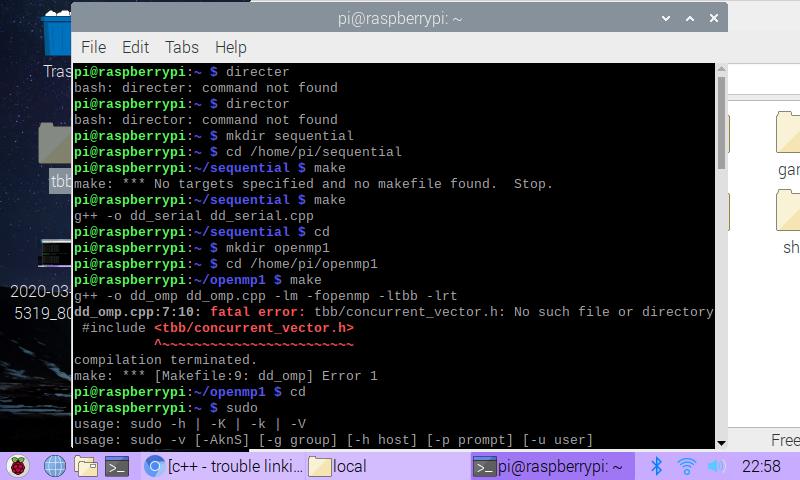
Zoe Kosmicki

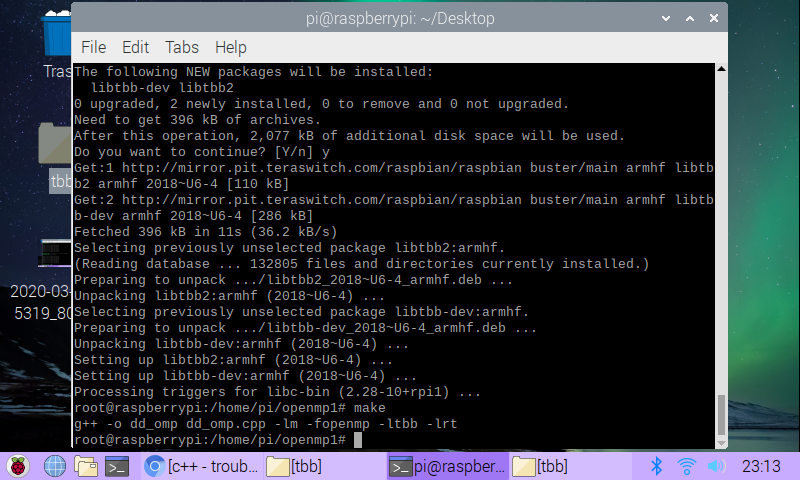
Assignment 5, Task 3

Part A

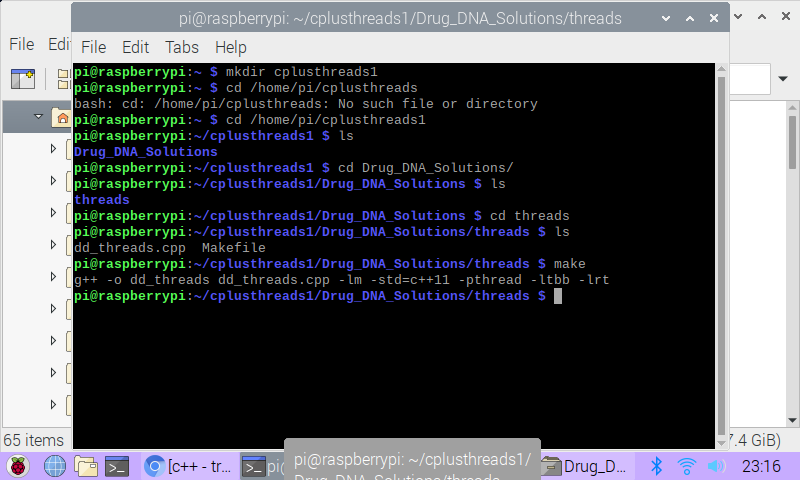
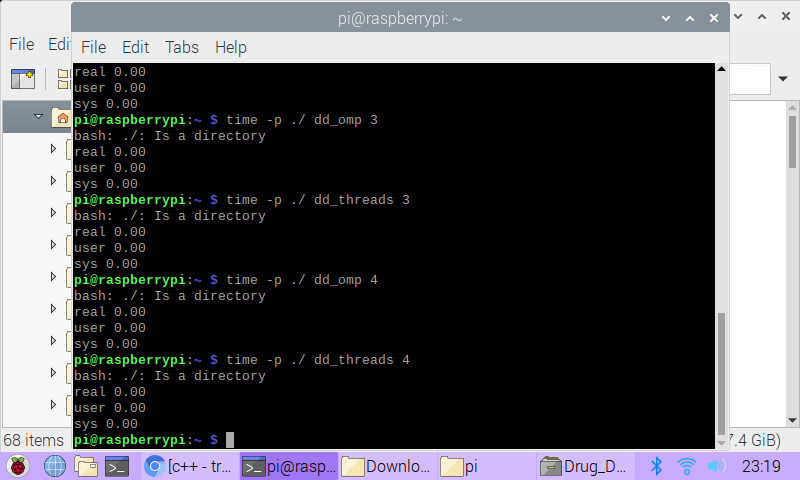
* What are the basic steps (show all steps) in building a parallel program? Show at least one example.
  + Step 1: find and identify tasks that can be computed in parallel
  + Step 2: make sure all tasks are independent of each other
  + Step 3: pick parallel pattern and implement into program
  + If you had to do calculations on two or more sets of data that are relevant to each other in a scientific sense but not a parallel sense, you could set the data up in an array, separate it into subarrays, and assign master/worker threads to do the needed operations on the data all at once, so it gets done faster.
* What is MapReduce?
  + MapReduce is a program model that allows for large quantities of data to be processed in parallel.
* What is map and what is reduce?
  + Map and reduce make up MapReduce, with a set of data being taken and reassembled into a new data set made up of smaller data chunks called a map, that can then be operated on using the reduce task, processing the data from map.
* Why MapReduce?
  + MapReduce is very good for writing scientific programs and applications because of its ability to work with so much data over a cluster.
* Show an example for Map Reduce?
  + The drug design program in part B.
* Explain in your own words how the MapReduce model is executed?
  + MapReduce starts with taking a large/several set(s) of data and chunking it into smaller pieces, which becomes the map. After the map is formed, the data is broken up even further and processed in parallel during the reduce stage. The reduce stage takes the map data and writes it to output files, finishing the model execution.
* List and describe three examples that are expressed as MapReduce computations.
  + Inverted Index: Takes a text and splits it up into individual words, and then are all grouped by word. The reduce function turns x instances of a word into that word, x times. The output shows how many times each word appears in the text.
  + Distributed Grep: Looks for patterns within data sets and links them together if found. The output is everything that matches the input pattern.
  + Count of URL Access Frequency: A computation that maps out all of the requests for a certain url and then reduces it by adding them all up to give the pair <URL, total count>.
* When do we use OpenMP, MPI, and MapReduce (Hadoop), and why?
  + OpenMP is the general use way of making your code run in parallel because of how robust it is; It has a wide usage range for all types of coding applications, and can be used on just one device (as long as that device has multiple processing cores). MPI is more for very precise scientific data handling, because it requires code to be very streamlined and in-sync to run efficiently. It also runs over more than one machine. MapReduce is used when you have a lot of data that needs to be processed, but doesn’t need streamlined code in the same way MPI does. MapReduce is also very good at continuing to operate normally under very unfavorable conditions, making it incredibly robust and reliable.
* In your own words, explain what a Drug Design and DNA problem is in no more than 150 words.
  + It’s code that can generate ligands for new drugs, test them on various types of proteins, and output which ligand is best suited for use in a given drug. Can be done in parallel to speed up the testing phase, which is very slow if done serially.

Part B

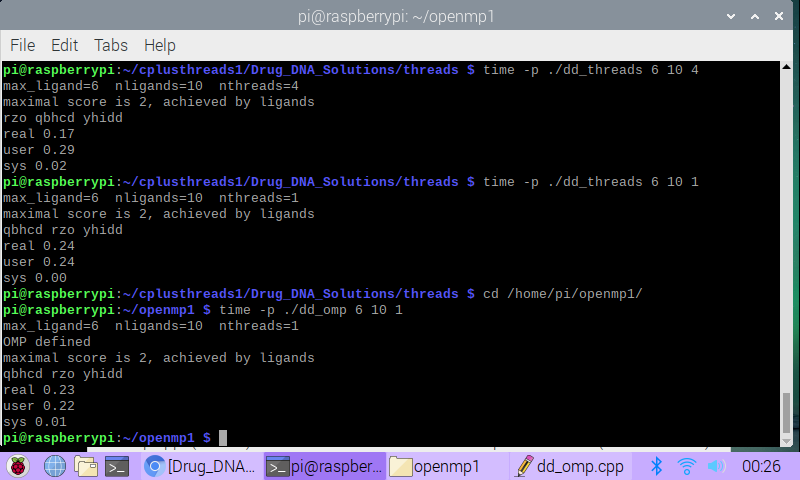
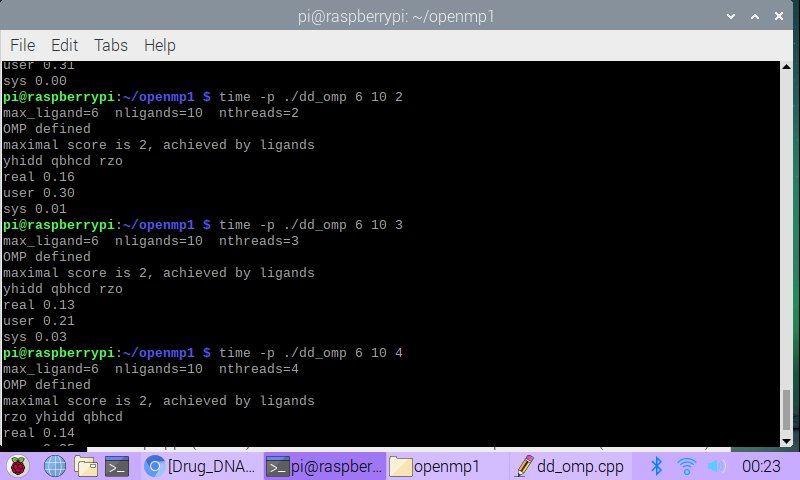
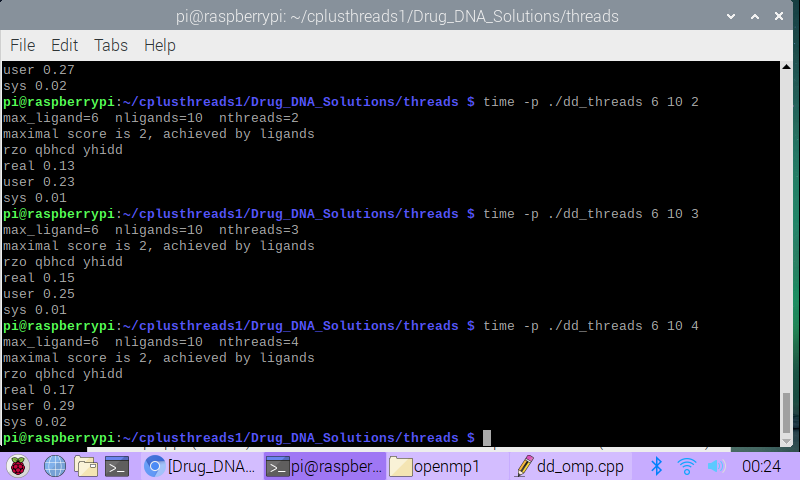
I started by creating directories for sequential and openmp1 for the drug design programs, and using make to compile them and create the executable. I had issues with the OpenMP version because certain libraries weren’t linking correctly, as shown below.I found a fix on Stack Overflow, and was able to install the libraries I needed in order to compile the program. After doing this, dd\_omp compiled just fine.



Next, I created the last directory for the C++11 version, and using the command window compiled it successfully.

Next, I made a few very silly mistakes and was very confused as to why all of the runtimes were 0 seconds. I eventually figured out my errors, and started getting correct times using time -p. The times I found with the correct command usage can be found in the tables below.

|  |  |
| --- | --- |
| **Implementation** | **Time (s)** |
| dd\_serial | 143.54 |
| dd\_omp | 0.23 |
| dd\_threads | 0.24 |

Below are the single thread timed commands I ran to get the values for dd\_omp and dd\_threads above.Next I got the run times for dd\_threads and dd\_omp with 2, 3, and 4 threads.

These times have been put into the table below.

|  |  |  |  |
| --- | --- | --- | --- |
| **Implementation** | **Time (s) 2 Threads** | **Time (s) 3 Threads** | **Time (s) 4 Threads** |
| dd\_omp | 0.16 | 0.13 | 0.14 |
| dd\_threads | 0.13 | 0.15 | 0.17 |

Discussion Questions

1. Which approach is the fastest?

The C++11 was mostly the fastest, with dd\_threads beating it with only two threads running.

2. Determine the number of lines in each file (use wc-l). How does the C++11 implementation compare to the OpenMP implementations?

The C++11 implementation has 137 lines, and the OpenMP implementation has 151 lines.

3. Increase the number of threads to 5 threads. What is the run time for each?

dd\_omp = 0.15 s, dd\_threads = 0.15 s

4. Increase the maximum ligand length to 7, and rerun each program. What is the run

time for each?

dd\_omp = 8.19 s, dd\_threads = 8.13 s

Below is the code snippet for increasing threads to 5 and max ligand length to 7.

