

### **Parallel Programming questions:**

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

In a parallel program, the first step is it divides the processes into individual threads and are run concurrently. The threads are joined back and then the entire program is then executed. An example of this is when the problem involves calculations such as finding the minimum or a maximum number in an array. To solve this program in parallel, we can split the array into smaller chunks and find the min/max value concurrently mentioning that each smaller chunk is not dependent on other chunks and then joining the results at the end and going through that to get the min/max.

2. What is MapReduce?

MapReduce is a parallel programming model that is convenient for executing large amounts of data. And there are two specific functions in this model called map and reduce.

3. What is a map and what is reduce?

Map function generates key value pairs by taking in and utilizing a wide range of inputs. The reduce function connects the inputs based on their keys and handles them according to the instructions they are grouped in.

4. Why MapReduce?

MapReduce is mainly utilized in situations where a large number of data is needed to be executed and executing calculations with that data.

5. Show an example for MapReduce.

An example of MapReduce is finding repeated numbers in an array. Every distinct number is mapped by its own distinct key following with data reduction and combining them at the end by just counting based on their key values.

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

MapReduce firstly processes by dividing the inputs into individual chunks so that the Mapper can be performed. Then the Mapper produces key value pairs based on the specific input. These keys are segregated into individual threads so that it can be performed concurrently inside which they are arranged in order and compressed in order for computing a valid result. Later after the threads have completed their tasks, they are then connected back and ultimately the result is calculated and generated.

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

Distributed Grep: the Map function looks for a match with a specific pattern and then produces the line if a match has been seen. The reduce function then proceeds the line to the result. Inverted Index: the Map creates <word, document ID> pairs according to the information present in various documents. The reduce then assembles all the events of a specific word and categorizes them as sets of <word,document list>. This generates an indication of every occurrence of every specific word in every document. Reverse

web-linked graph: the Map produces <target, source> pairs for every URL indicating to a specific source. The reduce categorizes according to values of target into sets of <target, source list> to structure a graph that shows all the interrelated URLs.

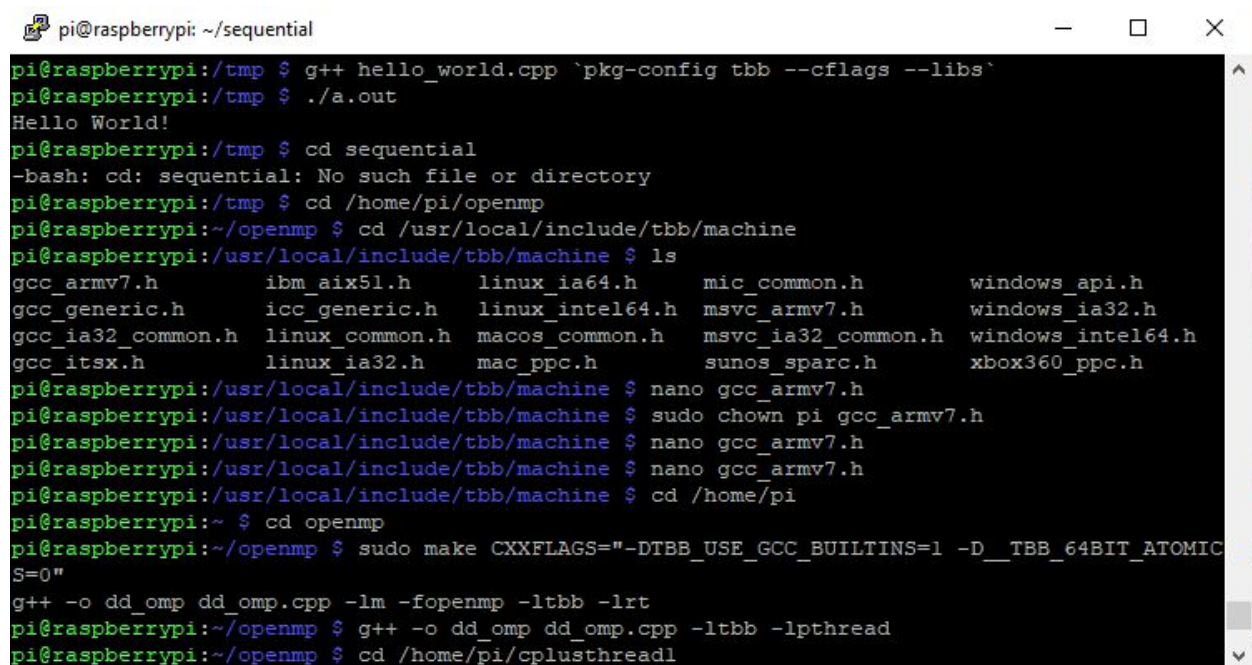
8. When do we use OpenMP, MPI and MapReduce (Hadoop), and why?

OpenMP is used when a program uses shared memory parallelism because every thread can share a unique iteration of a recurrent process. MPI works best for strongly integrated code where it can run among various platforms and this is done on parallel scientific applications. MapReduce (Hadoop) is used when a problem needs to generate a huge number of data since it can categorize and handle huge inputs in a short time using the Map and Reduce functions.

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

The main target of the Drug Design is to produce a form where it can control the DNA into generating proteins based on the shape wanted. Since there is a huge amount of combinations a ligand can occur as, a high number of ligands should be checked based on which protein they are suitable for and how they control the shapes of the proteins. There needs to be a high power for computation and complex algorithms are needed so that the tests are executed in a productive way.

### Programming:



```
pi@raspberrypi: ~/sequential
pi@raspberrypi:/tmp $ g++ hello_world.cpp `pkg-config tbb --cflags --libs`
pi@raspberrypi:/tmp $ ./a.out
Hello World!
pi@raspberrypi:/tmp $ cd sequential
-bash: cd: sequential: No such file or directory
pi@raspberrypi:/tmp $ cd /home/pi/openmp
pi@raspberrypi:/openmp $ cd /usr/local/include/tbb/machine
pi@raspberrypi:/usr/local/include/tbb/machine $ ls
gcc_armv7.h      ibm_aix51.h      linux_ia64.h      mic_common.h      windows_api.h
gcc_generic.h    icc_generic.h    linux_intel64.h    msvc_armv7.h      windows_ia32.h
gcc_ia32_common.h linux_common.h    macos_common.h    msvc_ia32_common.h windows_intel64.h
gcc_itsx.h       linux_ia32.h      mac_ppc.h         sunos_sparc.h     xbox360_ppc.h
pi@raspberrypi:/usr/local/include/tbb/machine $ nano gcc_armv7.h
pi@raspberrypi:/usr/local/include/tbb/machine $ sudo chown pi gcc_armv7.h
pi@raspberrypi:/usr/local/include/tbb/machine $ nano gcc_armv7.h
pi@raspberrypi:/usr/local/include/tbb/machine $ nano gcc_armv7.h
pi@raspberrypi:/usr/local/include/tbb/machine $ cd /home/pi
pi@raspberrypi:~ $ cd openmp
pi@raspberrypi:/openmp $ sudo make CXXFLAGS="-DTBB_USE_GCC_BUILTINS=1 -D__TBB_64BIT_ATOMIC_S=0"
g++ -o dd_omp dd_omp.cpp -lm -fopenmp -ltbb -lrt
pi@raspberrypi:/openmp $ g++ -o dd_omp dd_omp.cpp -ltbb -lpthread
pi@raspberrypi:/openmp $ cd /home/pi/cplusthreadl
```

For the omp and the thread solutions to compile and execute, it was necessary for us to install a tbb library that could process the solutions. The guide used was: <https://url.cn/5lDAmwS>.

```

pi@raspberrypi:~/sequential $ time -p ./dd_serial
maximal score is 5, achieved by ligands
acehch ieehkc
real 146.78
user 146.77
sys 0.00
pi@raspberrypi:~/sequential $

```

The serial solution takes 146.78 seconds of realtime which is around 2.5 minutes for the default number of ligands and the ligand length.

```

pi@raspberrypi:~/openmp $ time -p ./dd_omp 1
max_ligand=1 nligands=120 nthreads=4
OMP not defined
maximal score is 1, achieved by ligands
p c r c n p p h o c c n n w i r r c p i o t r c e t a w h r n i r r o w y r h y y c w a c y
e o p e
real 0.01
user 0.01
sys 0.00

```

```

pi@raspberrypi:~/sequential $ cd /home/pi/openmp
pi@raspberrypi:~/openmp $ time -p ./dd_omp 2
max_ligand=2 nligands=120 nthreads=4
OMP not defined
maximal score is 2, achieved by ligands
o p t a h c a r o r h h a c t n a p r r t o n o
real 0.02
user 0.02
sys 0.00

```

```

pi@raspberrypi:~/openmp $ time -p ./dd_omp 3
max_ligand=3 nligands=120 nthreads=4
OMP not defined
maximal score is 3, achieved by ligands
w h t c h w h o p
real 0.09
user 0.08
sys 0.01

```

```

pi@raspberrypi:~/openmp $ time -p ./dd_omp 4
max_ligand=4 nligands=120 nthreads=4
OMP not defined
maximal score is 3, achieved by ligands
r d r y h k i c h c h w h t c i o o r d t o r e l i h j o
real 0.42
user 0.40
sys 0.02
pi@raspberrypi:~/openmp $

```

```

pi@raspberrypi:~/openmp $ time -p ./dd_omp 7
max_ligand=7 nligands=120 nthreads=4
OMP not defined
maximal score is 5, achieved by ligands
acehch ieehkc
real 149.84
user 149.83
sys 0.00

```

The runtime for the openmp solution begins with 0.01 seconds with a ligand length being 1 and increases upto 0.42 seconds in realtime when the ligand length is 4. And with the default nthreads being 7, the realtime is 149.84. Compared with the serial solution, openmp's solution

is much faster in execution. For both threads and openmp, the command means the same thing, where the first number indicates the max\_ligands and when you are entering a single number it by default means that you are indicating the max\_ligand. Based on the results, it is again concluded that the higher the ligand, the higher the realtime.

```
pi@raspberrypi:~/openmp $ time -p ./dd_omp 7 120 1
max_ligand=7 nligands=120 nthreads=1
OMP defined
maximal score is 5, achieved by ligands
acehch ieehkc
real 149.25
user 149.22
sys 0.02
```

```
pi@raspberrypi:~/openmp $ time -p ./dd_omp 7 120 2
max_ligand=7 nligands=120 nthreads=2
OMP defined
maximal score is 5, achieved by ligands
acehch ieehkc
real 133.72
user 177.76
sys 0.01
```

```
pi@raspberrypi:~/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 116.21
user 197.01
sys 0.02
```

```
pi@raspberrypi:~/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 100.52
user 203.84
sys 0.00
```

As it states in the question again to find the realtime when the threads are set to 2, 3 and 4. The command should have default values of 7 as a max\_ligand and 120 for nligands. And then initializing the threads according to the question. It is clearly seen that when the threads increase, the realtime decreases since the tasks are split among the threads to execute quicker.

```
pi@raspberrypi:~/cplusthreads1 $ time -p ./dd_threads 1
max_ligand=1 nligands=120 nthreads=4
maximal score is 1, achieved by ligands
i n r p p e c c c n n h w c c i o r p t p o r t c e a r w n h y y i y r c o w r o r p c e
w a h y
real 0.02
user 0.01
sys 0.01
```

```

pi@raspberrypi:~/openmp $ cd /home/pi/cplusthreads1
pi@raspberrypi:~/cplusthreads1 $ time -p ./dd_threads 2
max_ligand=2 nligands=120 nthreads=4
maximal score is 2, achieved by ligands
to no ac hh rr tn ta op ap or ar hc
real 0.02
user 0.00
sys 0.03

```

```

pi@raspberrypi:~/cplusthreads1 $ time -p ./dd_threads 3
max_ligand=3 nligands=120 nthreads=4
maximal score is 3, achieved by ligands
wht hop chw
real 0.04
user 0.11
sys 0.00

```

```

pi@raspberrypi:~/cplusthreads1 $ time -p ./dd_threads 4
max_ligand=4 nligands=120 nthreads=4
maximal score is 3, achieved by ligands
hkic orel ihjo rdry hch cio wht ordt
real 0.18
user 0.63
sys 0.00
pi@raspberrypi:~/cplusthreads1 $

```

```

pi@raspberrypi:~/cplusthreads1 $ time -p ./dd_threads 7
max_ligand=7 nligands=120 nthreads=4
maximal score is 5, achieved by ligands
ieehkc acehch
real 78.11
user 274.28
sys 0.04

```

Compared to the omp solution, the thread solution executes runtime that is similar but slightly faster by showing 0.18 seconds for realtime when it was executed with a ligand length of 4. When the command `time -p ./dd_threads` is entered, the numbers are based as the `max_ligand`, `nligands` and then the `nthreads` numbers. So when you type in that command with an individual number, it is mentioning the `max_ligand` number. So based on this it can be said that as the `max_ligands` increase, the realtime increases too because it takes longer to find the combination of a bigger letter than compared to finding a smaller combination of letter.

```

pi@raspberrypi:~/cplusthreads1 $ time -p ./dd_threads 7 120 1
max_ligand=7 nligands=120 nthreads=1
maximal score is 5, achieved by ligands
acehch ieehkc
real 153.57
user 153.55
sys 0.00

```

```

pi@raspberrypi:~/cplusthreads1 $ time -p ./dd_threads 7 120 2
max_ligand=7 nligands=120 nthreads=2
maximal score is 5, achieved by ligands
ieehkc acehch
real 113.01
user 210.24
sys 0.01

```



```

pi@raspberrypi:~/cplusthreads1 $ time -p ./dd_threads 7 120 3
max_ligand=7 nligands=120 nthreads=3
maximal score is 5, achieved by ligands
acehch ieehkc
real 94.11
user 257.05
sys 0.03

```

```

pi@raspberrypi:~/cplusthreads1 $ time -p ./dd_threads 7 120 4
max_ligand=7 nligands=120 nthreads=4
maximal score is 5, achieved by ligands
ieehkc acehch
real 72.80
user 255.80
sys 0.06

```

Since the question asks us to record the realtime it takes for 2 threads, 3 threads and 4 threads, it is reasonable to do command `time -p ./dd_threads 7 120` and then the number of threads. Because doing so, you are specifically indicating the `nthreads`. The realtime goes from 153.57, 113.01, 94.11 to 72.80 seconds. The realtime it takes decreases as the number of threads increases because the more threads there are, the more efficient the process becomes because the tasks are split among the threads so that the execution becomes faster.

```

pi@raspberrypi:~/cplusthreads1 $ cd /home/pi/openmp
pi@raspberrypi:~/openmp $ g++ -o dd_omp dd_omp.cpp -ltbb -lpthread -fopenmp
pi@raspberrypi:~/openmp $ wc -l dd_omp
152 dd_omp
pi@raspberrypi:~/openmp $ █

```

```

pi@raspberrypi:~ $ cd cplusthreads1/
pi@raspberrypi:~/cplusthreads1 $ wc -l dd_threads
145 dd_threads
pi@raspberrypi:~/cplusthreads1 $ █

```

Since there are 152 lines in openmp and 145 lines in the threads, the threads solution is little more compatible than the openmp solution according to the file sizes.

```

pi@raspberrypi:~/cplusthreads1 $ time -p ./dd_threads 7 120 4
max_ligand=7 nligands=120 nthreads=4
maximal score is 5, achieved by ligands
ieehkc acehch
real 72.80
user 255.80
sys 0.06

```

```

pi@raspberrypi:~/cplusthreads1 $ time -p ./dd_threads 7 120 5
max_ligand=7 nligands=120 nthreads=5
maximal score is 5, achieved by ligands
acehch ieehkc
real 70.51
user 245.72
sys 0.06

```

```
pi@raspberrypi:~/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 100.52
user 203.84
sys 0.00
```

In this we test the runtime by the default settings then by incrementing the threads from 4 to 5. Based on the numbers we see on the openmp solution, it can be said that the execution of the solution for openmp was improved from 100.52 to 98.51 which is a decrease of 2.01 seconds in time. And when using 5 threads instead of 4, the runtime improved for the threads solution as well going from 72.80 to 70.51 seconds which is a decrease of 2.29 seconds. The runtime between the threads and openmp stayed almost the same but threads decreased in time slightly more than openmp meaning it increased speed. This might be because the program might need more time compiling a larger amount of threads. Since the openmp had more number of lines compared to the threads, it is understood that the difference between 4 and 5 th for openmp will be slightly smaller than the difference between threads 4 and 5 since there are only 145 lines in threads.