HPC BLAST: User Manual

Application Acceleration Center of Excellence (AACE) and
Intel Parallel Computing Center (IPCC) at the
Joint Institute for Computational Sciences
University of Tennessee

May 26, 2016

Contents

1 Introduction

This document is intended to be a user manual that demonstrates how to compile and execute HPC-BLAST, the high performance implementation of NCBI BLAST developed at the Intel Parallel Computing Center at the University of Tennessee, Knoxville. Since HPC-BLAST compilation will happen as a result of compiling NCBI BLAST, this document also covers the compilation and execution of the reference implementation provided by NCBI. The documentation also covers compilation and execution of mpiBLAST in order to provide a second parallel BLAST implementation for comparison purposes. The targeted architectures covered by the documentation include standard x86 based processors and the Many Integrated Cores (MIC) architecture as realized by the Intel Xeon Phi coprocessor. The documentation also covers usage of auxiliary software tools commonly used in conjunction with NCBI BLAST, tools specific to mpiBLAST, and tools specific to HPC-BLAST. Finally, specific examples will be given to illustrate how to use the BLAST implementations, with priority given to the use of HPC-BLAST. Note, many of the provided usage examples will include commands and tools that are specific to the **Beacon** cluster located at the Joint Institute for Computational Sciences.

A special thanks is due to Dr. Albert Golembiowski of Intel who provided excellent documentation for compiling NCBI BLAST version 2.2.30. The approach Dr. Golembiowski outlined is used as a basis for the compilation guide of BLAST versions 2.2.[28,29,30,31] provided in this document.

1.1 HPC-BLAST Overview

This section is intended to quickly describe the programming model of HPC-BLAST in order to understand how HPC-BLAST allocates work to both threads and processes. HPC-BLAST utilizes the Message Passing Interface (MPI) to distribute search tasks between processes and OpenMP to parallelize searches within each process. At the process level, the central idea is that of a replication group. A replication group is simply a collection of MPI processes (ranks) that each own a subset of the complete database. These database subsets are produced by the NCBI tool *makeblastdb*, included as part of the BLAST toolkit, which partitions the entire set of subject sequences. For a given replication group, each process owns different database components. Within this replication group, each process is given the same set of queries to scan. It is also possible to have a single rank comprise a replication group. In this case, the single rank searches its set of queries against the entire database. HPC-BLAST allows multiple replication groups to be launched so that different replication groups may search only a portion of the entire query input file. The ranks in an HPC-BLAST parallel job also support multithreading. The distribution of work within the rank using threads is similar to how work is distributed at the MPI level. There are three different classifications for the threads that HPC-BLAST can spawn; thread group, team leader, and search thread. Further, all HPC-BLAST ranks in a parallel job have the same distribution of threads. The exception to this is when running in symmetric, or hybrid, mode when all ranks on the host processor have the same thread distribution and all ranks running on the Xeon Phi coprocessors have the same distribution, but the thread distribution between different architectures is not necessarily equal.

The thread classifications are now briefly described. A thread group is a collection of threads, similar to replication groups, where each member of the thread group owns a portion of the database component assigned to the process. As in the case of replication groups, threads in a thread group, called team leader threads, search with the same set of queries. Thus, team leader threads in different thread groups search with different portions of the query set assigned to the MPI process spawning the threads. The team leader threads are also able to spawn search threads. The search threads are the threads that are provided by the NCBI BLAST toolkit. The number of search threads is given on the command line as is done with NCBI BLAST. The particular distribution of threads into the different classifications is done at runtime by creation of a file and command line specification. Examples are given in Section 5.1. If NCBI-compliance is important for your work, the HPC-BLAST development team recommends using only thread groups and search threads in the thread distribution of a parallel job. Here, NCBI-compliance means that the results produced by HPC-BLAST will match the results from NCBI BLAST up to dependency in the order in which the queries were searched. Experiments have shown that the use of team leaders in the thread distribution leads to HPC-BLAST finding more matches than NCBI BLAST. If finding additional matches is not problematic, you may find that using some team leaders leads to better performance than using thread groups and search threads alone. In general, best performance of HPC-BLAST is seen when the thread distribution uses mostly thread groups.

However, when using the maximum number of threads on a particular architecture, it is generally best to allocate 2 to 3 threads in the search thread component and the remaining threads as thread groups. For example, we wish to run 240 threads on an Intel Xeon Phi coprocessor and we want to use two search threads per thread group. Thus, we specify a distribution of 120 thread groups, 1 team leader per thread group, and 2 search thread per team leader, since $120 \times 1 \times 2 = 240$.

2 HPC-BLAST and Tools Source Location

For the most up-to-date version of HPC-BLAST, and associated auxiliary tools we recommend that you download from our Git repository. This can be accomplished via cloning the repository from the command line, or by downloading a zip from our GitHub page.

- Git Repository Clone:
  - Use the following command to clone HPC-BLAST to your machine:
  - `>$ git clone https://github.com/aace/HPC-BLAST.git`
  - Once cloned, you can update the code to the newest version using the following command (when in the HPC-BLAST directory):
  - `>$ git pull`
- Git Zip Download:
  - Simply use the "zip download" option on our webpage at:
  - https://github.com/aace/HPC-BLAST

The sources for all auxiliary tools associated with HPC-BLAST, and described in Chapter 4, can be found at:

`/PATH/TO/REPO/AUXILIARY-TOOLS`

The list of files needed for the auxiliary tools are: *database_sample.c*, *distribute_queries.cpp*, *get_query_stats.c*, *stitch_blast_output.cpp*, *strip_header.c*, *compare_results.c*, *SFMT.c*, *SFMT-sse2.h*, *SFMT-params.h*, *SFMT-params607.h*, and *compile.dat*. The *compile.dat* file is a plain text file that contains instructions on how to build the *database_sample* and *stitch_output* binaries. The files for the remaining binaries (*distribute_queries*, *get_query_stats*, and *strip_header.c*) may be compiled without any special instructions. For example, you may compile on the command line with:

```
$> gcc -O2 strip_header.c -o strip_header
```

3 Compiling BLAST

This section of the user manual focuses on the compilation strategies employed to successfully build working binaries of the BLAST implementations for both Intel Xeon processors and Intel Xeon Phi coprocessors. For the latter, execution is done in native mode; i.e. all execution is done on the coprocessor device. The example build guides assume the use of the Intel compilers and the Intel MPI libraries. The build guides also provide suggested make flags that have been tested to work; others combinations may be possible, but have not been tested.

NOTE: In order to compile NCBI BLAST, and by extension HPC-BLAST for the Xeon Phi, BLAST must first be compiled for the host processor, at least once.

3.1 Compiling NCBI BLAST v2.31

3.1.1 Compiling for the Host

First, NCBI BLAST will be built for the host processor. We may retrieve the NCBI BLAST source tar ball from the command line using:

```
$> wget ftp://ftp.ncbi.nlm.nih.gov/blast/executables/blast+/2.2.31/ncbi-blast-2.2.31+-src.tar.gz
```

Now, uncompress the tar file:

```
$> tar xvzf ncbi-blast-2.2.31+-src.tar.gz
```

and enter the top level directory of the toolkit:

```
$> cd ncbi-blast-2.2.31+-src/c++
```

Next, we copy the files for HPC-BLAST into the appropriate directories of the NCBI BLAST toolkit. For simplicity, we will assume that a local copy of the GitHub repository is available for copying the files.

```
$> cp /PATH/TO/REPO/api/blast_options_handle.cpp src/algo/blast/api
$> cp /PATH/TO/REPO/include/local_blast.hpp include/algo/blast/api
$> cp /PATH/TO/REPO/include/prelim_stage.hpp include/algo/blast/api
$> cp /PATH/TO/REPO/include/traceback_stage.hpp include/algo/blast/api

$> cp /PATH/TO/REPO/api/local_blast.cpp src/algo/blast/api
$> cp /PATH/TO/REPO/api/prelim_stage.cpp src/algo/blast/api
$> cp /PATH/TO/REPO/api/traceback_stage.cpp src/algo/blast/api

$> cp /PATH/TO/REPO/blast/hpc_blastp_app.cpp src/app/blast
$> cp /PATH/TO/REPO/blast/hpc_blastn_app.cpp src/app/blast
$> cp /PATH/TO/REPO/blast/Makefile.hpc_blastp.app src/app/blast
$> cp /PATH/TO/REPO/blast/Makefile.hpc_blastn.app src/app/blast
$> cp /PATH/TO/REPO/blast/Makefile.in src/app/blast
```

Before beginning the compilation, there are two source code modifications that need to be made to ensure working binaries. The first modification is to the `BLAST_GetGappedScore` function in the file:

```
/PATH/TO/ncbi-blast-2.2.31+-src/c++/src/algo/blast/core/blast_gapalign.c
```

Just before the definition of the function, insert `#pragma intel optimization_level 1` (at line 3401). This will force the Intel compiler to drop the optimization level to 1 for this function so that the compiled binaries will operate correctly. The second modification is to the file:

```
/PATH/TO/ncbi-blast-2.2.31+-src/c++/include/corelib/ncbifloat.h
```

Look for the line (line number 71):

```
#  if __cplusplus >= 201103L && defined(_GLIBCXX_CONSTEXPR)
```

and add to it the following:

```
&& !defined (__MIC__)
```

which will disable the macro when compiling for the MIC architecture.

Now, we can begin the compilation. Enter the `c++` directory. The first step is to set up the environment variables:

```
$> export CC=mpiicc
$> export CXX=mpiicpc
$> export LD=xild
$> export AR="xiar crs"
```

Depending on your system's configuration, you may need to execute the Intel compiler suite script to expose the compilers. Alternatively, the full path to the compilers can be substituted. Finally, note that we are explicitly using the MPI variants of the Intel compilers in order to build HPC-BLAST during the NCBI BLAST build. Next, the configuration script is executed:

```
$> ./configure --without-3psw --with-bin-release --with-static-exe --with-mt \
 --without-debug --without-boost --without-strip --with-build-root=< ... >
```

where `< ... >` is the name of the desired build directory and is relative to the `c++` directory. Specifying the root directory of the build is optional. If left off, the configure script will create a default directory labeled `ReleaseMT`, possibly prepended with the compiler version. Now, enter the build directory of the root directory:

```
$> cd ROOT-DIR/build
```

Next, the file `Makefile.mk` is to be edited. The instructions are: Cange the `-O2` flag on lines 93 and 94 corresponding to `CONF_CFLAGS` and `CONF_CXXFLAGS` to `-O3`, `-O` to `-O3` on line 98 corresponding to `CONF_LDFLAGS` Now, the build process can begin with

```
$> make all_r
```

The compilation may require a few hours depending on processor load and system configuration. The process may be dramatically sped up using multiple cores. For example, if we were to compile on a **Beacon** node, we might instead use:

```
$> make -j8 -k all_r
```

where `-j8` instructs the compiler to use up to 8 cores in parallel and `-k` instructs the compiler to ignore warning and continue compiling. After completing, the binaries, including HPC-BLAST, will be placed in:

```
./ROOT-DIR/bin
```

Included in the *bin* directory is the `datatool` binary. This binary is required for the compilation of the Phi native build so it should be moved to a convenient location or the current directory should be noted.

3.1.2 Compiling for the Xeon Phi

This section assumes that all the steps in 3.1.1 have been followed and the `datatool` binary has been successfully built. First, enter into the `c++` directory of the top level source directory:

```
$> cd PATH/TO/ncbi-blast-2.2.31+-src/c++
```

Set up the appropriate environment variables:

```
$> export CC=mpiicc
$> export CXX=mpiicpc
$> export LD=xild
$> export AR="xiar crs"
```

Next, the configuration script is executed:

```
$> ./configure --without-3psw --with-bin-release --with-static-exe --with-mt \
 --without-debug --without-boost --without-strip --with-build-root=< ... >
```

As before, `< ... >` is the name of the desired build directory. *It is highly recommended that a build directory is specified so that the Xeon processor build is not overwritten.* If a build directory was specified for the Xeon compilation, be sure to choose a different build directory for the Phi native compilation. Now, enter the build directory of the root directory:

```
$> cd ROOT-DIR-PHI/build
```

Open `Makefile.mk` in a text editor. Several lines will need to be changed in order to build for the Intel Xeon Phi. Depending on the specific version of the toolkit, the edits are: On lines 93 and 94, change the `-O2` flag to `-O3 -mmic`, corresponding to variables `CONF_CFLAGS` and `CONF_CXXFLAGS`; change line 98 from `-O` to `-O3 -mmic` for the `CONF_LDFLAGS` variable; replace `-axSSSE3` with `-mmic` on lines 220-222 corresponding to the `FAST_CFLAGS`, `FAST_CXXFLAGS`, and `FAST_LDFLAGS` variables Now, set the environment variable to point to the `datatool` created in the host build:

```
$> export NCBI_DATATOOL_PATH=/PATH/TO/DATATOOL/BINARY
```

The compilation can now proceed with

```
$> make all_r
```

Alternatively, use

```
$> make -j8 -k all_r
```

to reduce the compile time. After completion, all Phi native binaries will be located in

```
./ROOT-DIR-PHI/bin
```

3.2 Compiling mpiBLAST

The source code for mpiBLAST version 1.6.0 can downloaded from:

```
http://www.mpiblast.org/Downloads/Stable
```

we may download it directly using:

```
$> wget http://www.mpiblast.org/downloads/files/mpiBLAST-1.6.0.tgz
```

Unpack the compressed file into a convenient location:

```
$> tar xvzf mpiBLAST-1.6.0.tgz
```

Note: Unlike HPC-BLAST, we may compile mpiBLAST for the Xeon Phi without first compiling for the host Xeon processor. However, it is recommended that mpiBLAST be built for the Xeon (or host) processor so that we may have the *mpiformatdb* binary available for database formatting. It is preferable to perform this task on the host processor since this is a serial task.

3.2.1 Compiling for the Xeon Host

Enter the top level directory of the mpiBLAST source directory,

```
$> cd /PATH/TO/mpiblast-1.6.0
```

As in the compilation of HPC-BLAST, mpiBLAST requires that NCBI BLAST is compiled. The appropriate NCBI BLAST version is already packaged with the mpiBLAST tarball. From the top level directory, go into the `ncbi/platform` directory.

```
$> cd ncbi/platform
```

Edit the file `linux_icc.ncbi.mk` by changing, on line 8, `NCBI_AR=ar` to `NCBI_AR=xiar`. On lines 11 and 12, remove the option `-tpp7` from the two variables `NCBI_LDFLAGS1` and `NCBI_OPTFLAG`. Next, go up one level in the directory structure and go into the `make` directory.

```
$> cd ../make
```

Then, edit the file `makedis.csh` by changing, on line 316, from

```
set NCBI_DOT_MK = ncbi/platform/${platform}.ncbi.mk
```

to

```
set NCBI_DOT_MK = ncbi/platform/linux_icc.ncbi.mk
```

Finally, uncomment lines 358 and 359 which set both `HAVE_OGL` and `HAVE_MOTIF` to 0. These changes will ensure that the NCBI BLAST build uses the Intel compilers to build the binaries. Now, move up two directory levels to the root directory and start the build process with:

```
$> ./ncbi/make/makedis.csh
```

Next, we compile mpiBLAST. Set up the appropriate environment variables.

```
$> export CC=icc
$> export CXX=icpc
$> export AR=xiar
```

Then, run the configure script:

```
$> ./configure
```

Next, open the make file in the `src` subdirectory, e.g.:

```
$> emacs src/Makefile
```

Make the following changes to this make file:

- Adjust the compiler on line 108 from `mpicc` to `mpiicc` to invoke the Intel MPI compiler.
- Change the optimization level on line 110 from `-O2` to `-O3`. Optionally, the `-g` flag may be removed.
- Adjust the compiler on line 118 from `mpicxx` to `mpiicpc` to invoke the Intel MPI compiler.
- Change the optimization level on line 120 from `-O2` to `-O3`. Optionally, the `-g` flag may be removed.

Finally, begin the compilation process with `make`.

```
$> make
```

The binaries (`mpiformatdb`, `mpiblast`, and `mpiblast_cleanup`) will be placed in the `src` subdirectory upon completion.

3.2.2 Compiling for the Xeon Phi

Note: The following directions assume a clean build. If you have previously built mpiBLAST for the Xeon processor, it is recommended that you:

- Rename the mpiBLAST binaries for the Xeon so that they do not get overwritten.
- Remove the host object files used by mpiBLAST:

  ```
  $> rm src/*.o
  ```

- Remove the object and source files used by NCBI on the host:

  ```
  $> rm ncbi/build/*
  $> rm ncbi/bin*
  ```

Also, be aware that the following instructions may have to be modified slightly since some of the directions overlap the host build instructions.

Enter the top level directory of the mpiBLAST source directory,

```
$> cd /PATH/TO/mpiblast-1.6.0
```

As in the compilation of HPC-BLAST, mpiBLAST requires that NCBI BLAST is compiled. The appropriate NCBI BLAST version is already packaged with the mpiBLAST tarball. From the top level directory, go into the `ncbi/platform` directory.

```
$> cd ncbi/platform
```

Edit the file `linux_icc.ncbi.mk` by changing, on line 8, `NCBI_AR=ar` to `NCBI_AR=xiar`. Modify line 9 to build for the Phi by inserting `-mmic` immediately after `icc`. On lines 11 and 12, remove the option `-tpp7` from the two variables `NCBI_LDFLAGS1` and `NCBI_OPTFLAG`. Next, go up one level in the directory structure and go into the `make` directory.

```
$> cd ../make
```

Then, edit the file `makedis.csh` by changing, on line 316, from

```
set NCBI_DOT_MK = ncbi/platform/${platform}.ncbi.mk
```

to

```
set NCBI_DOT_MK = ncbi/platform/linux_icc.ncbi.mk
```

Finally, uncomment lines 358 and 359 which set both `HAVE_OGL` and `HAVE_MOTIF` to 0. These changes will ensure that the NCBI BLAST build uses the Intel compilers to build the binaries. Now, move up two directory levels to the root directory and start the build process with:

```
$> ./ncbi/make/makedis.csh
```

Next, we compile mpiBLAST. Set up the appropriate environment variables.

```
$> export CC=icc
$> export CXX=icpc
$> export AR=xiar
```

Then, run the configure script:

```
$> ./configure
```

Next, open the make file in the `src` subdirectory, e.g.:

```
$> emacs src/Makefile
```

Make the following changes to this make file:

- Adjust the compiler on line 108 from `mpicc` to `mpiicc -mmic` to invoke the Intel MPI compiler and build for the Phi.
- Change the optimization level on line 110 from `-O2` to `-O3`. Optionally, the `-g` flag may be removed.
- Adjust the compiler on line 118 from `mpicxx` to `mpiicpc -mmic` to invoke the Intel MPI compiler and build for the Phi.
- Change the optimization level on line 120 from `-O2` to `-O3`. Optionally, the `-g` flag may be removed.

Finally, begin the compilation process with `make`.

```
$> make
```

The binaries (`mpiformatdb`, `mpiblast`, and `mpiblast_cleanup`) will be placed in the `src` subdirectory upon completion.

4 Auxiliary Tools

This section introduces the use of various tools that are either packaged with NCBI BLAST or are part of the HPC-BLAST toolchain. For tools included as part of the NCBI BLAST distribution, refer to the NCBI website (`http://www.ncbi.nlm.nih.gov/books/NBK1763/`) for complete instructions.

4.1 *makeblastdb*

Note: This tool is compiled as part of the NCBI BLAST toolkit.

The `makeblastdb` tool is built as part of the NCBI BLAST compilation and is used to format sequence databases from FASTA format to the appropriate BLAST format used during searches. The simplest example of use is:

```
$> ./makeblastdb -in <DB file> -dbtype <TYPE>
```

where *DB file* is the name of the FASTA database and *TYPE* is either *prot* or *nucl* to indicate protein or nucleotide sequences, respectively. If the database name is *fasta-db* and consists of proteins, the output will be *fasta-db.psq*, *fasta-db.phr*, and *fasta-db.pin*. The three output files correspond to the sequence file, header information file, and the index file for accessing the other files. If the file contained nucleotides instead, the three output files would be of extensions *nsq*, *nhr*, and *nin* instead. The previous example also assumes that the database is small enough, i.e. the index file does not exceed 2GB in size. If instead the database is such that the index file exceeds 2GB, the database of subject sequences is partitioned. Each subset of the partition has the three files (sequence, header, and index) and are labeled by an index starting at 00 and going to $P - 1$, where P is the number of subsets in the partition. Additionally, an alias file is created: either as *fasta-db.pal* or *fasta-db.nal*. The alias file lists all the indices of the subsets in the set so that BLAST can cycle through the partition one subset at a time.

The `makeblastdb` tool can also take the additional argument `-max_file_sz` to limit the maximum file size of any file in the partition. The maximum file size may be specified in bytes or higher units: e.g. 1000000000B, 1000000KB, 1000MB, or 1GB. This is particularly useful for HPC-BLAST as this provides a natural mechanism for breaking up a database into smaller constituent subsets for distribution between MPI ranks. Through trial and error of the appropriate maximum file size, a reasonably balanced partitioning of the database can be generated. When doing so, keep in mind that it is the *.*sq* files that should be balanced as these contain the actual subject sequence contents.

4.2 *blastdbcmd*

Note: This tool is compiled as part of the NCBI BLAST toolkit.

The `blastdbcmd` tool is useful in that it allows the conversion from BLAST formatted files into FASTA format. Suppose we have a BLAST database that we wish to examine in FASTA format. The database files are *fasta-db.psq*, *fasta-db.phr*, and *fasta-db.pin*. Then, to convert them to FASTA format, we would issue:

```
$> ./blastdbcmd -db fasta-db -dbtype prot -entry all -out <file name> -outfmt %f
```

`blastdbcmd` is also useful for querying BLAST formatted databases to determine the number of sequences, longest sequence, and counting the total number of residues. This is done with:

```
$> ./blastdbcmd -db <db name> -dbtype <residue type> -info
```

where the residue type is either *prot* or *nucl*. In the case of a database that is partitioned, the database file name can be either the name of the set, such as *nr*, or a subset of the partition, such as *nr.00*.
The `-info` option is particularly important for HPC-BLAST when employing a distributed database approach; see Section 5.2 for examples. In order for HPC-BLAST to compute the correct E-values, the total number of residues must be known and passed on the command line. For example, suppose there is a BLAST formatted protein database called *seq-db* that has been partitioned into 8 subsets. The total number of residues can be see in the output from:

```
$> ./blastdbcmd -db seq-db -dbtype prot -info
Database: .seq-db.fasta
5,815,854 sequences; 2,006,616,771 total residues

Date: Jun 1, 2014  9:00 PM Longest sequence: 41,943 residues

Volumes:
/PATH/TO/DB/seq-db.00
/PATH/TO/DB/seq-db.01
/PATH/TO/DB/seq-db.02
/PATH/TO/DB/seq-db.03
/PATH/TO/DB/seq-db.04
/PATH/TO/DB/seq-db.05
/PATH/TO/DB/seq-db.06
/PATH/TO/DB/seq-db.07
```

Here, the database contains 2,006,616,771 residues.

Note: The above example assumes that the alias file exists and has not been renamed so that each subset can be read in one at a time. If the alias file has been deleted, the operation can still be performed, but all database subsets of the partition will have to be listed on the command line in the following manner:

```
$> ./blastdbcmd -db "seq-db.00 seq-db.01 seq-db.02 seq-db.03 seq-db.04 \
  > seq-db.05 seq-db.06 seq-db.07" -dbtype prot -info
```

4.3 *strip_header*

Note: This tool is part of the HPC-BLAST toolchain and needs to be manually compiled. Please refer to Chapter 2.

The `strip_header` tool is used to remove excess sequence identifiers from FASTA formatted databases. For the *nr* database, the sequence identifier lines can be several thousand characters long. This results in having the associated **.phr* files be as long or longer than the associated sequence data. The tool can be used to cut off the identifier line after a specified length:

```
$> ./strip_header -i db.fasta -o db.mod.fasta -l XX
```

where `XX` is the number of characters to retain and the `-l` is l as in length.

4.4 *database_sample*

Note: This tool is part of the HPC-BLAST toolchain and needs to be manually compiled. Please refer to Chapter 2.

The `database_sample` tool is used as part of testing the performance of various BLAST implementations. It is largely designed to sample sequences from a FASTA database in order to generate a representative sample that can be used as an input query file. A basic example of usage is:

```
$> ./database_sample -i <input> -o <output> -s <x>
```

where `x` is the number of sequences desired. There are additional optional parameters that can give a more refined search. For example, using

```
$> ./database_sample -i <input> -o <output> -s <x> -f 1000
```

will cause the program to attempt to sample `x` sequences with length `1000`. If there are fewer than `x` sequences of this length, an attempt is made to search within a narrow band around the specified length; i.e., the specified length plus or minus some tolerance. We can also provide a ranged search using the `-b` and `-e` arguments to supply a beginning and ending range to sample from. These arguments can be used together or separately. If only one is used, then the minimum or maximum sequence length is used as the other parameter. Lastly, the total number of residues desired in the output can be specified with `-t`. Using this argument, sampling will terminate once the total number of residues has hit or exceeded the desired amount.

As a final example, suppose we wished to generate a query file consisting of 20,000 residues with all sequences between 500 and 2000 residues in length from the file *db.fasta*. This can be accomplished by:

```
$> ./database_sample -i db.fasta -o query.fasta -s 500 -b 500 -e 2000 -t 20000
```

It is typical to over estimate the number of sequences in the output when specifying a total number of residues, especially when the sequences are sampled over a range of lengths. This is done to ensure that we have enough sequences to write to the query file. When both `-s` and `-t` are specified on the command line, the `-t` option will take precedence over `-s` and sampling will stop once the desired number of letters (residues) has been achieved.

4.5 *distribute_queries*

Note: This tool is part of the HPC-BLAST toolchain and needs to be manually compiled. Please refer to Chapter 2.

In Section 4.4, the construction of a query file by sampling a FASTA database is shown. The generated query file is sufficient for use with NCBI BLAST and other implementations such as mpiBLAST. For HPC-BLAST, a second step is required; at least with the current version. The next step is to distribute the queries for the different replication groups and thread groups within each rank. This is accomplished with the `distribute_queries` tool. `distribute_queries` reads in the query file and then distributes the queries evenly between all replication groups. Within each replication group, the queries are distributed to the thread groups using a simple greedy algorithm. Each thread group is associated with a bucket. At the start of the distribution, all buckets are empty and the queries are sorted by descending length. Next, the code loops over the queries and assigns a query to the bucket with the least residues in it. If after adding a new sequence, the number of residues in a bucket exceeds a threshold, the bucket is marked as full and stops taking new sequences. If all buckets become full and there are still sequences to distribute, the buckets are reset to empty (i.e. number of residues is set to 0) and the process continues. The value of the threshold is dependent on whether the sequences are amino acids or nucleotides. The usage of the application is:

```
$> ./distribute_queries -i <FASTA input> -r X -g Y -t Z -o <output name>
```

where `X` is the number of replication groups in the parallel job, `Y` is the number of thread groups each rank utilizes, and `Z` is either 'a' for amino acids or 'n' for nucleotides. The output name is optional. If absent, the input file name will be used for the output files. Output files are generated for each thread group and each replication group. For example, consider a parallel job with 4 replication groups and 32 thread groups per rank. The name of the input file is *query.prot*. We would then use:

```
$> ./distribute_queries -i query.prot -r 4 -g 32 -t a
```

This would result in output files identified by *query.prot.x.y*, where x (ranges 0..3) corresponds to the replication group and y (ranges 0..31) corresponds to the thread groups.

As a final note, `distribute_queries` also produces the file *query_ids.unsorted*. This file is used to recombine all output from all ranks in the parallel job into the correct order corresponding to the original query ordering before distribution.

4.6 *get_query_stats*

Note: This tool is part of the HPC-BLAST toolchain and needs to be manually compiled. Please refer to Chapter 2.

The `get_query_stats` tool can be used to get high level information on a FASTA file, including the number of sequences, maximum/minimum sequence length, average sequence length, and total number of residues. It can be used as:

```
$> ./get_query_stats query-file.fasta
```

Note: `get_query_stats` can be used on database files in FASTA format to determine the number the number of residues. Here, the output will specify the total number of letters; letters is synonymous with residues.

4.7 *stitch_blast_output*

Note: This tool is part of the HPC-BLAST toolchain and needs to be manually compiled. Please refer to Chapter 2.

The `stitch_blast_output` tool is used to recombine all the output files generated from an HPC-BLAST run into a single, merged output file. This requires that both the *query_ids.unsorted* file from the distribution step and the *job_params* file from the job configuration step (see Section 5.2) are present in the directory where the HPC-BLAST output files are located so that the correct order of sequences can be established. It is used as:

```
$> ./stitch_blast_output <HPC-BLAST output name> <optional new output name>
```

where the HPC-BLAST output name is the name given during the execution of the search.

Note: The `stitch_blast_output` tool only supports output files written with the pairwise format. This is the default behavior of NCBI-BLAST when no output format is specified on the command line. HPC-BLAST has only been tested with the pairwise output but should operate correctly with other output formats. However, there is no supported way of merging the various files written by HPC-BLAST if another output format is desired.

4.7.1 Special Note for Stitching Output

When merging files from parallel jobs that used database decomposition, it is likely that `stitch_blast_output` will generate errors stating that a subject match to a particular query has a 0 length alignment to write to the merged output. This situation arises when there are many subject matches that all have comparable E-values and Bit Scores appearing at, or near, the cutoff for alignment output (by default, NCBI BLAST will output 500 description lines but only 250 alignments). The nature of the problem is that there is not, in general, sufficient floating point precision to fully sort and merge the results.

4.8 *compare_results*

Note: This tool is part of the HPC-BLAST toolchain and needs to be manually compiled. Please refer to Chapter 2.

The `compare_results` tool is used to quantitatively evaluate the differences in sequence alignments between different implementations and versions of BLAST. It reads in two BLAST output files and examines the alignments found for each query. Currently, the tool only supports the ASCII pairwise format that is the default for BLAST. It is used with:

```
$> ./compare_results -r <file1> -R <file2> -o <output>
```

5 Running BLAST

This section demonstrates the basic mechanisms for executing BLAST searches using a variety of implementations. This section also assumes that the binaries have all been built for the appropriate architectures and general familiarity with the tools described in previous sections.

5.1 Running NCBI BLAST

This section describes the basic steps of executing a BLAST search using the NCBI implementation. NCBI BLAST offers a large number of options and parameters that are not covered in this document. Refer to the NCBI website at `http://www.ncbi.nlm.nih.gov/books/NBK1763` for a more thorough introduction. Since running BLAST is functionally equivalent on both the host and Phi architecture and the toolkit version 2.2.31, unified examples are presented. It is assumed that both a BLAST formatted database and query file have been created.

To run a protein search with NCBI BLAST, we can execute with:

```
$> ./blastp -db <db name> -query <query file> -num_threads <x> -out <file name>
```

where `x` is an integer specifying the number of threads to be used by the BLAST engine.

To run a nucleotide search with NCBI BLAST, we can execute with:

```
$> ./blastn -task blastn -db <db name> -query <query file> -num_threads <x> /
-out <file name>
```

Here, it is important to use `-task blastn` in order to use the canonical BLAST search for nucleotides. This is particularly important for versions 2.2.29 and above of NCBI BLAST since this option is necessary to use the contributions made by Dr. A. Golembiowski.

5.2 Running HPC-BLAST

This section demonstrates the steps to run HPC-BLAST for both the host and Phi architectures. Several examples are presented to illustrate how HPC-BLAST is used in a variety of contexts. The examples include a single rank job, multiple ranks each sharing the database, and multiple ranks with replication groups.

5.2.1 Single Rank Example

The first step is to determine how many threads are to be used in the HPC-BLAST run. For example, say we are using the Xeon Phi coprocessor in native mode and want to use a total of 60 threads. Next, we determine how we wish the threads to be distributed in the single rank. Suppose we want to there to be 4 thread groups, 5 team leaders per thread group, and each team leader will use a total of 3 BLAST search threads ($4 \times 5 \times 3 = 60$). To express this for HPC-BLAST, we write the parameters of the job into a file called *job_params*. The *job_params* file is read by HPC-BLAST and by *stitch_blast_output* and is expected to be in the current working directly. For this example, we can express the parallel job configuration with:

```
$> echo 4 5 1 1 > job_params
```

This means that HPC-BLAST will be executed with 4 thread groups, 5 team leaders per thread group, 1 rank per replication group, and 1 replication group. Recall from Section 1.1 thread groups are used to distribute queries within a rank and team leaders are used to distribute the database within a thread group. The *job_params* must be in the directory where HPC-BLAST will be executed. Next, the queries are distributed as shown in Section 4.5.

```
$> ./distribute_queries -i query.fasta -r 1 -g 4 -t a
```

Now, we can start the BLAST search. For example, suppose we are using a compute node on the **Beacon** cluster. We can start the job with:

```
$> micmpiexec -n 1 -host $HOSTNAME-mic0 -wdir /PATH/TO/WORKDIR \
/PATH/TO/WORKDIR/hpc_blastp -db db-name -query query.fasta -num_threads 3 \
-out blast.ouput
```

After HPC-BLAST finishes execution, a single output file will be generated: *blast.output.0.0.0.0.0.*

5.2.2 Single Rank Replication Groups

In the next example, we will utilize multiple replication groups on the same host device. Each replication group will use the same database; thus, only query distribution will occur at the MPI level. Suppose we use 4 replication groups and wish to run on the Xeon host processor. On the **Beacon** cluster, each compute node has dual Xeon E5-2670 processors, each with 8 physical threads and 8 hyper-threads, for a total of 32 threads. So, each rank of HPC-BLAST should be limited to 8 threads, at most, to prevent over-saturating the hardware. We then decide to use 2 thread groups with 2 team leaders and two search threads for each rank. We write these parameters for the run into the *job_params* file:

```
$> echo 2 2 1 4 > job_params
```

This means that HPC-BLAST will run with two thread groups and two team leaders per rank and 4 replication groups, each group comprised of a single rank. Since each rank is its own replication group, the queries can be distributed across the MPI ranks. The queries are distributed with:

```
$> ./distribute_queries -i query.nucl -r 4 -g 2 -t n
```

Next, we can start the job on a single node on the **Beacon** cluster:

```
$> micmpiexec -n 4 -host $HOSTNAME -wdir /PATH/TO/WORKDIR \
/PATH/TO/WORKDIR/hpc_blastn -task blastn -db db-name -query query.nucl \
-num_threads 2 -out blast.output
```

After HPC-BLAST finishes execution, several files will be generated following the convention *blast.output.x.x.0.0.0*, where x corresponds to the replication groups and ranges over [0..3]. Finally, the output created from the HPC-BLAST run can be merged into a single output file with:

```
$> ./stitch_blast_output blast.output
```

For the next example, suppose we will use 8 replication groups each with a single rank. We will also use the Xeon Phi coprocessor in native mode for this example and place 2 MPI ranks on each Xeon Phi device. Since each compute node of the **Beacon** cluster has 4 Xeon Phi coprocessors, a single compute node will be necessary for this parallel job. Since the Xeon Phi 5110p coprocessors available on **Beacon** support 240 hardware threads and two ranks will be launched on each Phi, we will limit each rank to use 120 threads. For this example, we will configure the ranks to use 40 thread groups with 3 team leaders and reflect this decision in the *job_params* file:

```
$> echo 40 3 1 8 > job_params
```

Next, the queries are distributed:

```
$> ./distribute_queries -i query.nucl -r 8 -g 40 -t n
```

To start the parallel job, we can launch with a colon separated list for all hosts, manually create a machine file, or use a **Beacon** specific custom script to automatically create a machine file. We will use the last option. The machine file can be created in an interactive job or in a *qsub* script with:

```
$> generate-mic-hostlist micnative 2 0 > machine.run
```

Note, the above command uses off of the `PBS_NODEFILE` environment variable. The parallel job can then start with:

```
$> micmpiexec -genv I_MPI_PIN_DOMAIN 120:compact -machinefile machine.run \
-n 8 -wdir /PATH/TO/WORKDIR /PATH/TO/WORKDIR/hpc_blastn -task blastn \
-db db-name -query query.nucl -num_threads 1 -out blast.output
```

The *MPI_PIN_DOMAIN* environment variable is used here to endow the threads created by the HPC-BLAST ranks with a compact affinity. After HPC-BLAST finishes execution, several files will be generated following the convention *blast.output.x.x.0.0.0*, where x corresponds to the replication groups and ranges over [0..7]. Finally, the output created from the HPC-BLAST run can be merged into a single output file with:

```
$> ./stitch_blast_output blast.output
```

5.2.3 Multiple Ranks per Replication Group

In the following examples, replication groups consisting of multiple ranks are used. In this situation, different ranks in the replication group(s) search across different databases or separate subsets of a database partition; the latter being the most common example of usage. As seen in Section 4.1, we can use *makeblastdb* to partition a database into pieces that are roughly the same size.

In the first example, we will assume we have a database that has been partitioned into 4 subsets: *db.00*, *db.01*, *db.02*, and *db.03*. As described in Section 4.2, we determine the number of residues in the database:

```
$> ./blastdbcmd -db db -dbtype prot -info
Database: db.fasta
679,231 sequences; 18,394,721 total residues

Date: Feb 4, 2015  11:00 AM Longest sequence: 11,055 residues

Volumes:
/PATH/TO/DB/db.00
/PATH/TO/DB/db.01
/PATH/TO/DB/db.02
/PATH/TO/DB/db.03
```

Since there are 4 subsets in the partition, we will need 4 ranks per replication group unless we want ranks to cycle over multiple databases. Let us also use 4 replication groups for the parallel job. Thus, we will launch with a total of 16 MPI ranks. We will also use the Xeon host processor and map a single rank to each compute node. On the rank level, we will use all 32 threads and configure them as follows: 16 thread groups with 2 team leaders using a single search thread. Now, put these parameters into the *job_params* file:

```
$> echo 16 2 4 4 > job_params
```

Next, distribute the queries:

```
$> ./distribute_queries -i query.prot -r 4 -g 16 -t a
```

Assuming we have submitted a batch job or an interactive job with 16 compute nodes, we can start the parallel job with:

```
$> micmpiexec -n 16 -wdir /PATH/TO/WORKDIR /PATH/TO/WORKDIR/hpc_blastp \
-db db -query query.prot -num_threads 1 -out blast.output -dbsize 18394721
```

Note: Recall that *makeblastdb* creates an alias file when a FASTA database is partitioned into subsets. This file should be deleted, or hidden, to prevent BLAST from attempting to cycle over the entire partition rather than use the database subset HPC-BLAST will assign the rank.

During the execution of HPC-BLAST, each rank will determine which replication group it is in based on its MPI rank and the runtime parameters specified in *job_params*. The ranks also determine which subset of the partition they are responsible for and read that database subset during the search process. For example, rank 7 of the parallel job will be in the second replication group and will search the *db.03* subset.

After HPC-BLAST finishes execution, several files will be generated following the convention *blast.output.x.x.y.0.0*, where x corresponds to the replication groups and ranges over [0..3], and y corresponds to the rank's index within the replication group and ranges over [0..3]. Finally, the output created from the HPC-BLAST run can be merged into a single output file with:

```
$> ./stitch_blast_output blast.output
```

For the second example, we will run natively on the Xeon Phi coprocessors. We will assume the database has been partitioned into 8 subsets and use 5 replication groups. Thus, a total of 40 ranks are used. Further, we will launch 4 ranks per Xeon Phi and limit each MPI rank of HPC-BLAST to use 60 threads configured as 10 threads groups, 2 team leaders, and 3 search threads. Since the database is the same, the number of residues shown in the previous example is still applicable even if the partition has changed. Thus, there is no need to run `blastdbcmd` a second time. These parameters are reflected in the *job_params* file:

```
$> echo 10 2 8 5 > job_params
```

Since we are using 4 ranks per Phi, we will need either a batch or interactive job with at least 3 **Beacon** compute nodes in order to get to 10 Xeon Phis. For this example, let us assume that we get 5 compute nodes because we have decided to only use `mic0` and `mic2` on any given node. Next, the queries are distributed:

```
$> ./distribute_queries -i query.prot -r 5 -g 10 -t a
```

As in the second example of Section 5.2.2, we will use the custom script *generate-mic-hostlist* to automatically generate a machine file to aid in launching the parallel job. Here, we are launching with 4 ranks per Phi:

```
$> generate-mic-hostlist micnative 4 0 > machine.run
```

Next, edit the machine file to remove the lines corresponding to `mic1` and `mic3` on all compute nodes in the job. The parallel job can then start with:

```
$> micmpiexec -genv I_MPI_PIN_DOMAIN 60:compact -machinefile machine.run \
-n 40 -wdir /PATH/TO/WORKDIR /PATH/TO/WORKDIR/hpc_blastp -db db -query query.prot \
-num_threads 3 -out blast.output -dbsize 18394721
```

The *MPI_PIN_DOMAIN* environment variable is used here to endow the threads created by the HPC-BLAST ranks with a compact affinity. And since there are four ranks per Phi, we use all of the 240 hardware threads. After HPC-BLAST finishes execution, several files will be generated following the convention *blast.output.x.x.y.0.0*, where x corresponds to the replication groups and ranges over [0..4], and y corresponds to the rank's index in the replication group and ranges over [0..7]. Finally, the output created from the HPC-BLAST run can be merged into a single output file with:

```
$> ./stitch_blast_output blast.output
```

5.2.4 Checkpointing

If HPC-BLAST stops prior to completion, for any reason, a resubmit of the original job will restart each search (for every replication group, for every rank within that replication group, for each thread group within that rank, and for every team leader within that thread group, there is a search) at the correct position in the search. As HPC-BLAST does its work, it keeps track of where it is, using one restart file per search, following the convention (using the previous configuration as an example) *blast.output.x.x.y.0.0.restart*, where x corresponds to the replication groups and ranges over [0..4], and y corresponds to the rank's index in the replication group and ranges over [0..7]. Note that if HPC-BLAST stops prior to completion, and one desires to resubmit the same job, but from the beginning, the restart files must be manually deleted.

5.3 Running mpiBLAST

This section provides examples on how to use mpiBLAST. It is not intended to be a complete guide to using mpiBLAST. Refer to the official user guide at `http://www.mpiblast.org/Docs/Guide` for further instruction. Also note that mpiBLAST requires a minimum of three ranks in order to run. One rank is the super master process, one rank is the master process, and the remaining rank is the worker process.

Note 1: mpiBLAST uses a modified *.ncbirc* file to read information on where to find files. The basic template of the *.ncbirc* file for mpiBLAST is:

```
[NCBI]
Data=/PATH/TO/mpiblast-1.6.0/ncbi/data
[BLAST]
BLASTDB=/PATH/TO/DB/SPACE
BLASTMAT=/PATH/TO/mpiblast-1.6.0/ncbi/data
[mpiBLAST]
Shared=/PATH/TO/DB/SPACE
Local=/PATH/TO/LOCAL/STORAGE
```

Note that `Data` and `BLASTMAT` paths match as do `BLASTDB` and `Shared`. The `Shared` path is where the database is expected to be for mpiBLAST execution. The `Local` path is used to specify the directory of locally mounted storage, i.e. storage connected directly to compute nodes. In the examples provided below, the use of the virtual database frags flag indicates that we do not wish to load database fragments to local storage but instead load them into RAM.

Note 2: In the examples below, it is assumed that the host version of the `mpiformatdb` binary is used. The Xeon Phi version may be used but it will perform substantially slower due to the serial nature of this application.

In the first example, we will run mpiBLAST using 14 worker processes on a single compute node of the **Beacon** cluster. First, we need to format the database with the tool provided in the mpiBLAST distribution:

```
$> ./mpiformatdb -N 14 -i nr -o F -p T
```

This will partition the *nr* database into 14 subsets. Next, we can launch the parallel job with:

```
$> micmpiexec -n 16 -host $HOSTNAME -wdir /PATH/TO/WORKDIR \
/PATH/TO/WORKDIR/mpiblast --partition-size=15 --use-parallel-write \
--use-virtual-frags -p blastp -d nr -i query.prot -a 1 -o mpiblast.out
```

Note that we launched with 16 MPI ranks: 14 workers, one master, and one super master. The option *–partition-size=15* tells mpiBLAST that a partition has 14 workers (plus one for the master). The options *–use-parallel-write* and *–use-virtual-frags* are suggested by the mpiBLAST documentation for best performance. Similarly, if we ran the same job configuration with nucleotides instead of proteins, we would use:

```
$> ./mpiformatdb -N 14 -i nt -o F -p F
```

And:

```
$> micmpiexec -n 16 -host $HOSTNAME -wdir /PATH/TO/WORKDIR \
/PATH/TO/WORKDIR/mpiblast --partition-size=15 --use-parallel-write \
--use-virtual-frags -p blastn -d nt -i query.prot -a 1 -o mpiblast.out
```

In the above cases, the *-a* option is to tell BLAST how many threads to use in the search.

For a second example, suppose we want to again use 14 worker processes, but now wish to have multiple groups that each have an aggregate copy of the database. This is analogous to the replication groups used in HPC-BLAST. We again partition the database with:

```
$> ./mpiformatdb -N 14 -i nr -o F -p T
```

Now, suppose we have requested 6 compute nodes for the parallel job. In total we will be launching 91 ranks ($6 \times (14 \text{ workers} + 1 \text{ master}) + 1$ super master). Now, we create a machine file that describes how the ranks are distributed to the compute nodes. On the **Beacon** cluster, this might look like:

```
beacon010:15
beacon009:15
beacon008:15
beacon007:15
beacon006:15
beacon005:16
```

Note, that the last compute node has one more rank than the other nodes. This is because mpiBLAST assigns the highest rank number the role of the super master. Distributing the ranks in this way makes sure that each node has a complete replication group. Finally, the job can be run with:

```
$> micmpiexec -n 91 -machinefile machine.run -wdir /PATH/TO/WORKDIR \
/PATH/TO/WORKDIR/mpiblast --partition-size=15 --use-parallel-write \
--use-virtual-frags -p blastp -d nr -i query.prot -a 1 -o mpiblast.out
```

When run with multiple replication groups, mpiBLAST will produce an output file for each group.