

Name: Minh Binh Nguyen
PantherID: 002-46-4288

Project A5 – Task 3 Report

Part A: Foundation

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

These are the basic steps in building a parallel program:

1. Decomposition: Identify the works that can be executed simultaneously, also the partitions of data that can be processed concurrently if needed.
2. Assignment: Specify the mechanism to divide the works and assign them to processes. Balance the workload, also try to reduce the communication and the management cost.
3. Orchestration: Create a model of the program by naming the data, structuring the communication, and organizing them.
4. Mapping: Determine which processes will run on the same processor, which will run on particular processor.

Example:

To determine which tasks can run concurrently, we consider its dependency. For example, we cannot calculate Fibonacci function using parallel technique because each time we compute a value, we have to wait for the previous one to finish.

Another example, we can apply parallel technique to a program which has a big array which can be broken down into sub-arrays if the same processing is required for each element, with no dependencies when computing.

What is MapReduce?

MapReduce is a parallel programming model and an associated implementation to process and generate big data sets using distributed algorithm on a cluster. The users can specify a map function that can process a value pair to generate sets of intermediate value pairs. Users can also specify a reduce function that can merge all intermediate values associated with the same intermediate key.

What is map and what is reduce?

In a functional language like Lisp, a map takes a function and a sequence of values as inputs, then applies that function to each value in the sequence. A reduce uses a binary operation to combine all the elements of a sequence. Example, we use reduce with a plus sign to add up all the elements in the sequence.

Why MapReduce?

When dealing with large amount of raw data, these data must be distributed to a number of machines to be processed quickly. This distribution is applied using parallel computing to perform the same computation on each CPU, but with different datasets. MapReduce allows

the engineers to simplify the computation while hiding the parallelization, data distributions, load balancing and fault tolerance.

Show an example for MapReduce.

Counting the number of appearances of each word in a very large document. By using MapReduce, the map function can count the appearances of each word, then the reduce function will sum the counts up.

Explain in your own words how MapReduce model is executed?

Firstly, the Map invocations are distributed to number of machines using partitioning the input data into a set of M number of splits. These splits can be run in parallel on different machines. Then the Reduce invocations are distributed using the intermediate key space into R number of small pieces using a user-defined partitioning function.

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

- + Distributed Grep: The map function maps a line if it finds a match with a given pattern. The Reduce's job is just an identity function that copy the provided data to the output.
- + Count of URL Access Frequency: The map function processes the logs of web requests and provide the pairs of URLs and the counts (in this case is 1 for each URL). The Reduce function then adds all the values of the same URL and produce the pairs of URLs and the total count.
- + Reverse Web-Link Graph: The Map function produces the pairs of the targets and sources for each link to a target URL that is found in a source page. The Reduce function then concatenates the list of all source URLs and produces the pairs of targets and list of sources.

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

- + OpenMP: This can be used to introduce shared memory parallelism to your code. OpenMP can split the loop into multiple threads, each of them can process a chunk of loop's iterations. Because OpenMP is a very neat and powerful library to do what you might write manual thread code.
- + MPI (Message Passing Interface): This can be used in distributed memory parallel model implementation. It is used to develop parallel scientific applications. MPI can also be used to write parallel code that can be ran on multiple machines, Because the scientific applications are tightly synchronous code and well load balanced. But MPI is not a best choice to implement dataflow styled parallel.
- + Hadoop MapReduce: This can be chosen over MPI when you have a large amount of data that you want to perform an ETL (extract, transform, and load). Because the data is distributed using HDFS (The Hadoop Distributed File System) that is designed to run on commodity hardware. After that, you can do some reduction on the produced results.

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


Our DNA is like a book of recipes that instructs our bodies to make proteins. Medicine is a protein's shape that determines the function it performs in a body. In order to design a drug, we need to find ligands (new pieces) to change those protein's shapes. Therefore, the idea to

write a drug design software is first to generate the ligands to try for particular protein, then to compute a score for each ligand that can show how well it fit in the protein and produce a desired shape, finally to record and to pick the highest scoring ligands and test them.

Part B: Programming

Drug Design and DNA in Parallel

Firstly, I put all the required files in the correct folders (serial, openMP, threads). I also learned after doing some researches that I have to install the library “libtbb-dev” (Threading Building Blocks) in order to make executable files (Figure 1).



```
nbminh — pi@raspberrypi: ~/Project/threads — ssh pi@192.168.28.15 — 80x34
Last login: Tue Apr 14 20:10:06 on ttys001

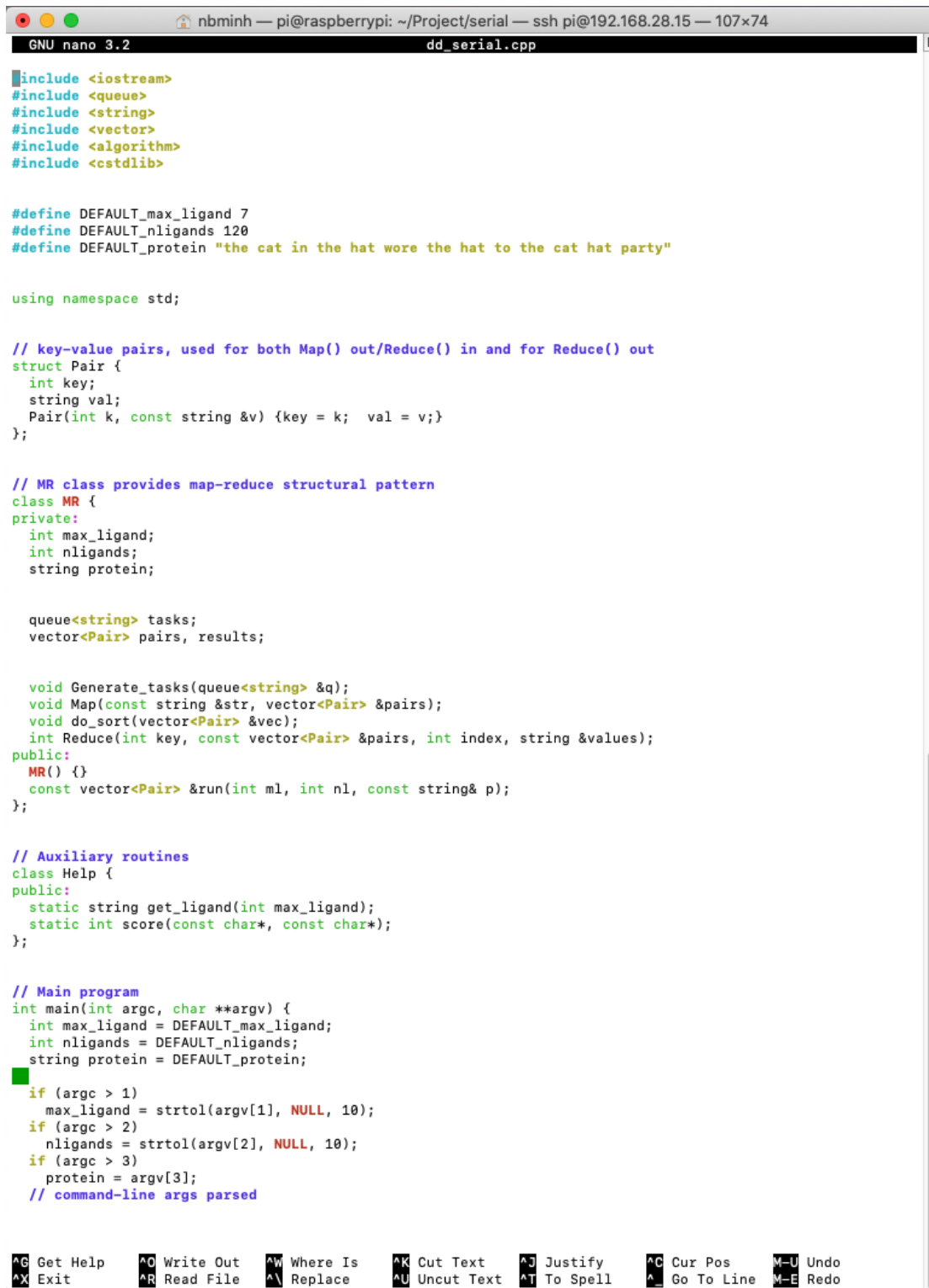
The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
[(base) Minhs-MacBook-Pro:~ nbminh$ ssh pi@192.168.28.15
pi@192.168.28.15's password:
Linux raspberrypi 4.19.50-v7+ #896 SMP Thu Jun 20 16:11:44 BST 2019 armv7l

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Fri Apr 10 22:03:21 2020
[pi@raspberrypi:~ $ sudo apt-get install libtbb-dev
Reading package lists... Done
Building dependency tree
Reading state information... Done
libtbb-dev is already the newest version (2018~U6-4).
0 upgraded, 0 newly installed, 0 to remove and 135 not upgraded.
[pi@raspberrypi:~ $ pwd
/home/pi
[pi@raspberrypi:~ $ cd ~/Project/serial
[pi@raspberrypi:~/Project/serial $ make
g++ -o dd_serial dd_serial.cpp
[pi@raspberrypi:~/Project/serial $ cd ~/Project/openMP
[pi@raspberrypi:~/Project/openMP $ make
g++ -o dd_omp dd_omp.cpp -lm -fopenmp -ltbb -lrt
[pi@raspberrypi:~/Project/openMP $ cd ~/Project/threads
[pi@raspberrypi:~/Project/threads $ make
g++ -o dd_threads dd_threads.cpp -lm -std=c++11 -pthread -ltbb -lrt
[pi@raspberrypi:~/Project/threads $ ]
```

Figure 1

I then opened up the code trying to figure out how things work. Then I learned that the max_ligand, nligands, and nthreads can be passed in as arguments respectively. And their default values are 7, 120, 4 respectively (Figure 2).



```
GNU nano 3.2 dd_serial.cpp

#include <iostream>
#include <queue>
#include <string>
#include <vector>
#include <algorithm>
#include <cstdlib>

#define DEFAULT_max_ligand 7
#define DEFAULT_nligands 120
#define DEFAULT_protein "the cat in the hat wore the hat to the cat hat party"

using namespace std;

// key-value pairs, used for both Map() out/Reduce() in and for Reduce() out
struct Pair {
    int key;
    string val;
    Pair(int k, const string &v) {key = k; val = v;}
};

// MR class provides map-reduce structural pattern
class MR {
private:
    int max_ligand;
    int nligands;
    string protein;

    queue<string> tasks;
    vector<Pair> pairs, results;

    void Generate_tasks(queue<string> &q);
    void Map(const string &str, vector<Pair> &pairs);
    void do_sort(vector<Pair> &vec);
    int Reduce(int key, const vector<Pair> &pairs, int index, string &values);
public:
    MR() {}
    const vector<Pair> &run(int ml, int nl, const string& p);
};

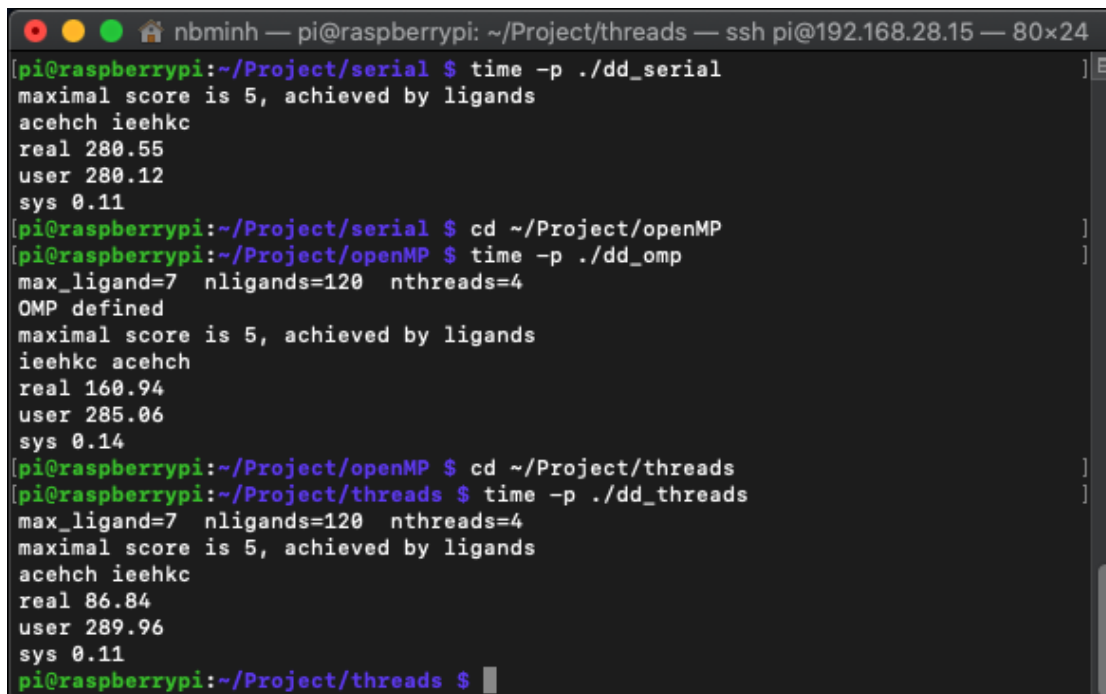
// Auxiliary routines
class Help {
public:
    static string get_ligand(int max_ligand);
    static int score(const char*, const char*);
};

// Main program
int main(int argc, char **argv) {
    int max_ligand = DEFAULT_max_ligand;
    int nligands = DEFAULT_nligands;
    string protein = DEFAULT_protein;
    if (argc > 1)
        max_ligand = strtol(argv[1], NULL, 10);
    if (argc > 2)
        nligands = strtol(argv[2], NULL, 10);
    if (argc > 3)
        protein = argv[3];
    // command-line args parsed
```

^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos M-U Undo
^X Exit ^R Read File ^_ Replace ^U Uncut Text ^T To Spell ^_ Go To Line M-E Redo

Figure 2

Then I went ahead and ran it to measure the run-time with their default values (I didn't pass any arguments in). And here is the result (Figure 3).

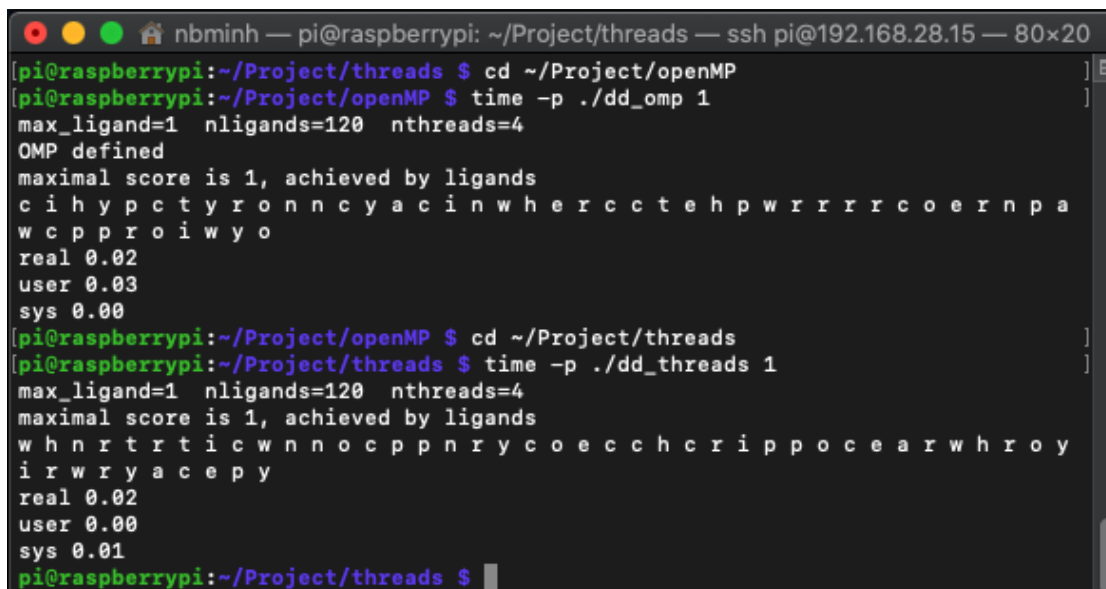


```
nbminh — pi@raspberrypi: ~/Project/threads — ssh pi@192.168.28.15 — 80x24
[pi@raspberrypi:~/Project/serial $ time -p ./dd_serial
maximal score is 5, achieved by ligands
acehch ieehkc
real 280.55
user 280.12
sys 0.11
[pi@raspberrypi:~/Project/serial $ cd ~/Project/openMP
[pi@raspberrypi:~/Project/openMP $ time -p ./dd_omp
max_ligand=7 nligands=120 nthreads=4
OMP defined
maximal score is 5, achieved by ligands
ieehkc acehch
real 160.94
user 285.06
sys 0.14
[pi@raspberrypi:~/Project/openMP $ cd ~/Project/threads
[pi@raspberrypi:~/Project/threads $ time -p ./dd_threads
max_ligand=7 nligands=120 nthreads=4
maximal score is 5, achieved by ligands
acehch ieehkc
real 86.84
user 289.96
sys 0.11
pi@raspberrypi:~/Project/threads $
```

Figure 3

As we can see, the run-time is dramatically decreased. The Serial version took longest (280.55 s), then the OpenMP version took 160.94 s to complete, and the Threads version was the fastest with 86.84 s. Note that I ran this with the default values.

I then ran the instructions as instructed to fill in the table (Figure 4).



```
nbminh — pi@raspberrypi: ~/Project/threads — ssh pi@192.168.28.15 — 80x20
[pi@raspberrypi:~/Project/threads $ cd ~/Project/openMP
[pi@raspberrypi:~/Project/openMP $ time -p ./dd_omp 1
max_ligand=1 nligands=120 nthreads=4
OMP defined
maximal score is 1, achieved by ligands
c i h y p c t y r o n n c y a c i n w h e r c c t e h p w r r r r c o e r n p a
w c p p r o i w y o
real 0.02
user 0.03
sys 0.00
[pi@raspberrypi:~/Project/openMP $ cd ~/Project/threads
[pi@raspberrypi:~/Project/threads $ time -p ./dd_threads 1
max_ligand=1 nligands=120 nthreads=4
maximal score is 1, achieved by ligands
w h n r t r t i c w n n o c p p n r y c o e c c h c r i p p o c e a r w h r o y
i r w r y a c e p y
real 0.02
user 0.00
sys 0.01
pi@raspberrypi:~/Project/threads $
```

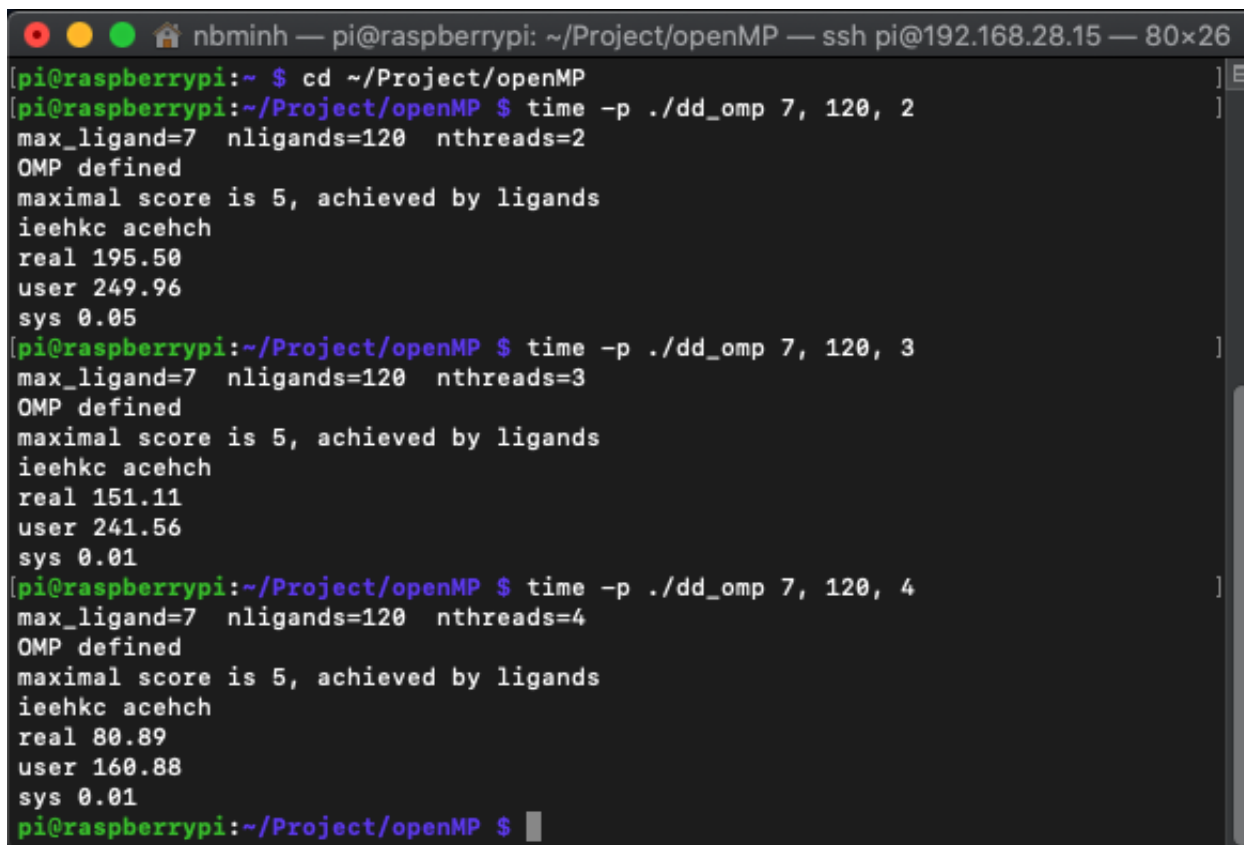
Figure 4

The run-times are recorded into the table below (Table 1).

Implementation	Time (s)
dd_serial (default)	280.55
dd_omp (default)	160.94
dd_threads (default)	86.84
dd_omp 1 (max_ligand = 1 as instructed)	0.02
dd_threads 1 (max_ligand = 1 as instructed)	0.02

Table 1

Then, I ran the OpenMP version and the Threads version with the specs of max_ligand = 7, nligands = 120, and nthreads = {2, 3, 4} for each (Figure 5 and 6).

A terminal window titled 'nbminh — pi@raspberrypi: ~/Project/openMP — ssh pi@192.168.28.15 — 80x26'. The terminal shows three commands being executed: 'time -p ./dd_omp 7, 120, 2', 'time -p ./dd_omp 7, 120, 3', and 'time -p ./dd_omp 7, 120, 4'. Each command outputs the following text: 'max_ligand=7 nligands=120 nthreads=2' (or 3 or 4), 'OMP defined', 'maximal score is 5, achieved by ligands', 'ieehkc acehch', and then timing statistics (real, user, sys). The timing statistics show that as the number of threads increases, the real time decreases significantly (from 195.50s to 151.11s to 80.89s), while user time remains relatively constant (around 249.96s to 160.88s).

```
[pi@raspberrypi:~ $ cd ~/Project/openMP
[pi@raspberrypi:~/Project/openMP $ time -p ./dd_omp 7, 120, 2
max_ligand=7 nligands=120 nthreads=2
OMP defined
maximal score is 5, achieved by ligands
ieehkc acehch
real 195.50
user 249.96
sys 0.05
[pi@raspberrypi:~/Project/openMP $ time -p ./dd_omp 7, 120, 3
max_ligand=7 nligands=120 nthreads=3
OMP defined
maximal score is 5, achieved by ligands
ieehkc acehch
real 151.11
user 241.56
sys 0.01
[pi@raspberrypi:~/Project/openMP $ time -p ./dd_omp 7, 120, 4
max_ligand=7 nligands=120 nthreads=4
OMP defined
maximal score is 5, achieved by ligands
ieehkc acehch
real 80.89
user 160.88
sys 0.01
pi@raspberrypi:~/Project/openMP $
```

Figure 5 – OpenMP version.

```
nbminh — pi@raspberrypi: ~/Project/threads — ssh pi@192.168.28.15 — 80x23
[pi@raspberrypi:~ $ cd ~/Project/threads
[pi@raspberrypi:~/Project/threads $ time -p ./dd_threads 7, 120, 2
max_ligand=7 nligands=120 nthreads=2
maximal score is 5, achieved by ligands
ieehkc acehch
real 150.39
user 277.05
sys 0.07
[pi@raspberrypi:~/Project/threads $ time -p ./dd_threads 7, 120, 3
max_ligand=7 nligands=120 nthreads=3
maximal score is 5, achieved by ligands
acehch ieehkc
real 109.13
user 287.20
sys 0.05
[pi@raspberrypi:~/Project/threads $ time -p ./dd_threads 7, 120, 4
max_ligand=7 nligands=120 nthreads=4
maximal score is 5, achieved by ligands
ieehkc acehch
real 85.90
user 288.62
sys 0.08
pi@raspberrypi:~/Project/threads $
```

Figure 6 – Threads version.

As we can see, the more threads we give the programs, the faster they perform. Overall, the Threads version is faster than the OpenMP version for each case. The run-times are recorded in the table below (Table 2).

Implementation	Time (s) 2 Threads	Time (s) 3 Threads	Time (s) 4 Threads
dd_omp	195.50	151.11	80.89
dd_threads	150.39	109.13	85.90

Table 2

Discussion Questions

1. Which approach is the fastest?

- As we measured, the fastest approach is the Threads version. Another thing to note is that the greater number of threads we give the program, the faster it can perform.

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

- I then ran the instructions “`wc -l`” to measure the line of code for each program. As the result, the Serial version is the shortest one with only 170 lines, the OpenMP version has 193 lines in it, and the Threads version is the longest with 207 lines (Figure 7).

```

nbminh — pi@raspberrypi: ~/Project/threads — ssh pi@192.168.28.15 — 80x10
[pi@raspberrypi:~/Project/threads $ cd ~/Project/serial
[pi@raspberrypi:~/Project/serial $ wc -l dd_serial.cpp
170 dd_serial.cpp
[pi@raspberrypi:~/Project/serial $ cd ~/Project/openMP
[pi@raspberrypi:~/Project/openMP $ wc -l dd_omp.cpp
193 dd_omp.cpp
[pi@raspberrypi:~/Project/openMP $ cd ~/Project/threads
[pi@raspberrypi:~/Project/threads $ wc -l dd_threads.cpp
207 dd_threads.cpp
pi@raspberrypi:~/Project/threads $

```

Figure 7

3. Increase the number of threads to 5 threads. What is the run time for each?
 - I ran the OpenMP version and Threads version again with the specs of max_ligand = 7, nligands = 120, nthreads = 5 for each one (Figure 8).

```

nbminh — pi@raspberrypi: ~/Project/threads — ssh pi@192.168.28.15 — 80x18
[pi@raspberrypi:~/Project/threads $ cd ~/Project/openMP
[pi@raspberrypi:~/Project/openMP $ time -p ./dd_omp 7, 120, 5
max_ligand=7 nligands=120 nthreads=5
OMP defined
maximal score is 5, achieved by ligands
ieehkc acehch
real 79.67
user 175.35
sys 0.04
[pi@raspberrypi:~/Project/openMP $ cd ~/Project/threads
[pi@raspberrypi:~/Project/threads $ time -p ./dd_threads 7, 120, 5
max_ligand=7 nligands=120 nthreads=5
maximal score is 5, achieved by ligands
acehch ieehkc
real 53.74
user 182.18
sys 0.11
pi@raspberrypi:~/Project/threads $

```

Figure 8

The run-times are recorded in the table below (Table 3).

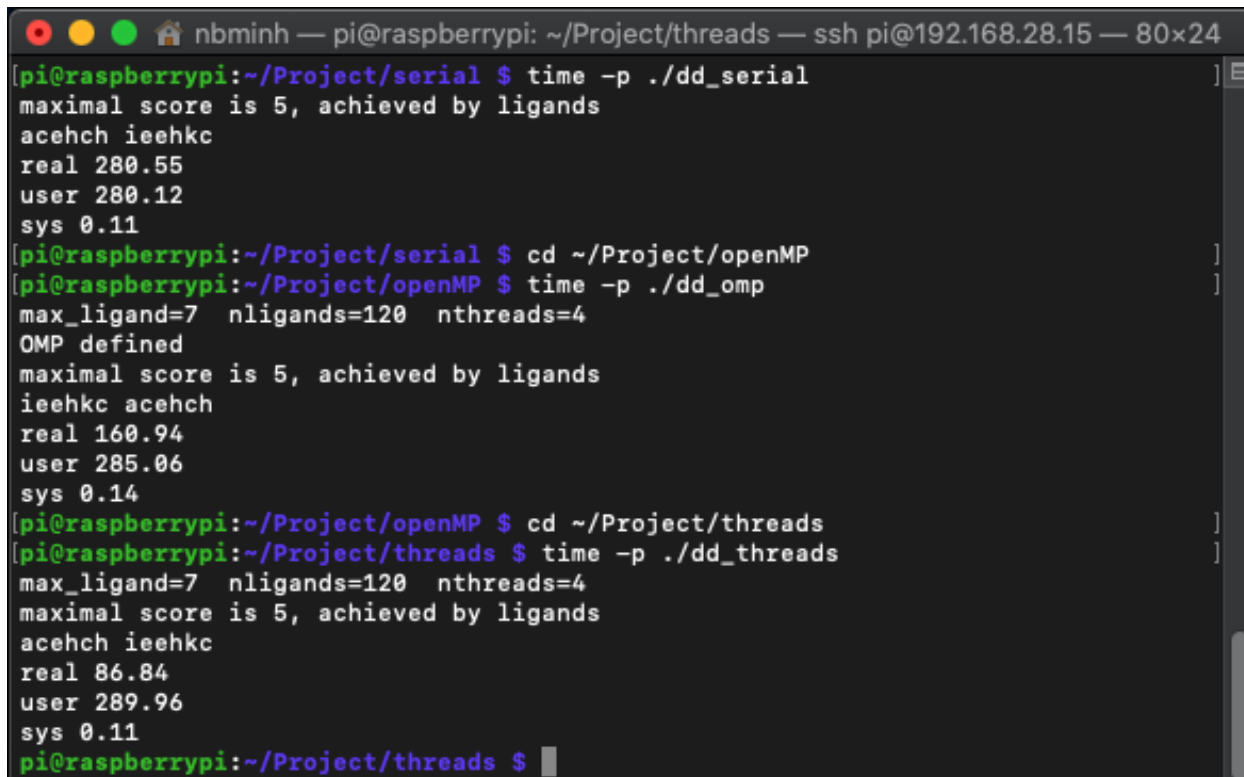
Implementations	Time (s) 5 Threads
dd_omp	79.67
dd_threads	53.74

Table 3

We can see that the Threads version is still faster than the OpenMP version. And by giving them 5 threads, they ran faster than when running them with 2, 3, or 4 threads.

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

- I have already ran each program with this specs at the beginning because the max_ligand by default is 7 (Figure 9).

A terminal window on a Raspberry Pi showing the execution of three programs: dd_serial, dd_omp, and dd_threads. The terminal output for each program shows the maximal score (5), the achieved ligands (acehch ieehkc), and the execution time (real, user, sys). The dd_serial program takes 280.55 seconds. The dd_omp program takes 160.94 seconds. The dd_threads program takes 86.84 seconds.

```
nbminh — pi@raspberrypi: ~/Project/threads — ssh pi@192.168.28.15 — 80x24
[pi@raspberrypi:~/Project/serial $ time -p ./dd_serial
maximal score is 5, achieved by ligands
acehch ieehkc
real 280.55
user 280.12
sys 0.11
[pi@raspberrypi:~/Project/serial $ cd ~/Project/openMP
[pi@raspberrypi:~/Project/openMP $ time -p ./dd_omp
max_ligand=7 nligands=120 nthreads=4
OMP defined
maximal score is 5, achieved by ligands
ieehkc acehch
real 160.94
user 285.06
sys 0.14
[pi@raspberrypi:~/Project/openMP $ cd ~/Project/threads
[pi@raspberrypi:~/Project/threads $ time -p ./dd_threads
max_ligand=7 nligands=120 nthreads=4
maximal score is 5, achieved by ligands
acehch ieehkc
real 86.84
user 289.96
sys 0.11
pi@raspberrypi:~/Project/threads $
```

Figure 9

The run-times are recorded in the table below (Table 4).

Implementation	Time (s)
dd_serial (max_ligand = 7)	280.55
dd_omp (max_ligand = 7)	160.94
dd_threads (max_ligand = 7)	86.84

Table 4