Compilation and Running the code:

1. Extract the archived file in an appropriate folder. This folder should be able to hold a large file (approx 10GB).
2. Run the Makefile by running the **`make`** command.
3. Now, four executable files are created namely **`sequentialWrite`**, **`parallelWrite`**, **`sequentialRead`** and **`parallelRead`**.
4. The executables **`sequentialWrite`** and **`parallelWrite`** are used to write the files and the other two for reading the file and computing the average.
5. Each of the executables can take an optional command line argument which is the filename. By default the filename of the created file of random numbers is "**`rand_no`**".
6. One of the two writing executables need to be run first to ensure that the file exists. Then any one of the two reading executable files can be run.
7. The time taken for each code to run is displayed on the screen at the end of successful completion.
8. Clean the created executable files by giving the command: **`make clean`** (**`make clean`** does not remove the generated file containing the numbers)

Approximate Times taken:

For 1 billion numbers, the approx times taken are as follows:

- Sequential Writing: Approx. 191 secs
- Parallel Writing: Approx. 700 secs
- Sequential Reading: Approx. 280 secs
- Parallel Reading: Approx. 190 secs (with 2 or 3 threads)

Thus the optimal way is to write the file sequentially, but read the file in parallel.

Problems and approach followed to solve them:

The problems dealt here in these codes are as follows:

1. Writing a billion random numbers (integers) to a file in the shortest time.
2. Reading the billion random numbers written and computing the average of the same.

For each of the following, I wrote two programs to do the task. One performs the task in a sequential manner, while the other performs the task using threads, thus doing certain independent tasks in parallel.

Sequential Writing:

In the code named “**`sequential_write.cpp`**”, I generate numbers and then write them to the file. The most naive solution would be to generate the numbers one by one and then write them as and when they are generated. But this would lead to tremendous slowdowns because it would lead to 1 billion write operations on a file.

To reduce the number of I/O operations (in this case write operations), random numbers are generated and stored in a temporary buffer stringstream. After a fixed number of numbers have been stored in the stringstream, the entire stringstream is written on to the file at one go. This reduces the number of write operations on the file, thus reducing time.

Parallel Writing:
In the code named “**`parallel_write.cpp`**”, the two independent tasks of random number generation and their writing to a file are performed independently in two separate threads in parallel.

In one thread, the random numbers are generated and appended to a stringstream. The other thread’s job is to obtain the mutex on the stringstream, write it to the file and clear it. Thus the writing thread takes any opportunity to write the stringstream buffer to disk whenever it is free. This should have resulted in a faster write operation according to me. But it didn’t. Infact, writing to the file in parallel takes more time than writing to it sequentially as above.

Sequential Reading:

In “**`sequential_read.cpp`**” the input file containing the numbers is opened and then each integer is read and added to a sum variable. There is no need to perform additional buffering like in sequential write, because the ifstream automatically does some buffering behind the scenes. Thus, the file is sequentially read and then its average is computed. This is very straightforward and slightly inefficient.

Parallel Reading:

In “**`parallel_read.cpp`**” multiple threads are given their own chunk of the file. Thus, if there are four threads, the file is divided into four equal parts and then each thread works on a different part in parallel. Thus each thread computes the partial sum for that part, and then all the partial sums are added to get the total sum. This is divided by the number of random numbers in the file to get the average. Thus, this should reduce the running time of the reading operation as independent parallel reads are performed on the file. And this does actually reduce the running time of the code and runs considerably faster than the sequential reading. One drawback of this approach is that due to the precision of double (which holds the sum per thread), there are very slight inaccuracies (in the 3rd or 4th decimal place) in the average when multiple threads are used.

Possible Improvements:

1. If the file of containing the random numbers isn't supposed to be read by humans, the entire file can be written in binary mode. This would result in a much smaller file on the disk, thus reducing time taken to read and write. The amount of data to be written and read will be considerably lower. If the integer is written in character form, it takes 9-10 bytes on an average as compared to 4 bytes in binary form. Moreover, while reading, the conversion of the number as an ASCII string to an int will be saved, as now the binary data can be retrieved as it is.

Also, the division of file into chunks while reading in parallel will be easy, as each integer is just 4 bytes (**`sizeof(int)`**) of memory and the total file size will always be a multiple of **`sizeof(int)`** Thus, we know beforehand where in the file each number begins and ends.

Potential improvement in writing: ~2.5 times faster

Potential improvement in reading: ~2.5 times faster

Con: File would be in binary and not human readable

2. For some reason (maybe cache size limitation, or file fragmentation), the reading of any file larger than 1 GB suddenly becomes very slow in Linux. Any file slightly smaller than 1GB can be read much faster than a file slightly larger than 1GB. Thus, splitting the file into multiple files with unique names (in order) say **`basefilename.01`**, **`basefilename.02`** and so on with each file being slightly smaller than 1GB, will result in faster read operation.

Potential improvement in reading: ~3 times faster

Con: File will have to be split up into smaller files