Sequence Alignment

Practical Introduction to HPC Exercise

Aims

The aims of this exercise are to get you used to using the command line and a text editor in a terminal window, compiling software, running jobs using the batch submission system, and exploring the parallel performance of a bioinformatics research software package.

Overview

In this exercise you will run the sequence alignment application HMMER to search a protein database for potential matches to a given protein sequence. You will progress through the following steps:

1. Log on to ARCHER2 by connecting to the frontend login nodes;
2. Download the HMMER source code to ARCHER2;
3. Unpack the source code archive;
4. Compile the source code to produce the HMMER executable *phmmer*;
5. Download protein sequence data to ARCHER2;
6. Run HMMER on a login node;
7. Run HMMER in serial on a compute node by submitting a job to the batch system;
8. Run HMMER in parallel on a single compute node using multithreading and investigate the parallel performance;
9. Run HMMER in parallel on multiple compute nodes using MPI and investigate the parallel performance;

We will be using ARCHER2 for this exercise. You can find more details on ARCHER2 and how to use it in the User Documentation available at https://docs.archer2.ac.uk/

Please do ask if anything in the instructions is unclear.

Instructions

Log on to ARCHER2 frontend

Start by logging on to ARCHER2 frontend following the instructions already provided.
See also https://docs.archer2.ac.uk/user-guide/connecting/

Download and extract the HMMER source code

Visit the HMMER website (http://www.hmmer.org), right click on "Download source" and copy the link. This link points to a single archive file, *hmmer.tar.gz,* which contains the latest version of all the files that make up the source code of HMMER.

Now download the source code directly to ARCHER2 from the copied link location using the *wget* command, as follows:

```
user@uan02:~> wget http://eddylab.org/software/hmmer/hmmer.tar.gz
```

Next, unpack the archive:

```
user@uan02:~> tar -xzvf hmmer.tar.gz
```

and enter the directory that was created:

```
user@uan02:~> cd hmmer-3.3.2
```

Aside: examining text files

If you're already familiar with viewing the contents of text files in the terminal you can skip this section

There are a few commands that are useful for displaying the contents of a file. Two such commands are *cat* and *less*. The first, *cat,* simply prints the contents of the file to the terminal window and then returns control to the command line. For example, try:

```
user@uan02:~> cat INSTALL
```

This is fine for small files where the text fits in a single terminal window. For longer files you can use the *less* command, which gives you the ability to scroll up and down in the specified file. For example, try:

```
user@uan02:~> less README.md
```

While examining a file using *less* you can use the 'n' (for "next") key to move a page down, 'p' (for "previous") to move a page up, and down and up arrow keys for line-by-line scrolling. To search for a particular word or phrase, type '/' followed by your search term and press enter. Move through the highlighted search result by pressing 'n' or 'p'. When you have finished viewing the file you can use 'q' to exit *less* and return to the command line.

Compile the HMMER source code to produce HMMER executables

Before we can run HMMER we need to compile the source code we downloaded. The process of compiling and making the resulting executables available to use is also generally known as "building" an application, and is something you may need to know how to do if an application you want to use on an HPC system is not already installed and available by loading a module, or if you have modified existing software or created your own programs.

Start by running the following command:

```
user@uan02:~> module restore PrgEnv-gnu
```

This command is specific to ARCHER2 (actually for all HPC machines from HPE Cray) and sets up your environment to use a particular compiler, namely GCC (Gnu Compiler Collection). The build process itself involves three stages: *configure*, *make*, and *install*.

First, run the following *configure* command to prepare HMMER compilation options:

```
user@uan02:~> ./configure CC=cc MPICC=cc --enable-threads --enable-mpi --enable-sse --prefix=/work/ta017/ta017/$USER/hmmer
```

Now compile the software by running *make*:

```
user@uan02:~> make
```

This turns the source code into the actual HMMER executables that can be run as commands, and should take no more than a couple of minutes. You may see some lines that say "warning:" but these can (usually!) be ignored. Now run the following check to make sure the build completed successfully, which should also take no more than a couple of minutes:

```
user@uan02:~> make check
```

Finally, install the software by running:

```
user@uan02:~> make install
```

This simply copies the built files into the location specified with the --prefix option during the *configure* stage. The HMMER executables should now be available to use, which you can verify as follows:

```
user@uan02:~> ls /work/ta017/ta017/$USER/hmmer/bin/
alimask  hmmconvert       hmmlogo  hmmsearch  makehmmerdb  hmmalign  hmmemit  hmmpgmd  hmmsim    nhmmer
hmmbuild  hmmerfm-exactmatch  hmmpress  hmmstat  nhmmscan    hmmc2    hmmfetch  hmmscan  jackhmmer  phmmer
```

Download protein sequence data

The HMMER executable we'll use for this practical is *phmmer*, which searches a protein sequence database for similarity-based matches to a given protein sequence. Before we can run *phmmer* we will need to download input data, namely a protein sequence database, and a protein sequence to query the dabase with.

Using *wget*, download the compressed protein sequence database containing all Swiss-Prot (annotated) non-human mammalian protein sequences to your home directory on ARCHER2 from the following location:

```
ftp://ftp.ebi.ac.uk/pub/databases/uniprot/current_release/knowledgebase/taxonomic_divisions/uniprot_sprot_mammals.dat.gz
```

Uncompress the file using *gunzip*:

```
user@uan02:~> gunzip uniprot_sprot_mammals.dat.gz
```

This will leave the file uniprot_sprot_mammals.dat in its place. Have a look at the contents of this file using *less*. Next, use *wget* to download the sequence file for the C3 protein (Uniprot ID P01024), which plays a role in the human innate immune system response mechanism, from the following link:

```
http://www.uniprot.org/uniprot/P01024.fasta
```

Set the PATH so *phmmer* is found

For convenience, you can set the PATH environment variable to include where *phmmer* is located by running the following command once in a given terminal session (or in a job script):

```
user@uan02:~> export PATH=/work/ta017/ta017/$USER/hmmer/bin:$PATH
```

The *phmmer* executable will then be found and executed whenever the command *phmmer* is run in the terminal (or in a job script). Otherwise you will need to specify the full path to where *phmmer* is located each time to run it.

Run a serial *phmmer* query on a login node

Now run a *phmmer* query to search for matches in the database for this protein:

```
user@uan02:~> phmmer --cpu 1 -E 1e-100 P01024.fasta uniprot_sprot_mammals.dat
```

The output from *phmmer* shows similarity matches for the human C3 protein in mammalian species other than humans. More matches with lower similarity score can be shown by choosing a smaller value for the –E option. For more information on HMMER and *phmmer* in particular see the User Guide located at http://eddylab.org/software/hmmer/Userguide.pdf. The database, query sequence, and sensitivity chosen above are small enough that *phmmer* will complete very quickly.

Note: the --cpu 1 option ensures *phmmer* runs serially, i.e. on a single thread using a single core on an ARCHER2 login node.

You may briefly already try running in parallel, i.e. with multiple threads each using one core, by specifying a different value for –cpu. However please consider that multiple people logged onto an ARCHER2 login node (128 cores total) each running on many cores or on a single core for more than a few minutes can slow down access for all users, which should be avoided. This caution applies to most HPC systems, and if you leave something running for an extended period of time on a login node you may be asked to stop, or what you are running may be terminated automatically. This is why we should switch to running on compute nodes.

Run a serial *phmmer* query on a compute node

Basic job script

Copy the protein sequence data from your home directory to your space on the work filesystem, namely /work/ta017/ta017/$USER/
Download the following job script to your directory on /work using *wget* and examine its contents:

https://epcced.github.io/20210322-intro-hpc-life-scientists/files/hmmer.slurm

The --noali (“no alignments”) option tells *phmmer* to print less detailed output information.

The –o option outputs results to a named file rather than to *stdout* (“standard out”). Whenever we run an application interactively *stdout* output goes to the terminal window, whereas for a Slurm batch job it ends up by default in a file labelled slurm-######.out, were ###### is a job’s unique numerical ID. However this file only becomes visible after the job terminates. Outputting explicitly to a named file allows us to track progress while the job is running. Even if an application does not have an option for this like *phmmer* does, the output to *stdout* from any command / application can be redirected to a named file using the > symbol, as follows:

```
user@uan02:~> command options input_arguments > named_output_file
```

To submit your job, use the *sbatch* command:

```
user@uan02:~> sbatch hmmer.slurm
```

The job ID number returned from the *sbatch* command can be used to check the status of the job or to delete it.

Aside: using the Emacs text editor

If you already know how to use a basic text editor in the terminal you can skip this section

It is important to be able to use an in-terminal text editor to at least be able to perform basic editing of job script on HPC machine. Certain text editors are available on almost all HPC systems including on ARCHER2, namely Emacs, Vim, and nano. This section describes how to use Emacs, but you should feel free to use whichever editor you prefer. Emacs can be launched with the command *emacs* and the name of the existing file you wish to edit or a new file you wish to create:

```
user@uan02:~> emacs job.slurm
```

The terminal will change to show that you are now inside the Emacs text editor. Typing will insert text as you would expect and backspace will delete text. You use special key sequences (involving the Ctrl and Alt buttons) to save files, exit Emacs and so on. Files can be saved using the sequence "Ctrl-x Ctrl-s" (usually abbreviated in Emacs documentation to "C-x C-s"). You should see "Wrote ./job.slurm" briefly appear in the line at the bottom of the window (the minibuffer in Emacs terms). To exit Emacs and return to the command line use the sequence "C-x C-c". If you have changes in the file that have not yet been saved Emacs will prompt you to ask if you want to save the changes or not.

Run a parallel *phmmer* query on a compute node using multiple threads (--cpu)

Matching the C3 protein against the mammals protein database with default sensitivity as above takes just a second or two even on one core. Try adding the *phmmer* option --max to your job script. This increases sensitivity to potential matches, but you will find your job takes longer to finish.

Now run *phmmer* in parallel by setting both the --cpu option and --cpus-per-task to some identical number between 1 and 128, thereby causing *phmmer* to run with as many threads as that number, each on a separate core. The srun option --hint=nomultithread may seem contradictory, however it should be left in place as part of the launch command as it refers to support for 'hardware-level threads' (simultaneous multithreads - SMT) within the AMD processor, not to *phmmer*'s software threads which we are varying here.

Note: on ARCHER2 the srun parallel application launcher is aware of the Slurm scheduling options so specifying --cpus-per-task is all that is required in this case. On some HPC machines you may need to specify options to srun explicitly. On any machine you want to use you will simply have to find out by reading the documentation provided how the scheduler and parallel application launcher have been configured and how they interact.

Consider this…

The *srun* command launches as many instance of *phmmer* as the value of --tasks-per-node specified in the job script. You could try changing this number to run two instances of *phmmer* in parallel (each with multiple threads). What do you think will happen? To see better what is going on you should remove the explicit output file option -o to *phmmer* and observe the job's overall Slurm output file after it finishes. What do you see, and does this confirm what you thought would happen? Suppose your original sequence aligment query takes a long time to complete, would running it this way, i.e. with --tasks-per-node greater than 1, speed up the solution?

Parallel performance – measuring runtimes & plotting speedups

If you examine the *phmmer* output file for some search query you will see that at the end it contains a line with timing information of the following form:

```
# CPU time: 103.91u 0.11s 00:01:44.02 Elapsed: 00:01:44.09
```

The final value (00:01:44.09 in the above) gives the elapsed wallclock time, that is, how long *phmmer* ran for. CPU time is distinct from walltime. It can be thought of as total core-seconds of time spent in the same way that human work can be measured in total person-hours spent in a job done by multiple people working at the same time, i.e. CPU time measures the cumulative amount of time used by all cores that run an application. In the above example a single core runs the application for ~104 seconds, which is equal to the elapsed wallclock time. The timing for 4 threads might however give a different result:

```
# CPU time: 113.48u 0.16s 00:01:53.64 Elapsed: 00:00:29.86
```

Here 4 cores accumulate a total CPU time of ~113 seconds (more than when the application was run on a single core), or an average of ~28 seconds per core, but because they are all working in parallel *phmmer* finishes in just under 30 seconds, almost 3.5 times faster than in the single core case, which took 104 seconds.

Our goal now is to find out systematically how the performance of *phmmer* varies as increasing numbers of cores are used. You should run a number of multithreaded jobs with different core counts and record the reported elapsed times as in Table 1 below in a file on your own machine. Instead of changing both --cpus-per-task and --cpu manually you can change the former and set the latter equal to $SLURM_CPUS_PER_TASK, so that it automatically obtains the value from the scheduler.

Table 1 - Times taken by multithreaded parallel *phmmer* protein search on a single compute node:

# Cores	Elapsed run time
1	
2	
4	
8	
16	
32	
64	
128	

Once you have completed Table 1 you can compute the speedups of the multithreaded runs as per Table 2. The speedup is the ratio of runtime on 1 core to the runtime on *N* cores.

Table 2 - Speedups for multithreaded parallel *phmmer* protein search on a single compute node:

# Cores	Ideal Speedup	Observed speedup
1		
2		
4		
8		
16		
32		
64		
128		

If you produce a speedup plot with number of cores on the x-axis versus speedup on the y-axis that will clearly show the trend in parallel performance with increasing numbers of cores. Feel free to use whatever plotting method you are comfortable with. To aid you with plotting graphs we provide a small Python plotting utility, plotxy.py, that you can download from the following location:

https://epcced.github.io/20210322-intro-hpc-life-scientists/files/plotxy.py

If you want to run this on ARCHER2 you should first load the cray-python module (module load cray-python). If you put your data in CSV (comma-separated value) form in a file then it will plot them for you. For example, to plot the speedup data you would create a file with three values per line: the number of cores followed by the ideal and observed speedup values for that core count:

```
<cores>, <ideal speedup>, <observed speedup >
```

The first couple of lines would look like:

```
1, 1.000, 1.000
2, 2.000, 1.93
```

Once you have the data in a CSV format you can plot it with the following command:

```
python plotxy.py speedup.csv speedup.png
```

The utility will save a PNG image containing the plot in the second file name you specify on the command line. This is a very simple plot to allow you to quickly preview results – if you were plotting for including in a report or paper you would produce something more elaborate (including axis labels, proper series labels, etc.).

Run *phmmer* on more than one compute node using MPI (--mpi)

So far we have used *phmmer* in parallel by allowing it to spawn multiple threads. However because of the nature of threads this only ever allows us to use the cores on a single node on ARCHER2. If we want to solve a more complex sequence alignment problem in the same time or faster we will need to use all the cores on many nodes in parallel. This can be accomplished using the MPI-parallelised functionality of *phmmer*.

Download (and then unpack) the entire Swiss_Prot database (i.e. not just non-human mammals) from
ftp://ftp.ebi.ac.uk/pub/databases/uniprot/current_release/knowledgebase/complete/uniprot_sprot.fasta.gz

To run MPI-parallel *phmmer* you need to replace the --cpu option with the --mpi option (but without specifying a number). In addition, you should now set --cpus-per-task always equal to 1 since we are no longer using multithreading, and set --tasks-per-node equal to the number of cores you want *phmmer* to run on. For

example, to match the same C3 protein query against the entire Swiss-Prot database (not just mammals) using 512 cores on 4 nodes, set --nodes=4, --tasks-per-node=128, --cpus-per-task=1, and launch *phmmer* as follows:

```
srun --distribution=block:block --hint=nomultithread phmmer --mpi --max --noali -E 1e-100 P01024.fasta uniprot_sprot.fasta
```

As opposed to the multithreaded case, srun will now launch a number of separate but concurrently running instances of *phmmer* – these are separate processes (as opposed to threads). These processes communicate with each other using messages (MPI) to distribute the sequence alignment work. In general applications can combine various implementations of parallelism, including threads and separately running processes that communicate using MPI, however as it happens *phmmer* forces us to choose either the one or the other.

Note: you will not be able to run *phmmer* with a single MPI process (--nodes-per-task=1) as the developers have not provided for this

Modify your job script to match C3 against this larger database by running on multiple nodes. Explore the parallel scaling performance by gathering data for Table 3 and Table 4 and plotting speedups. **To calculate speedups for MPI-parallel runs, divide the elapsed time for a given run by the runtime for 2 processes – the reason why will be explained in the wrap up discussion. This also allows a fair comparison between the performance of multithreaded *phmmer* and MPI-parallel *phmmer* for a given alignment query.**

Finally, to get an overview of parallel performance, you could compute and plot the parallel efficiency for both MPI-parallel and multithreaded *phmmer*.

Table 3 - Times taken by MPI-parallel *phmmer* protein search:

# Nodes	# Cores	Elapsed run time
1	1	n/a
1	2	
1	4	
1	8	
1	16	
1	32	
1	64	
1	128	
2	256	
4	512	

Table 4 - Speedups for MPI-parallel *phmmer* protein search:

# Nodes	# Cores	Ideal Speedup	Observed speedup
1	1	n/a	n/a
1	2	2	1*
1	4		
1	8		
1	16		
1	32		
1	64		
1	128		
2	256		
4	512		