**Victoria Routon**

**Foundation**

**1. What are the basic steps (show all steps) in building a parallel program? Show at least one example.**

Steps in parallelization:

* Decomposition into tasks
* Assignment to processes: This is used to balance the load and maximize locality.
* Orchestration: In this step, name and data are accessed and communicate data. Synchronization is also performed among processes.
* Mapping: In this step, the processes are assigned to the processor.

Example of parallel programming:

* Scientific computation using parallel processing.

**2. What is MapReduce?**

MapReduce is a program model and implementation that essentially deals with Big Data using parallelism.

**3. What is Map and what is Reduce?**

Map is a function that is used to generate and process key/value pairs in the MapReduce program. Reduce is a function that is used to merge or combine values pertaining to their key pairs.

**4. Why MapReduce?**

MapReduce is a very much cost-effective process when implementing it to the traditional databases when it comes to performing computations and storing data, then MapReduce is better than OpenMP.

**5. Show an example for MapReduce.**

Example of MapReduce: MapReduce is implemented with Hadoop to operate on large amounts of data by creating in the format of tuples to make the things simple.

**6. Explain in your own words how MapReduce model is executed?**

The MapReduce program executes in three stages, namely map stage, shuffle stage, and reduce stage. Map stage − The map or mapper's job is to process the input data. Generally, the input data is in the form of file or directory and is stored in the Hadoop File system (HDFS). Shuffle stage- The output of the map will be written into local disk randomly by buffered pairs. Reduce stage- Read the buffered data from local disk and sort it by intermediate keys, after that, return the result to the user.

**7. List and describe three examples that are expressed as MapReduce computations:**

Distributed Grep -The map function emits a line if it matches a given pattern. The reduce function is an identity function that just copies the supplied intermediate data to the output.

Term-Vector per Host - term vector summarizes the most important words that occur in a document or a set of documents as a list of <word,frequency> pairs.

Inverted Index: the map function parses each document and emits a sequence of <word, document ID> pairs.

**8. When do we use OpenMP, MPI and, MapReduce (Hadoop), and why?**

OpenMP: If you want to introduce shared memory parallelism to your code, then OpenMP is an efficient directive-based library that you could use. Using OpenMP, you can put a directive as #pragma parallel forbefore the forloop and it’ll split the loop into multiple threads, so each of them would handle a chunk of the loop’s iterations.

MPI: Message Passing Interface is a distributed memory parallel model implementation, typically used to develop parallel scientific applications. Because MPI can be used to develop pretty much any parallel code that runs over multiple machines, but if the program has stragglers make MPI program inefficient, that is why it is better to use MPI to develop scientific applications.

MapReduce (Hadoop): When you have terabytes of data that you want to do ETL (extract, transform and load) like operations. Because fault tolerance is one of the most important features when using Hadoop.

**9. In your own words, explain what a Drug Design and DNA problem is in no more than 150 words:**

The Drug Design is using computational capabilities such to make sure the effectiveness of the drug remains intact and is able to fight against diseases. DNA problem is the term used for the problems that are inherent to the DNA computers while performing operations related to DNA.

**Programming Drug Design and DNA in Parallel**

I began by installing Threading Building Blocks library to my Raspberry PI (Figure 1).

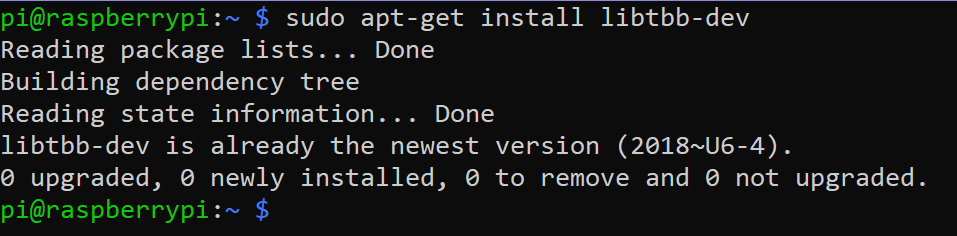


Figure 1 – library is installed

After the library was installed, I created three directories for each of the programs : serial for dd\_serial.cpp file (Figure 2); threads for dd\_threads.cpp file (Figure 3) and openmp for dd\_omp.cpp file (Figure 4). I also added Makefile files into their specific directories.

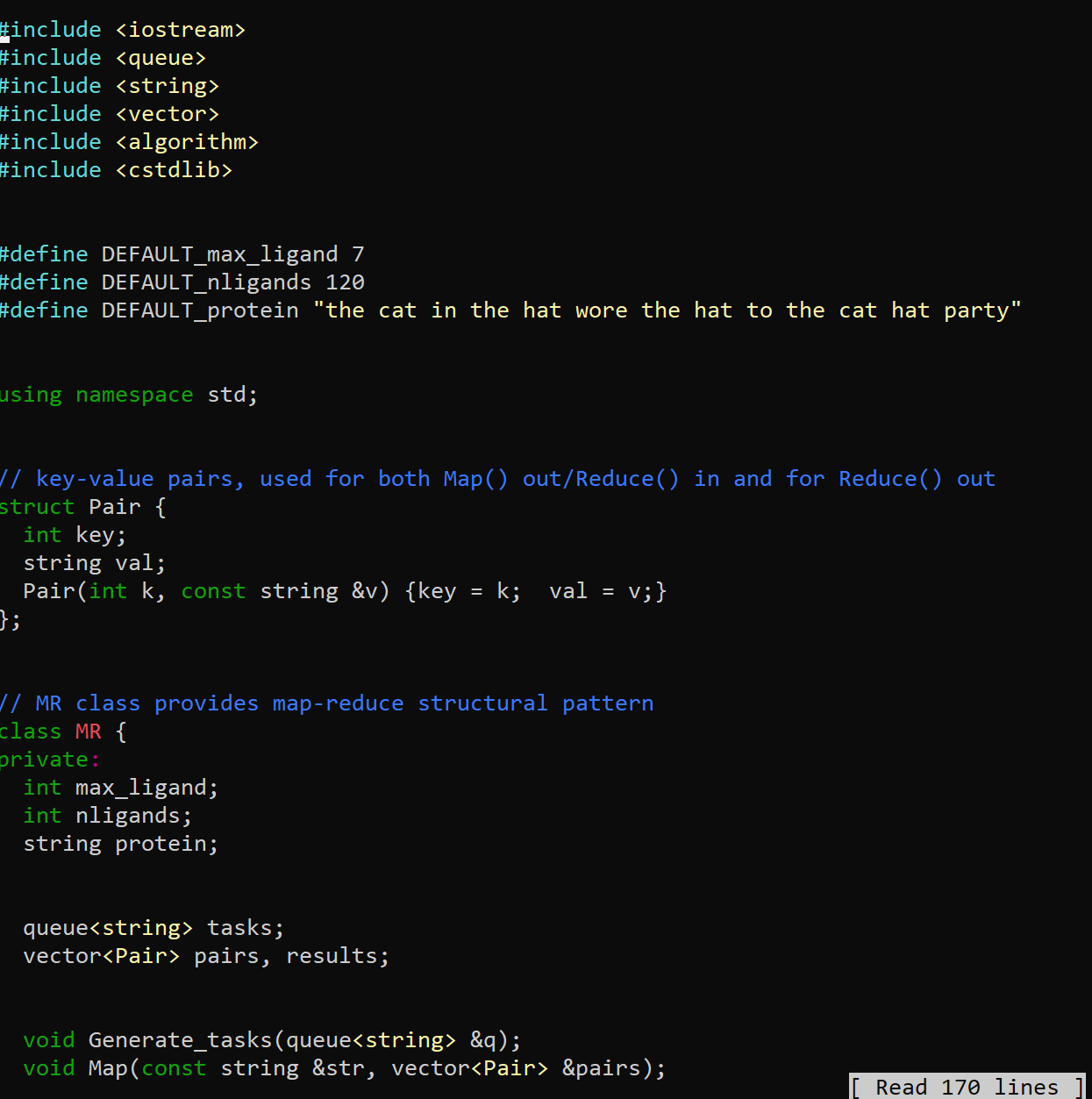


Figure 2 – fragment of dd\_serial.cpp program

Figure 3 – fragment of dd\_threads.cpp program

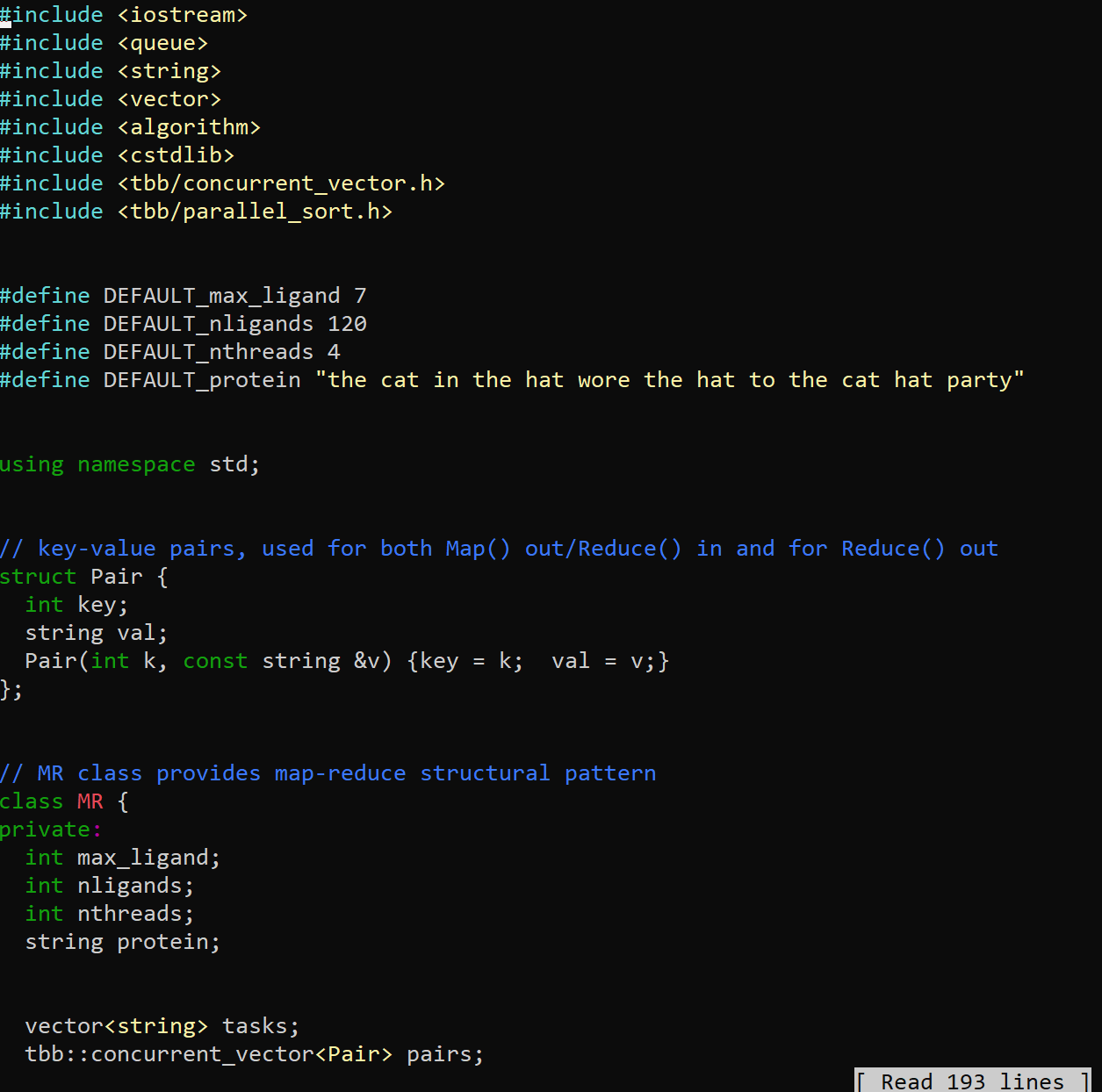


Figure 4 – fragment of dd\_omp.cpp program

Then, I made all three of those files executable files and ran dd\_omp.cpp (Figure 5), dd\_threads.cpp (Figure 6) and dd\_serial.cpp (Figure 7).

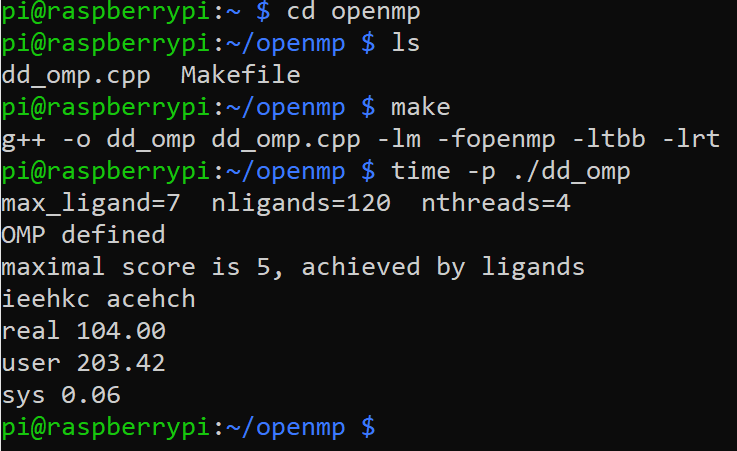


Figure 5 – compiling and running dd\_omp.cpp

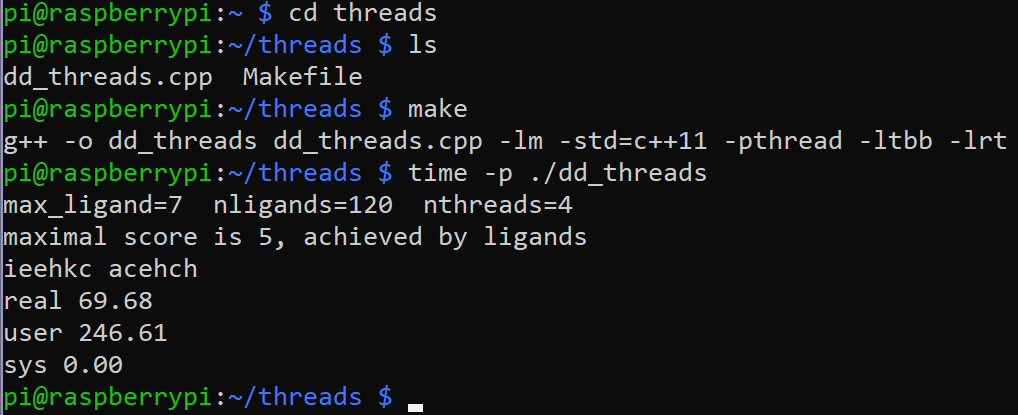


Figure 6 – compiling and running dd\_threads.cpp

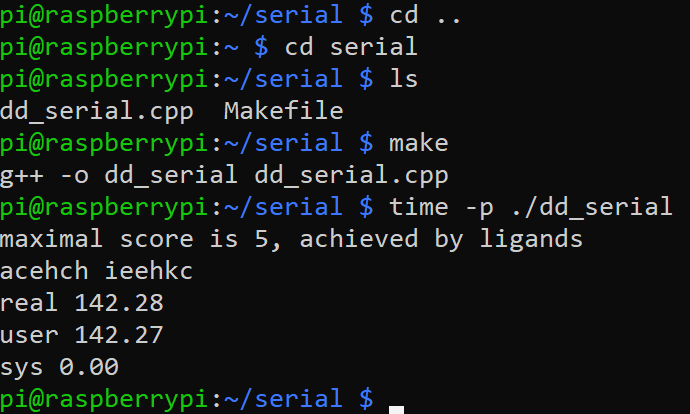


Figure 7 – compiling and running dd\_serial.cpp

Each of these three files calculate run time. By not passing any arguments both OpenMP and Threads have a default number of max\_ligants=7, ligants=120 and nthreads=4. We can see that Serial runtime was the slowest, while Threads was the fastest (Table 1). Then I changed the number of max\_ligant by passing a number as an argument in a command line. Both OpenMP and Threads programs ran at 0.02 at max\_ligand=1 (Figure 8).

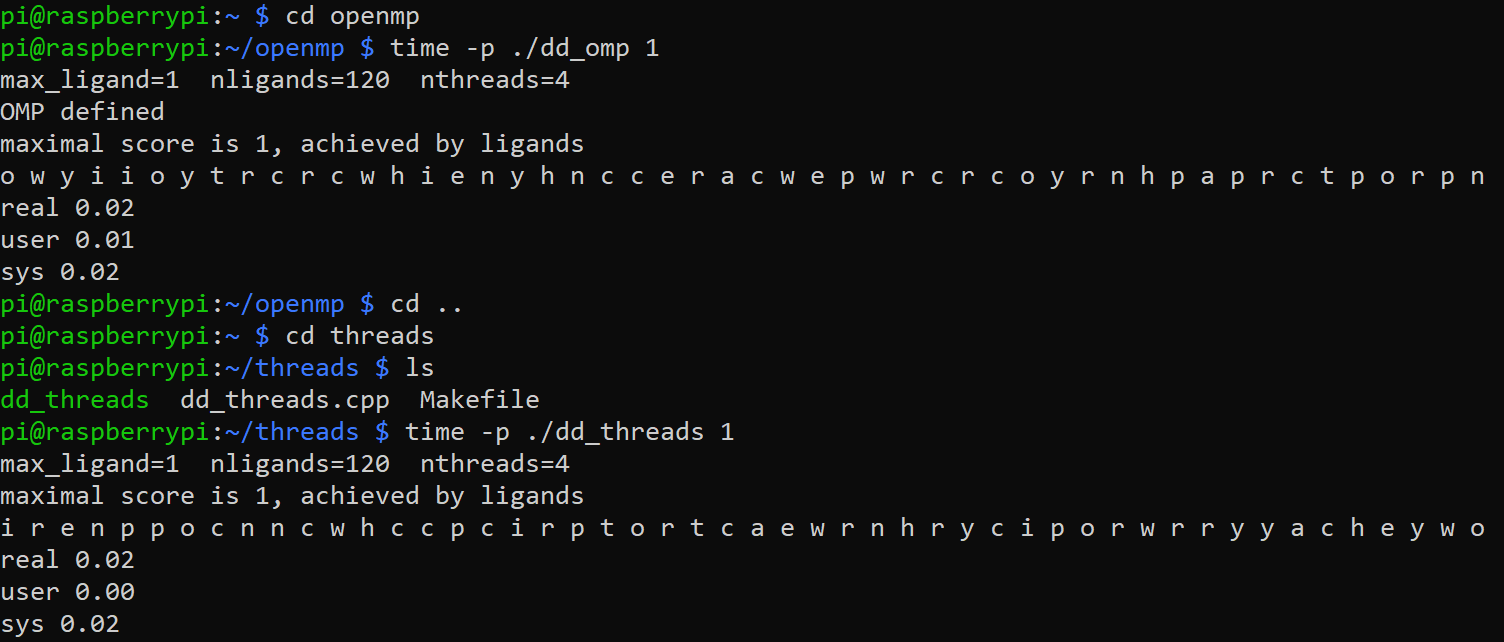


Figure 8 – running dd\_threads and dd\_omp with max\_ligand = 1

Table 1 - run-time of each approach

|  |  |
| --- | --- |
| **Implementation** | **Time (s)** |
| dd\_serial | 142.28 |
| dd\_omp | 104.00 |
| dd\_threads | 69.68 |

Then, I ran dd\_omp program multiple times and measured its runtime (Figure 9). Each time I changed the number of threads to see if it’ll affect its time.

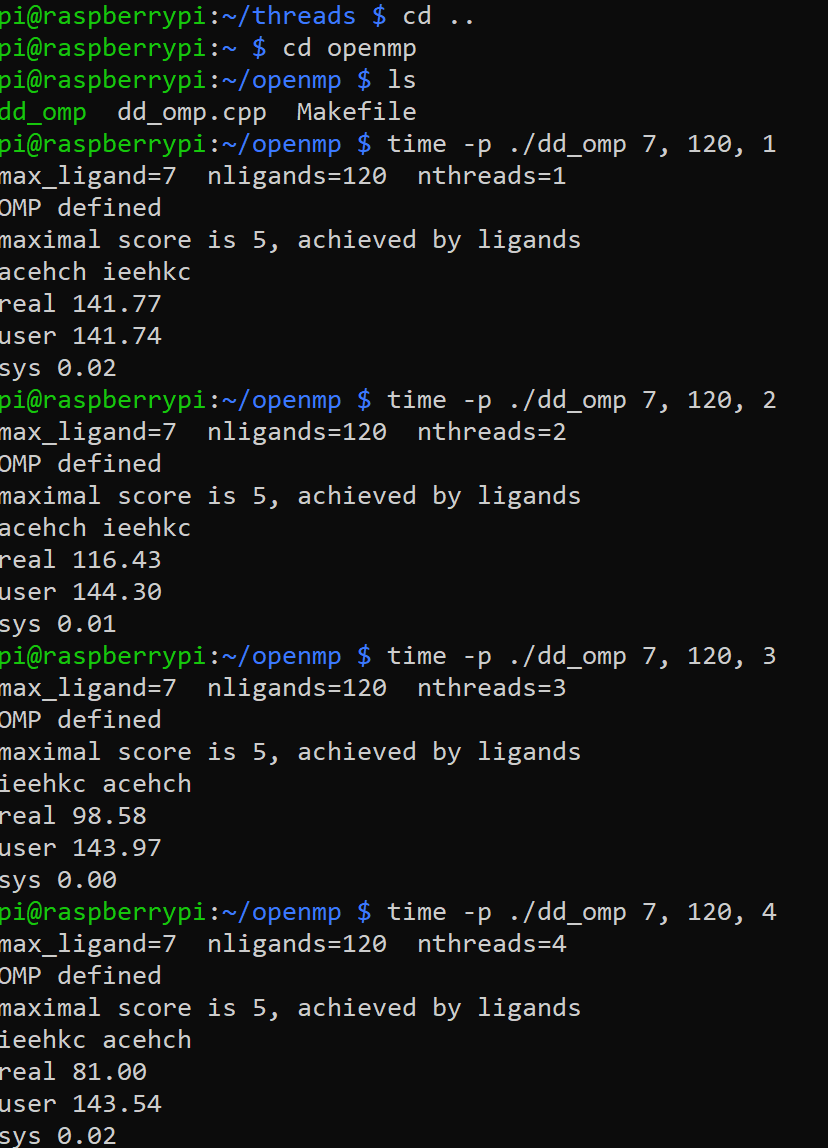


Figure 9 – running dd\_omp with multiple threads

I repeated the same steps for dd\_threads program (Figure 10).

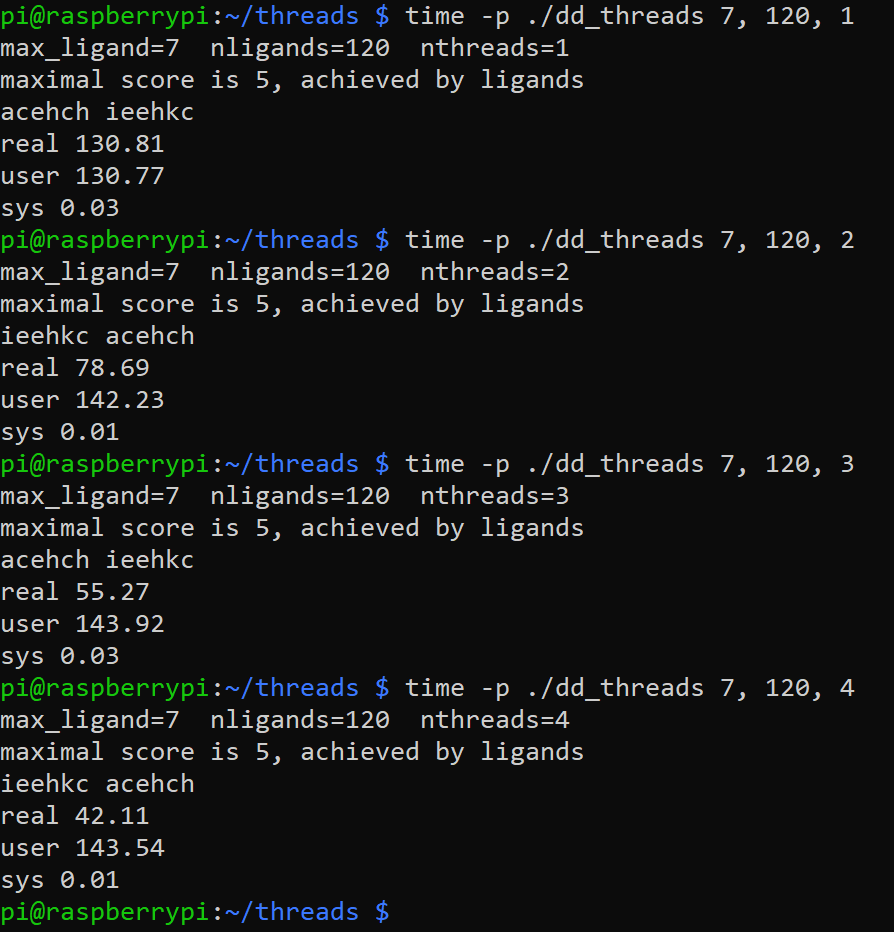


Figure 10 – running dd\_threads with multiple threads

The runtime of the program seemed to be much faster the more threads the user would input. The runtime of program dd\_omp at 4 threads was 81.00 while at 2 threads it was 116.43. The dd\_threads program was overall much faster regardless of the number of threads I input. At 2 threads, dd\_threads runtime was 78.69 while at 4 it was 42.11. I recorded the runtimes for both programs in Table 2.

Table 2 – runtime of dd\_omp and dd\_threads at different threads

|  |  |  |  |
| --- | --- | --- | --- |
| **Implementation** | **Time (s) 2 Threads** | **Time (s) 3 Threads** | **Time (s) 4 Threads** |
| dd\_omp | 116.43 | 98.58 | 81.00 |
| dd\_threads | 78.69 | 55.27 | 42.11 |

Discussion Questions:

1. The fastest approach was using dd\_threads program.
2. Using the wc-l command, we find out that dd\_omp program has 193 lines, dd\_serial has 170 and dd\_threads program has 207 lines (Figure 11).

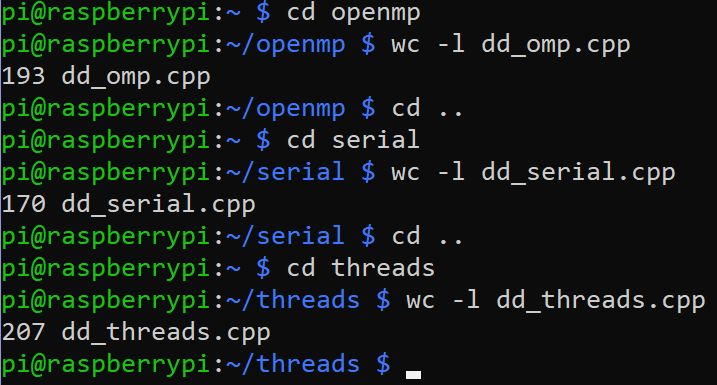


Figure 11 – number of lines of each program

1. I set max\_ligand to 5 and threads number to 5. The runtime of dd\_omp was 0.83, dd\_threads was 0.68 and dd\_serial was 2.43 (Figure 12).

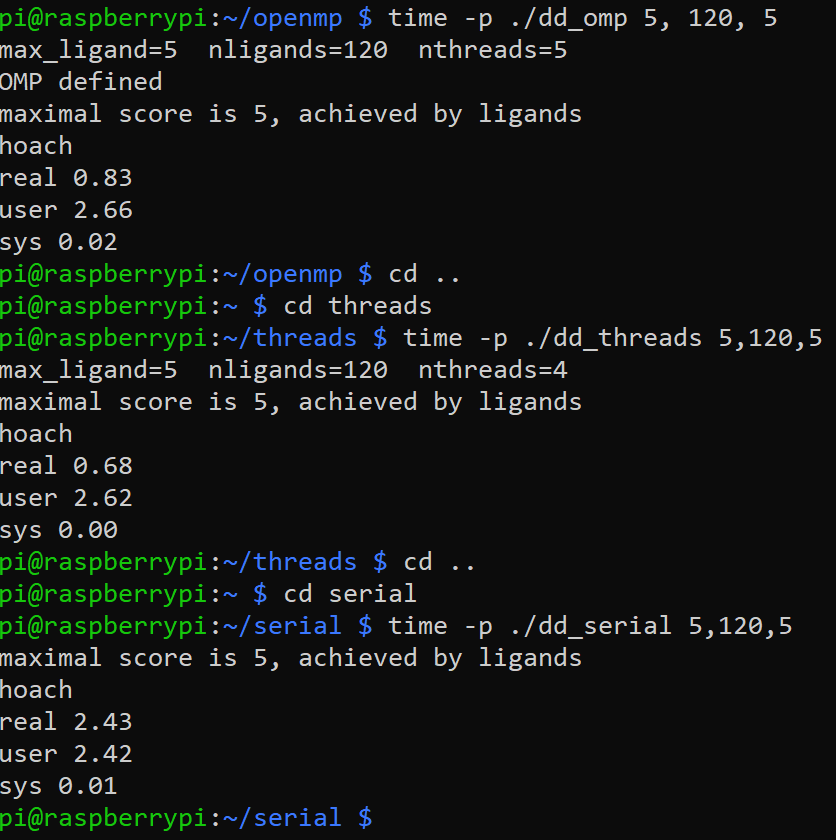


Figure 12 – runtime of each programs at 5 threads

1. I changed the number of max\_ligant=7 and ran each program. The runtime of dd\_omp was 80.69, dd\_threads was 42.07 and dd\_serial was 142.22 (Figure 13).

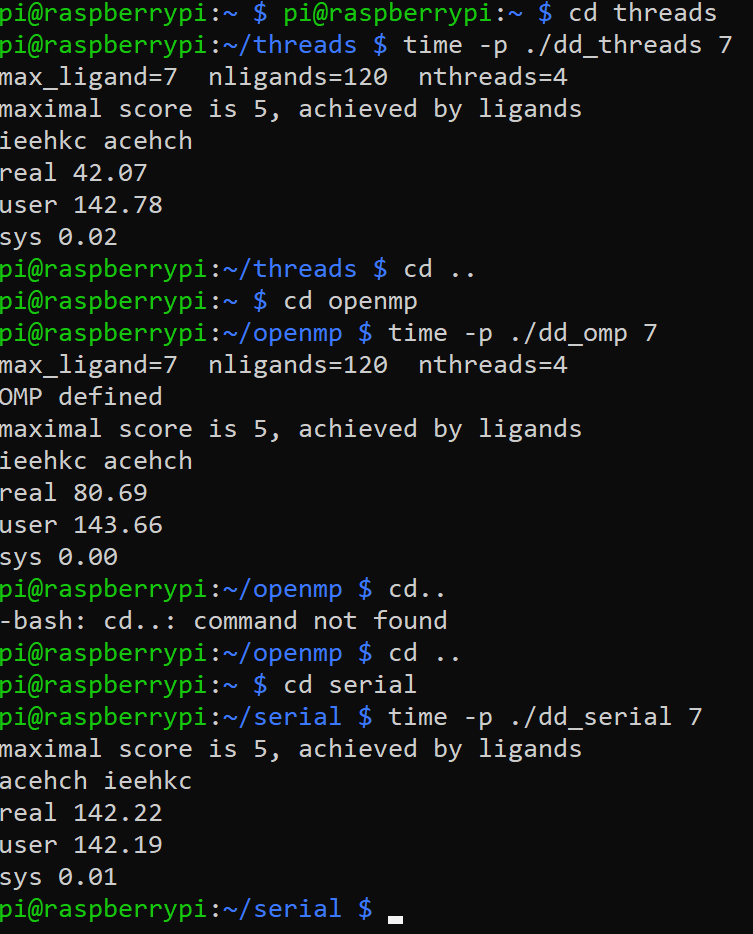


Figure 13 – runtime of each program at max\_ligant=7