	1.39609 seconds
28 cores	1.74212 seconds
32 cores	1.71794 seconds
36 cores	1.71004 seconds
40 cores	1.92373 seconds
44 cores	2.0182 seconds
48 cores	2.07888 seconds

If we create a plot of the data above, we get the following:



From this, we can conclude that this particular problem scales strongly up to 16 cores because the amount of computation per processor descreases unless the problem size is also increased. Beyond this, the execution time begins to increase rather than decrease. This is likely due to **communication overhead** in the program, or the time spent waiting for MPI processes to communicate with each other rather than doing meaningful calculations. As the number of processes increases, the number of necessary communications also increases, thus increasing the communication overhead.

Scoring Rubric

Problem #1 can be graded, answers as follows:

a) GATGAG over G-TGTG
b) CTCGAATTCCG over GTCCTCC----
c) ---AAGATC-- over ATAAAGCCGTC

Each of the `results.txt` files from the Standalone BLAST and mpiBLAST exercises can be handed in for a grade. The table from the strong scaling exercise can also be handed in for a grade.

Suggested grading: 10 points for Problem #1, 30 points for each of the exercise deliverables, for a total of 100 points.

Possible Extensions

1. Explore running standalone BLAST and mpiBLAST with different-sized databases. How does this affect the execution time of a query? Perform a weak scaling exercise – vary the size of the database as you vary the number of cores. What is the result?

2. Explore running standalone BLAST and mpiBLAST with different-sized queries on a single database. How does this affect the execution time? Perform a weak scaling exercise – vary the size of the query as you vary the number of cores. What is the result?

3. Explore running standalone BLAST and mpiBLAST with different word sizes (e.g. 2, 3, 4) using the `-word_size` argument to `blastn`. Does mpiBLAST scale the same with different word sizes?

References

Altschul, S. F., Gish, W., Miller, W., Myers, E. W., & Lipman, D. J. (1990). Basic local alignment search tool. *Journal of Molecular Biology*, *215*(3), 403-410. doi:10.1006/jmbi.1990.9999

Altschul, S. F., Madden, T. L., Schäffer, A. A., Zhang, J., Zhang, Z., Miller, W., & Lipman, D. J. (1997). Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Research*, *25*(17), 3389-3402.

Chao, X.M. and Zhang L. (2008). Homology Search Tools. In *Sequence Comparison: Theory and Methods*, p.63-79.

Darling, A., Carey, L. & Feng W. (2003) The Design, Implementation, and Evaluation of mpiBLAST. *4th International Conference on Linux Clusters: The HPC Revolution* in conjunction with *ClusterWorld Conference & Expo*.

Deonier, R. C., Tavaré, S., & Waterman, M. S. (2005). Rapid Alignment Methods: FASTA and BLAST. In *Computational Genome Analysis: An Introduction* (1st ed.). Springer, p.167-194.

Lin, H., Balaji, P., Poole, R., Sosa, C., Ma, X. & Feng, W. (2008) Massively Parallel Genomic Sequence Search on the Blue Gene/P Architecture. *IEEE/ACM SC2008: The International Conference on High-Performance Computing, Networking, and Storage*.

Madden, T. (2003) The BLAST Sequence Analysis Tool. In *The NCBI handbook* [Internet]. McEntyre, J. and Ostell, J. Eds. Bethesda (MD): National Library of Medicine (US), National Center for Biotechnology Information. Available from http://www.ncbi.nlm.nih.gov/bookshelf/br.fcgi?book=handbook&part=ch16

Wu, X. and Tseng, C.W. (2006). Searching Sequence Databases Using High-Performance BLASTs. In *Parallel Computing for Bioinformatics and Computational Biology: Models, Enabling Technologies, and Case Studies*. Wiley-Interscience, p.211-232.