**Developing Soft and Parallel Programming Skills Using Project-Based Learning**

**Fall 2019**

**Team-Raspberry**

**Project A5**

**Hao-Yun Lo, Jomo Morris, Christian Moronta, Amber Pradhan**

**Planning and Scheduling:**

We started the project on December 1st, and delegated tasks to each group member would be responsible for. We created a chart to ensure the team knew their responsibilities.

Hao-Yun Lo finished planning and scheduling in the first time meeting, and he is also the coordinator of this project. Jomo updates Github status for us to track what is our next task, and Hao-Yun contacted other group member through Slack to coordinate the meeting.

Our next meeting was on December 2nd , and this time we tried to complete parallel programming skills. We separate the questions for every group members, then tried to find answers in the document on iCollege. Hao sum up the answers for us, and put it in a doc document.

We also have a short meeting on December 3rd just to make sure everyone was on track, and double check that every group member knows their parts.

Parallel programming and Assembly code was done in our November 12th meeting. Christian type the code, and taught us how the code works.

We filmed the short video on December 4th, we discussed what we learn and what can the thing we learned apply on our next project or future job.

Amber finished the report before December 5th, and we check the final report before we submit the project report.

**Team Raspberry’s Planning and Scheduling**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Assignee**  **Name** | **Email** | **Task** | **Duration** | **Dependency** | **Due**  **Date** |
| Christian Moronta | cmoronta1@student.gsu.edu | Arm & Parallel  Programming | 2 hours | None | 12/03/2019 |
| Hao-Yun Lo | hlo3@student.gsu.edu | Foundation of parallel  Programming -  Answering section | 2 hours | None | 12/04/2019 |
| Jomo Morris | jmorris82@student.gsu.edu | Github/Slack  Video | 2 hours | None | 12/03/2019 |
| Amber Pradhan | amberpradhan12@gmail.com | Github  Reports | 2 hours | None | 12/04/2019 |

**Task Parallel Programming Skills:**

1. **What are the basic steps(show all steps) in building a parallel program?**

Decomposition, Assignment, Orchestration, and Mapping. The example will be scientific computing that uses parallel processing.

1. **What is MapReduce?**

MapReduce is a programming model that help users to process a huge amount of data. The algorithm works on two tasks, which are *map* and *reduce.*

1. **What is map and what is reduce?**

Map is used the set of data, and also converts the data into different set of data. Reduce is used for combining the data from Map, and turns into smaller set.

1. **Why MapReduce?**

MapReduce is important because parallel computing systems let programmers use MapReduce to run models over large sets of data and use machine learning techniques to make predictions, or find patterns.

1. **Show an example for MapReduce.**

Hadoop Mapreduce is an example of MapReduce. The software is for distributed processing of large data.

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

First, break the input file into M pieces, then the cluster of machine will have several copies. One of the copies will become master, and the rest are workers. The master will assign jobs to workers, either map or reduce job. The workers that receive map task will start to read the content of broken pieces of file, then it will pairs the input data and each pair will send to the user-defined map function, then the thing goes to memory. The location of these paris will send back to master, and master will notify reduce workers. When reduce workers are notified, they remotely read the buffered data in the local disk. Reduce workers will start to sort the data after they read all the data, and for each unique intermediate key, workers send to user-defined reduce function. When all the files complete, the master will wake up the user program.

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

a. *Distributed Grep* – is a powerful map function that takes a word (or pattern) and a file or set of files and returns the line in which each instance of that word or pattern appears in the files.

b. *Count of URL Access Frequency* – is a map function that processes logs of web page requests and returns the <URL, 1>. It also sums all values for the same URL and returns a <URL, total count> pair.

c. *Reverse Web-Link Graph* — This map function outputs <target, source> pairs for each link to a target URL found in a page named “source”. The reduce function concatenates the list of all source URLs associated with a given target URL and emits the pair: <target list(source)>.

1. **When do we use OpenMP, MPI and MapReduce, Hadoop, and why?**

OpenMP: is useful for introducing shared memory parallelism to code. OpenMP is an economical directive-based library that allows the user to split for loop into multiple threads, to handle each of the loop’s iterations. This is extremely helpful for programmers with regards to efficiency as they would otherwise have to manually write each thread.

MPI: Message Passing Interface is a distributed memory parallel model implementation, used mostly to develop parallel scientific applications. It works well for scientific modelling because the code is tightly synchronous and well load balanced. You can also combine MPI with OpenMP to use threads within machines. This technique is called hybrid programing.

Hadoop MapReduce: uses two constructs that is applied over massive data quantities which you can now map and reduce. The idea is data is distributed, typically using HDFS, and you want to apply some operation for each data element. Then you'll do some reduction over the results. While this pattern could appear straightforward it's possible to chain maps and reduce design complex problems. Hadoop is better suited than MPI for applications where we need to extract, transform and load terabytes of data. Hadoop has a much better fault tolerance than MPI. So in this context OpenMP is best suited for threading and creating shared memory parallelism, MPI is preferred for parallel scientific applications and Hadoop works best for big data applications that need to be mapped and reduced.

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

Drug design is a problem that biology and the computer sciences have come together to find solutions to. The aim is to find small molecules called ligands that are good candidates for use as drugs. The challenge is in identifying ligands that bind with the protein structures of a target disease. The best way to do this is to

create a computational molecular model to rate the compatibility of ligands being tested against the protein structure of the disease of interest. In the simplest sense, drug design involves the design of molecules that are complementary in shape and charge to the biomolecular target with which they interact and therefore will bind to it. Drug style are aided by laptop modeling techniques. This type of modeling is sometimes referred to as computer-aided drug design.

There are several approaches that can be used to create these simulations such as a map reduce strategy which can be implemented by using a master worker design pattern. A sequential solution as well as OpenMP are other approaches that are used to help compare millions of ligands to target disease DNA for possible matches.

Ideally, the computational method will be able to predict affinity before a compound is synthesized which saves enormous time and cost. Present computational methods are still being perfected and at best qualitatively accurately. It still takes several iterations of design, synthesis, and testing before an optimal drug is discovered. Computational methods have accelerated discovery by reducing the number of iterations required and have often provided novel structures.

**Parallel Programming:**

The parallel programming part of this assignment consisted of three different files. The task was to test the score of ligands to a protein and output which ligands had the highest score. In this case the ligands and protein were represented by strings and the score was how many characters matched from the ligand string to the protein string.

The three programs that were involved were: a program that computed the scores/output in serial, another program that computed the scores/output using openMP, and another that computed the scores/output using C++ threads. The implementation of each of them was similar with a little bit of variance. Each program took command line arguments of max\_ligand (max length of ligand string), nligands (the number of ligand strings to test), and protein (string that represents protein to match with). Both the omp and threads version of the program contained an extra command line argument called nthreads (the number of threads to use). The program consisted of: a struct called Pairs that held an int key (score of ligand) and a string val (the ligand string), a class called MR that provided the map-reduce structural pattern (I will go into more detail of this class later), a class called Help that provides helper functions to the MR class, and the main function which calls all the previously stated code to get the output of ligands with the highest score by getting the first item in a sorted result vector.

The main part of the program that actually produces the scores of the ligands is the function run, located in the MR class. This order of this function in the serial program goes as follows:

1. Inputs for the function are max\_ligand number, nligands, and the protein string.
2. Once that input is saved into variables the function calls the Generate\_Tasks function that is in the same MR class.
   1. This function takes in a queue (tasks, located in the same MR class as a property).
   2. Iterates through the amount of ligands (nligands) and uses a helper function in the Help class called get\_ligand(max\_ligand) which creates a randomly generated ligand string taking as input the max ligand size.
3. A while loop then goes through the queue as long as it’s not empty
   1. Inside the loop the function Map is called with the inputs being the front of the queue and a vector called pairs located in the MR class as a property. This vector holds Pair structs.
      1. Map function creates a score for the ligand string by calling the Help class function score.
      2. creates a Pair with key = score and ligand = the ligand that was passed into the function. Pushes this pair into the back of the pairs vector.
   2. Then dequeue what’s in front of the task queue.
4. Sort the list of pairs by calling do\_sort(tasks). Sorting is in descending order based on the key (score) of the ligand.
5. Iterate through the list of pairs
   1. Call Reduce function in MR class with inputs next (iterator), values (empty string), key (key/score of the pair at next), list of pairs
      1. reduce will go through each item in the list until either goes through full list or the key of current pair isn’t the same as the key passed into the function.
      2. essentially it just takes all the pairs with the same score and edits the values string to be a string with all the ligands that had the same score separated by a space.
      3. returns the index that it stopped at.
   2. create a new pair with inputs score (the key) and the string of all ligands with that same score.
   3. add this new pair to the back of results vector (vector in MR class)
6. Return the result vector.

The difference between this serial implementation and the openMP implementation is that the map part is changed to parallel by turning the while loop into a for loop. Because of this change we need to change the pairs queue into a concurrent vector because it may cause data race if left as as a normal queue when passed into Map function. Also the do\_sort function changes from sort to parallel\_sort in the tbb library.

The difference between the openMP version and the threads version is that in the map

stage, we create an array pool of threads to perform the calls to Map(), then wait for those threads to complete their work by calling the join() method for each thread. Since multiple threads may access the shared task queue at the same time, that task queue must be thread safe, so we define it as a TBB container.

The runtimes of each implementation was measured in two ways. One of which was measuring all three programs on one thread. We see here that they are all similar, yet the threads version takes a little longer with only one thread. The fastest was the openMP version.

|  |  |
| --- | --- |
| **Implementation (1 Thread)** | **Time (s)** |
| dd\_serial | 15.69 |
| dd\_omp | 15.66 |
| dd\_threads | 17.19 |

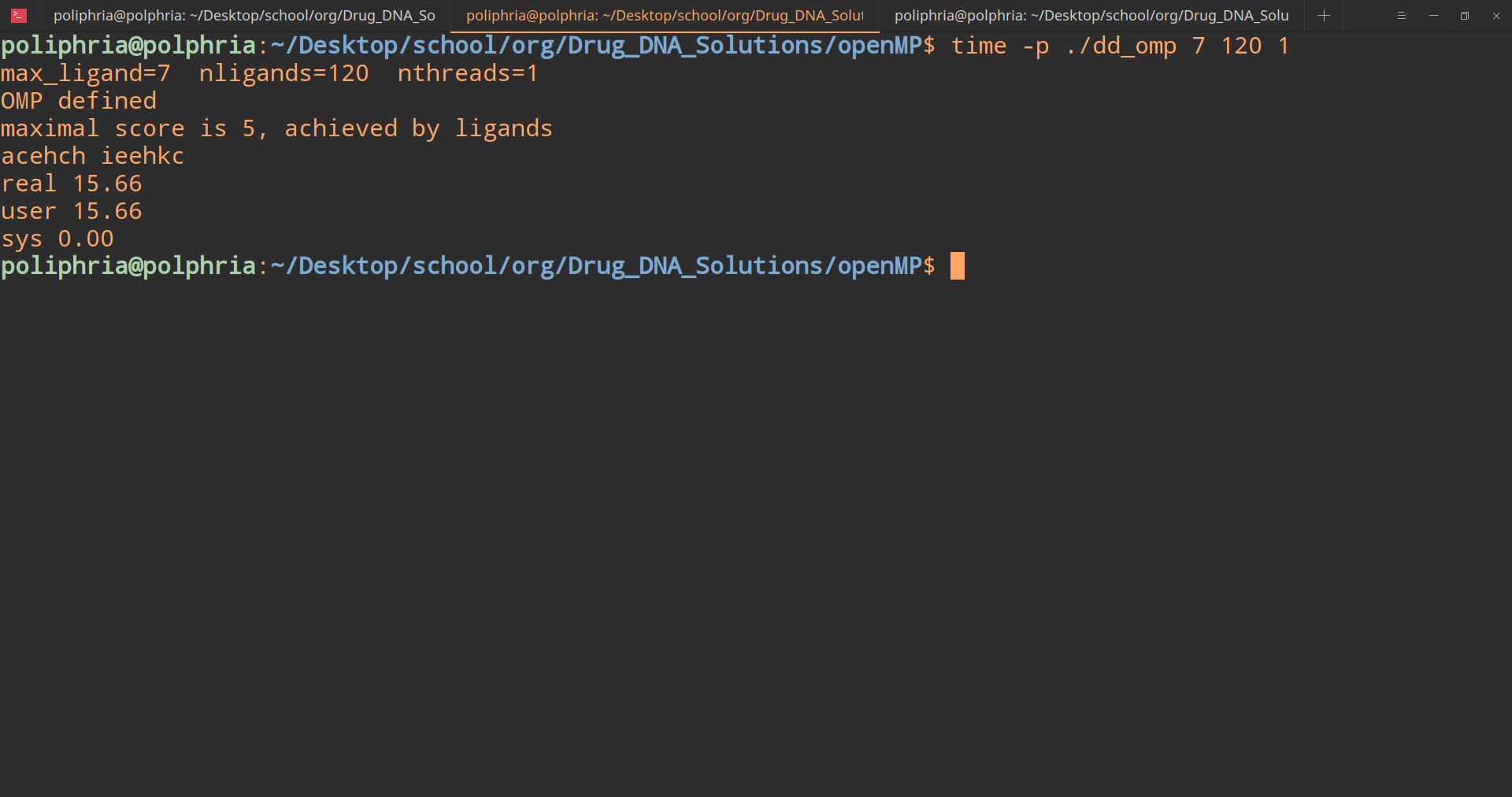
We also measured the runtime of both of the parallel programs using different thread numbers. As you can see below, the fastest was the threads version of the program in every iteration no matter the number of threads.

|  |  |  |  |
| --- | --- | --- | --- |
| **Implementation** | **Time (s) 2 threads** | **Time (s) 3 Threads** | **Time (s) 4 Threads** |
| dd\_omp | 12.62 | 10.67 | 9.87 |
| dd\_threads | 9.18 | 7.03 | 5.73 |

**Parallel Programming Discussion Questions:**

1. Which approach is the fastest?
   1. Based on the observations recorded in the tables above, the fastest with one thread was the openMP version. The fastest with more than one thread was the threads version of the program
2. Determine the number of lines in each file (wc -l). How does the C++11 implementation compare to the openMP implementations
   1. WC serial: 170
   2. WC openMP: 193
   3. WC threads: 207
   4. The threads version only has a little bit more lines of code, but substantially beats the openMP version in terms of speed.
3. Increase the number of threads to 5 threads. What is the run time for each?
   1. openMP run time: 8.28s
   2. Threads run time: 4.76
4. Increase the maximum ligand length to 7, and rerun each program. What is the run time for each?
   1. openMP run time: 8.28s
   2. Threads run time: 4.76

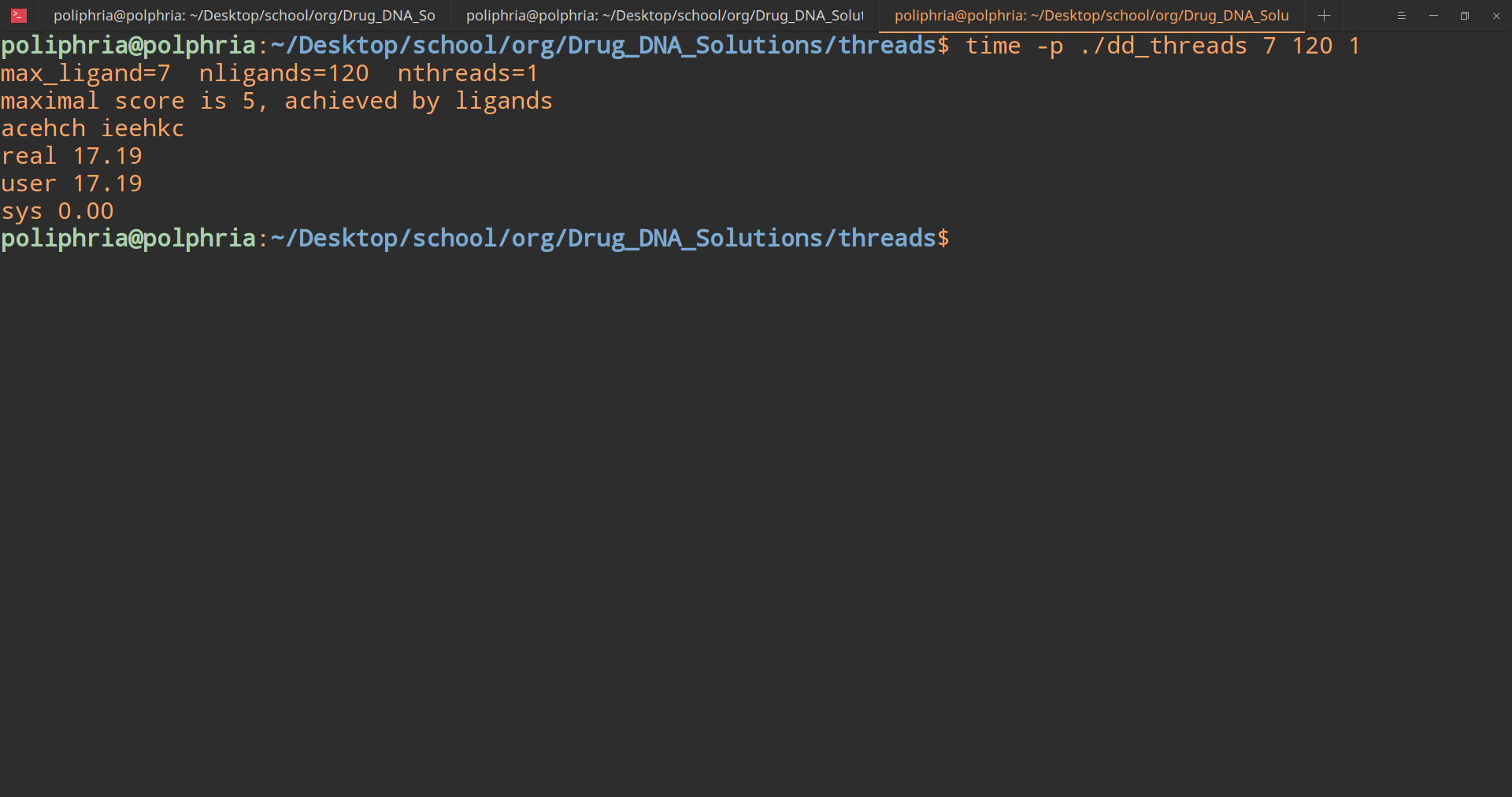
**Appendix:**

****

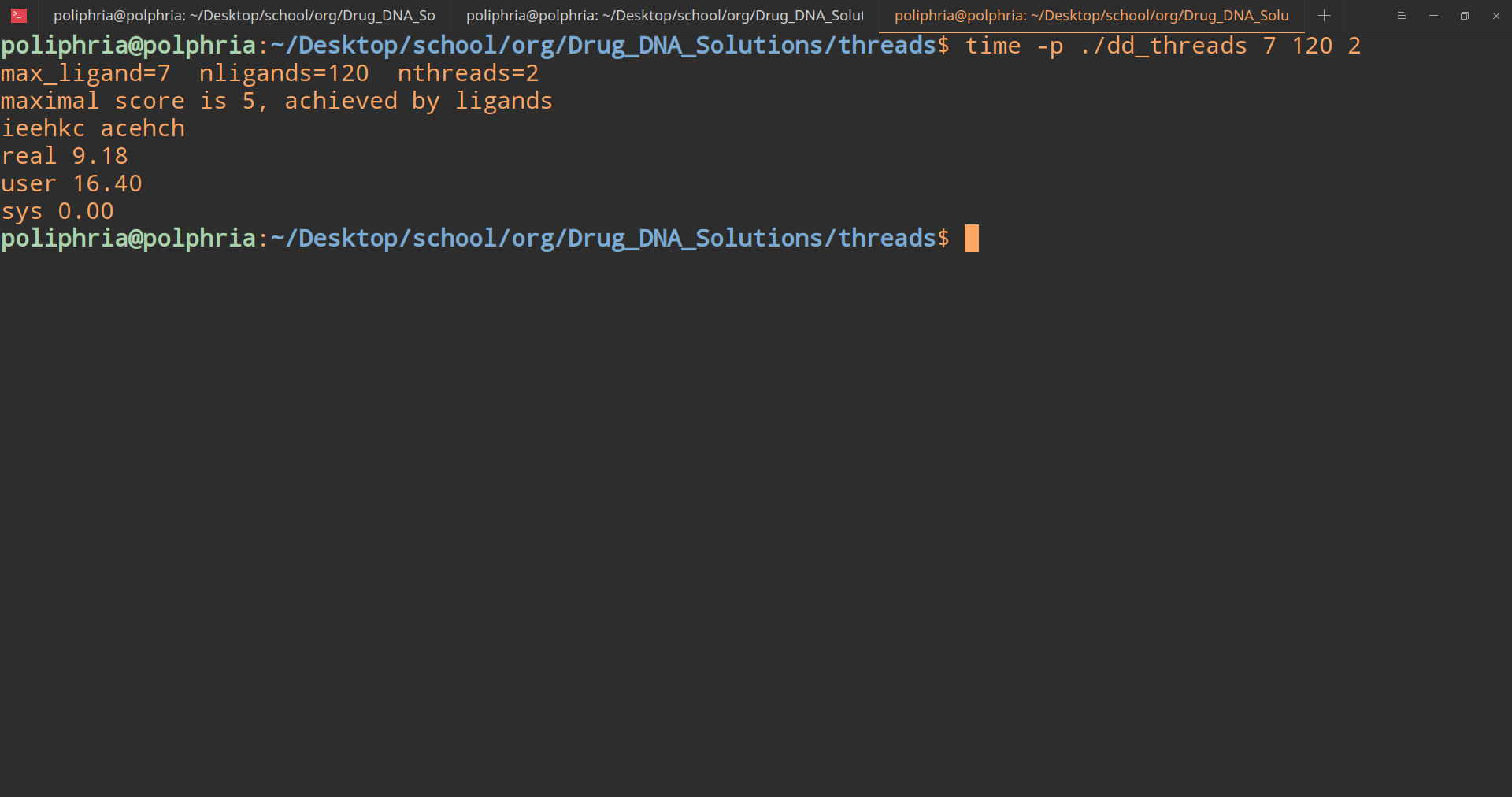
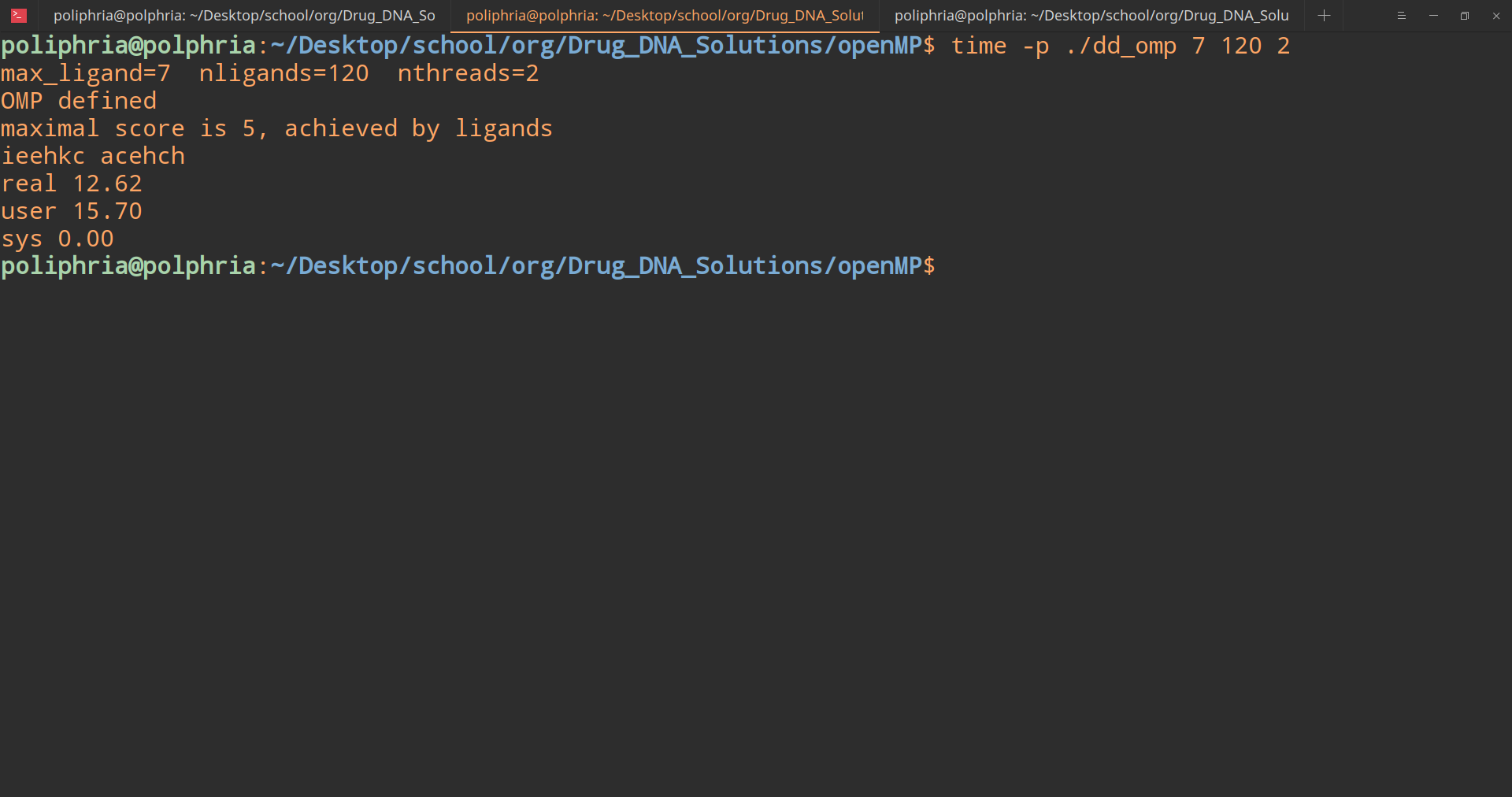
**Time for OpenMP program with one thread**

****

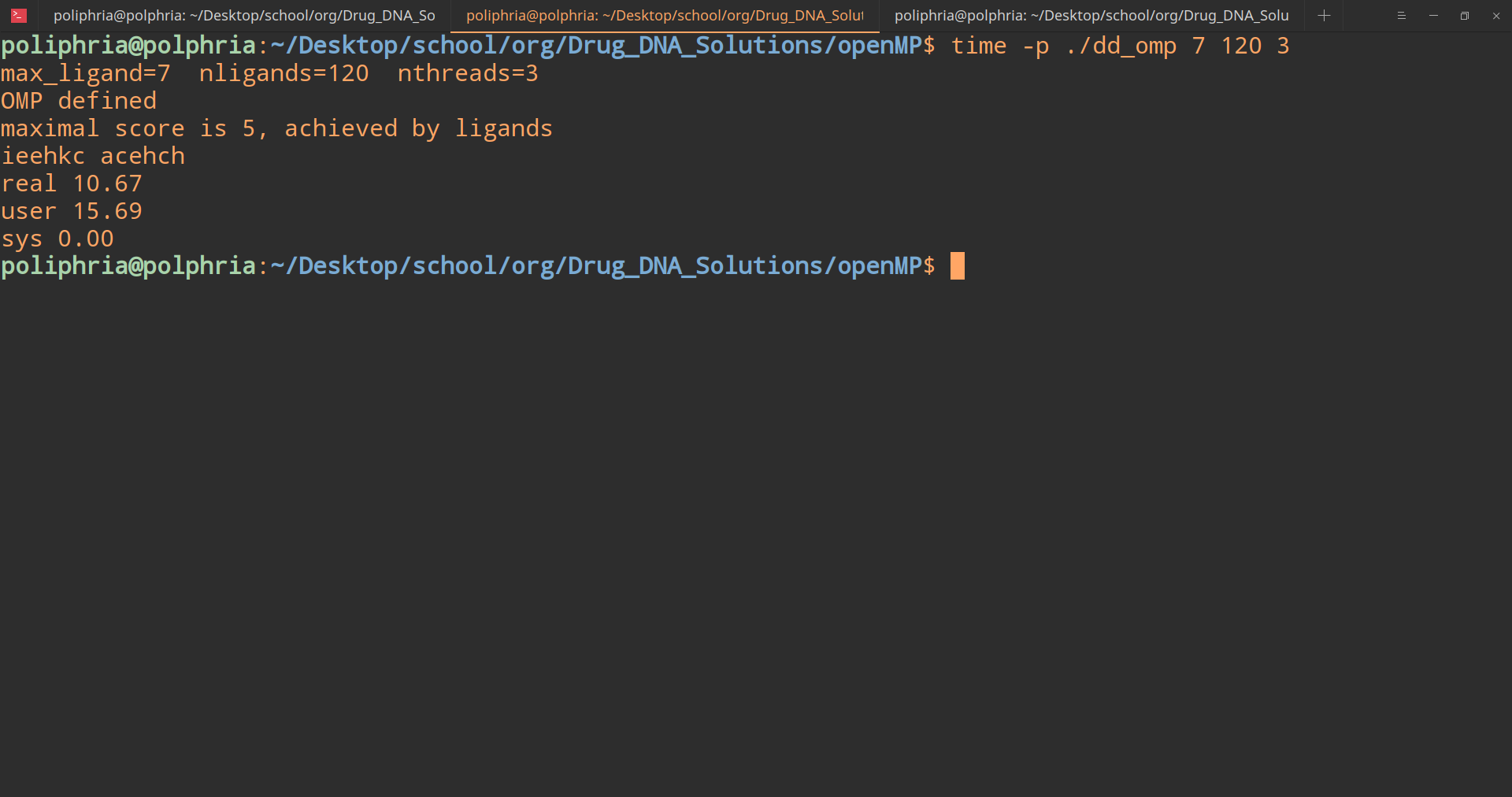
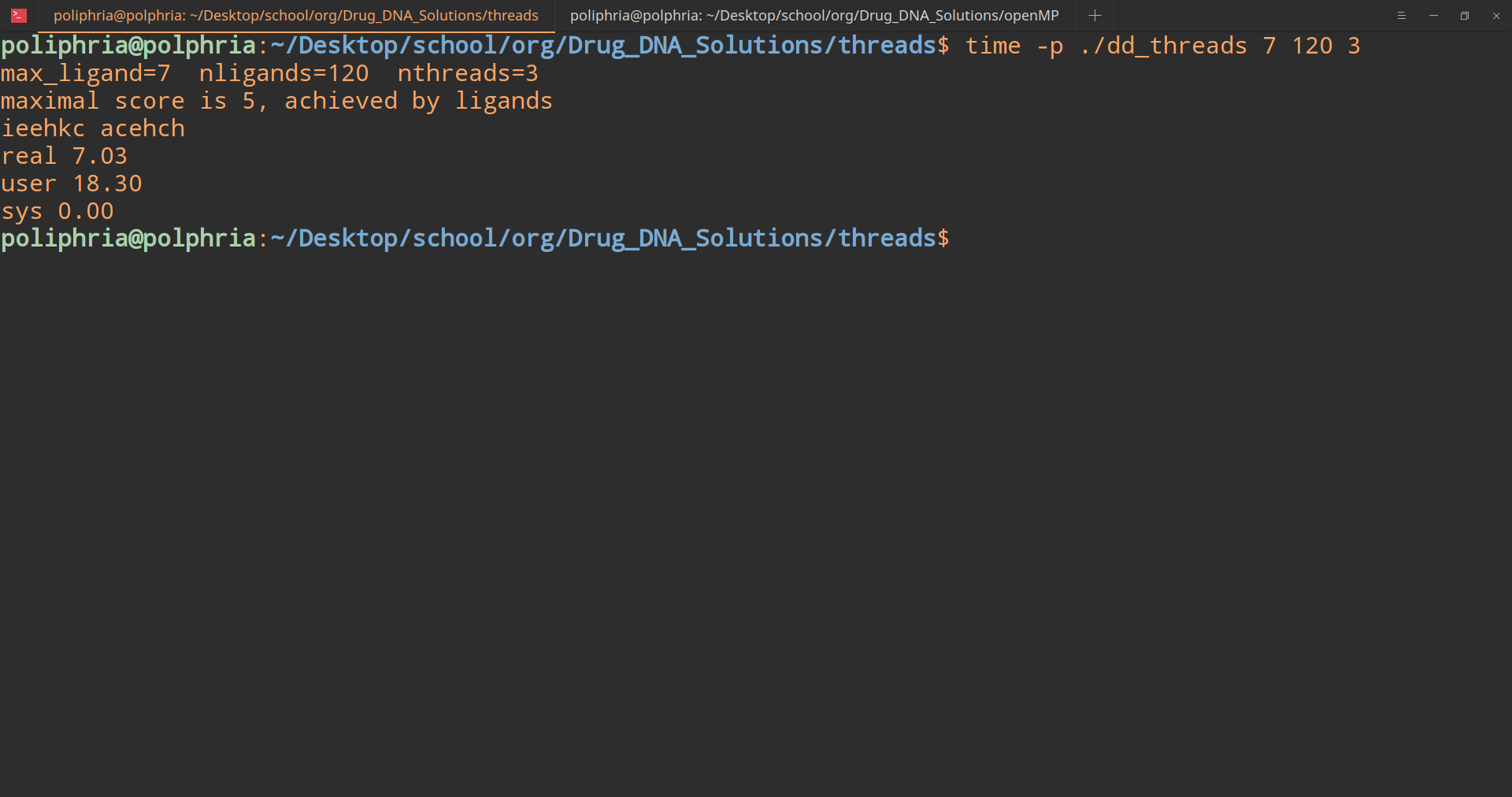
**Time for serial program**

****

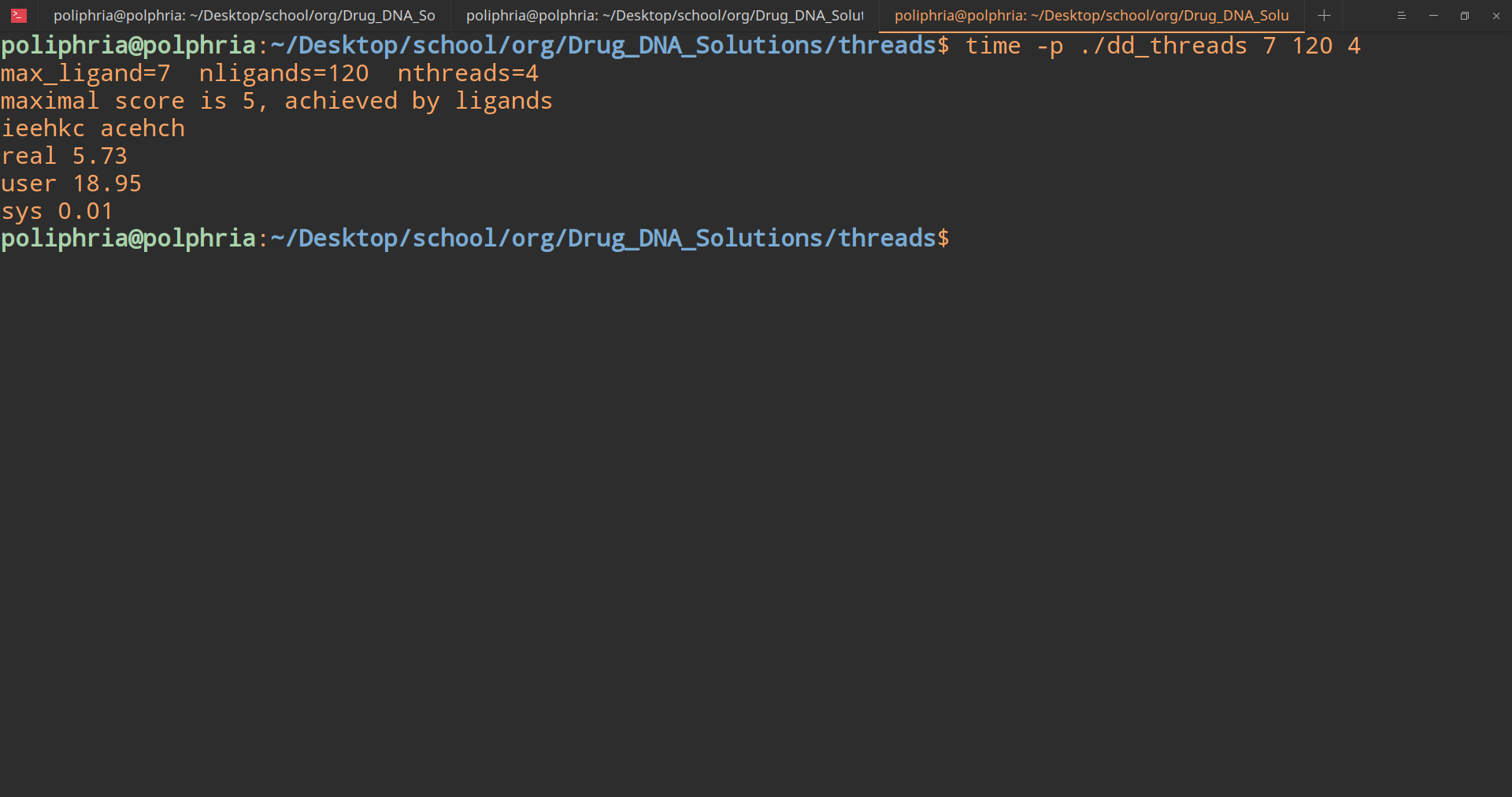
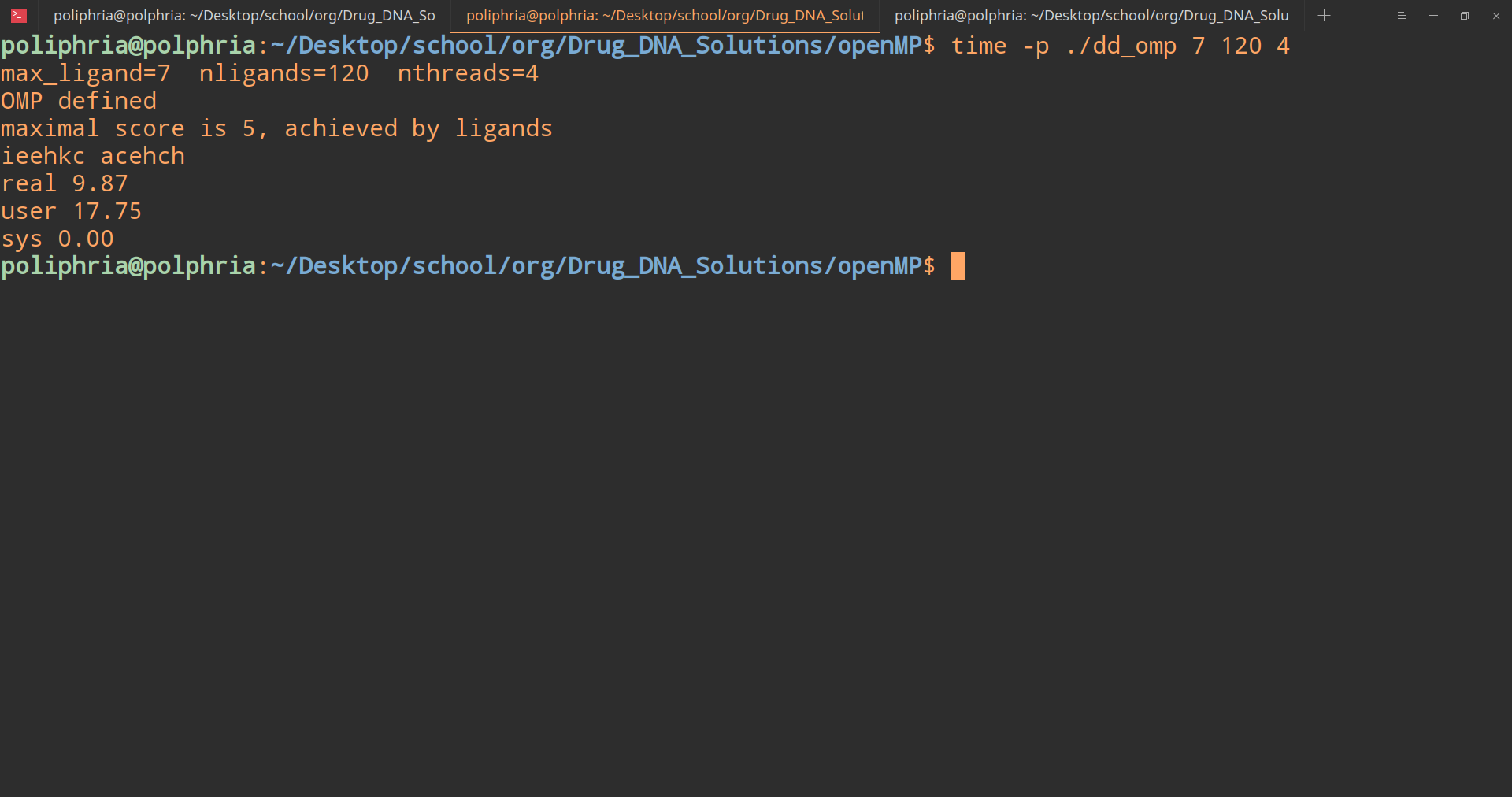
**Time for OpenMP program with one thread**

****

**Time for openMP and threads version with 2 threads**

****

**Time for openMP and threads version with 3 threads**

****

**Time for openMP and threads version with 4 threads.**

**Links:**

* **Github:** <https://github.com/Team-Raspberry>
* **Slack:** [teamraspberryorg.slack.com](http://teamraspberryorg.slack.com)
* **Youtube:** <https://youtu.be/Fmo7SeuleS8>