

University of the West of England

Parallel Computing

LOGBOOK

Prepared by: 20034743

Course Code: UFCFFL-15-M

Nov 2, 2021

Introduction

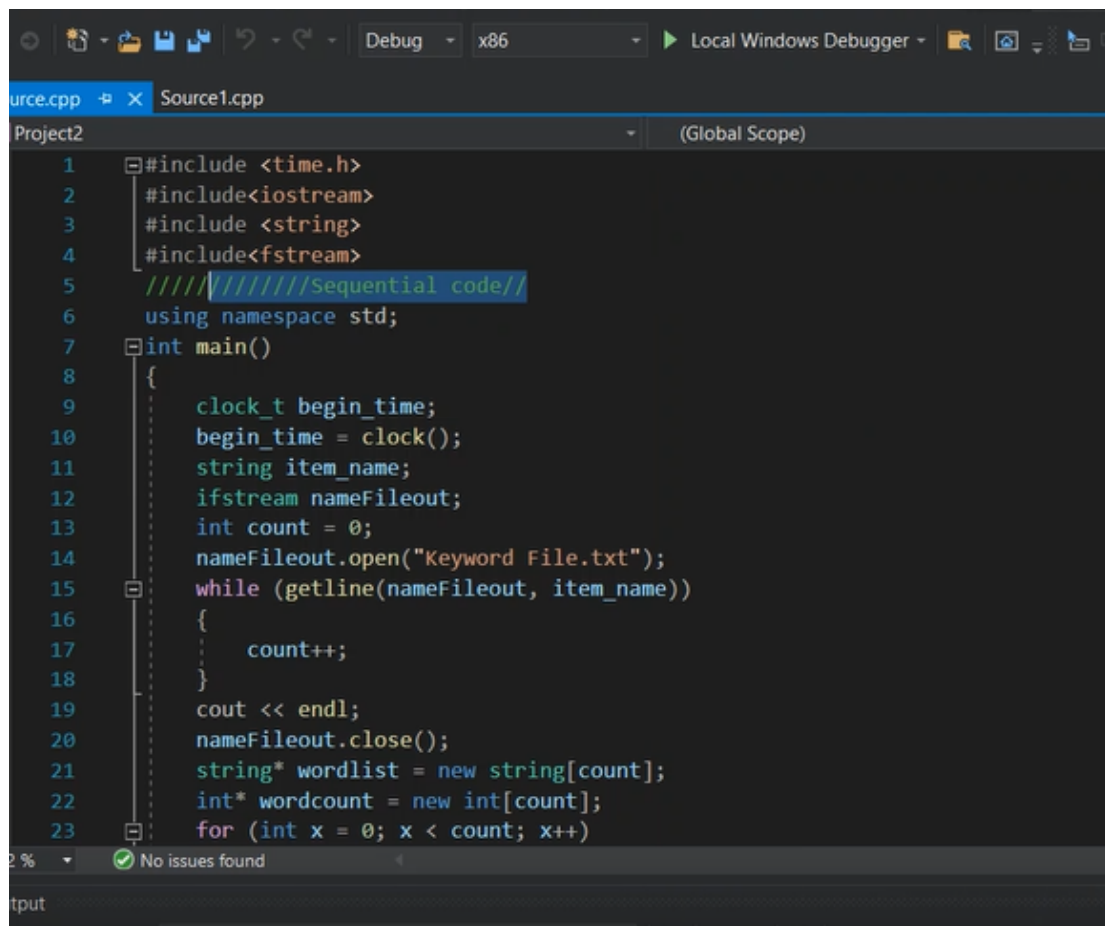
This report is about the parallelization using 2 Libraries mainly MPI and OpenMp. The report is about the detailed discussion on how the task are parallelized in terms of both these libraries and how these libraries work related to each other and what are the advantages and disadvantages of these libraries.

Also how the coding is being done and what are the strategies of implementing data decomposition and other methods to achieve speedups depending on the library at hand.

Progress Logs

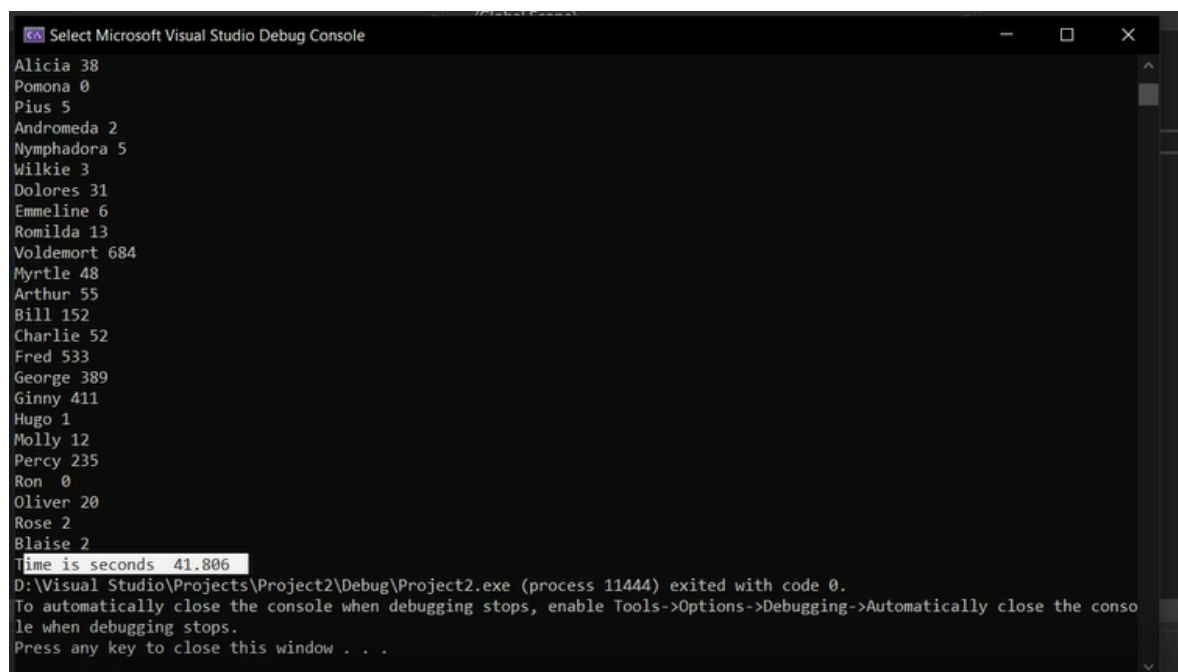
The code that we have to implement was based on finding a specific string value from a file. The code was implemented in c++ because of the better computational power that this language possesses and also the 2 libraries are also well suited for this language.

The code is first build upon sequential algorithm meaning it was running on one process at the time. The sequential code as its name suggests does not have any thread or process based coding in it thus the efficiency is not good as compared to the thread or process based if the data set that we want to compute on is large. If a small data set is provided then the sequential code is applied because it will work faster than the other 2 libraries because there will be no communication overhead between the processes or threads and no initialization of the environments.



```
1 #include <time.h>
2 #include<iostream>
3 #include <string>
4 #include<fstream>
5 //Sequential code//
6 using namespace std;
7 int main()
8 {
9     clock_t begin_time;
10    begin_time = clock();
11    string item_name;
12    ifstream nameFileout;
13    int count = 0;
14    nameFileout.open("Keyword File.txt");
15    while (getline(nameFileout, item_name))
16    {
17        count++;
18    }
19    cout << endl;
20    nameFileout.close();
21    string* wordlist = new string[count];
22    int* wordcount = new int[count];
23    for (int x = 0; x < count; x++)
```

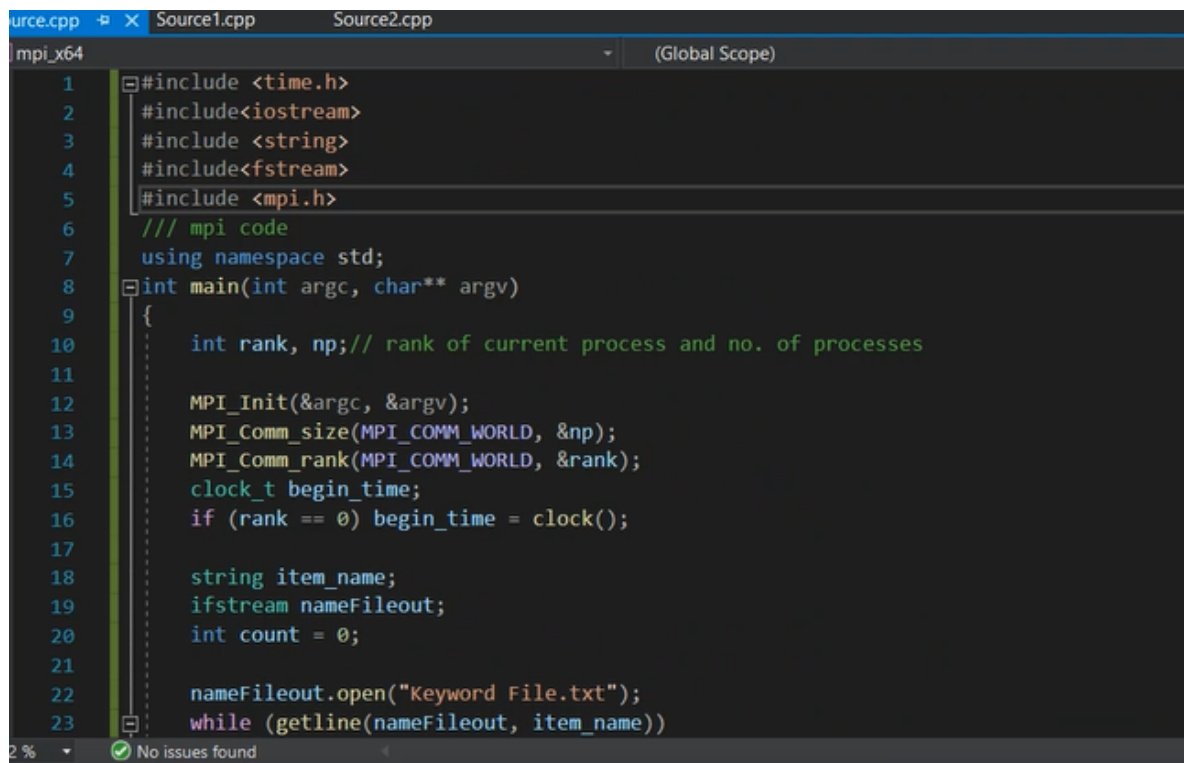
The output is:



```
Select Microsoft Visual Studio Debug Console
Alicia 38
Pomona 0
Pius 5
Andromeda 2
Nymphadora 5
Wilkie 3
Dolores 31
Emmeline 6
Romilda 13
Voldemort 684
Myrtle 48
Arthur 55
Bill 152
Charlie 52
Fred 533
George 389
Ginny 411
Hugo 1
Molly 12
Percy 235
Ron 0
Oliver 20
Rose 2
Blaise 2
Time is seconds 41.806
D:\Visual Studio\Projects\Project2\Debug\Project2.exe (process 11444) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

MPI Code

The code was then developed for the use of MPI processes so that it takes less time to find the string value. The MPI approach requires each process to have their own set of input file object and their own chunk in the file to execute. Thus when we made the chunks of each process the processes read from their own chunk and try to find the specific value. This approach is called data decomposition as we have made the data divided among process so that the computation get less time and is more efficient. The MPI process then after finding their value from the chunk prints them out in the console.

A screenshot of a C++ code editor with a dark theme. The editor shows a file named 'Source1.cpp' with the following code:

```
1 #include <time.h>
2 #include<iostream>
3 #include <string>
4 #include<fstream>
5 #include <mpi.h>
6 /// mpi code
7 using namespace std;
8 int main(int argc, char** argv)
9 {
10     int rank, np;// rank of current process and no. of processes
11
12     MPI_Init(&argc, &argv);
13     MPI_Comm_size(MPI_COMM_WORLD, &np);
14     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
15     clock_t begin_time;
16     if (rank == 0) begin_time = clock();
17
18     string item_name;
19     ifstream nameFileout;
20     int count = 0;
21
22     nameFileout.open("Keyword File.txt");
23     while (getline(nameFileout, item_name))
```

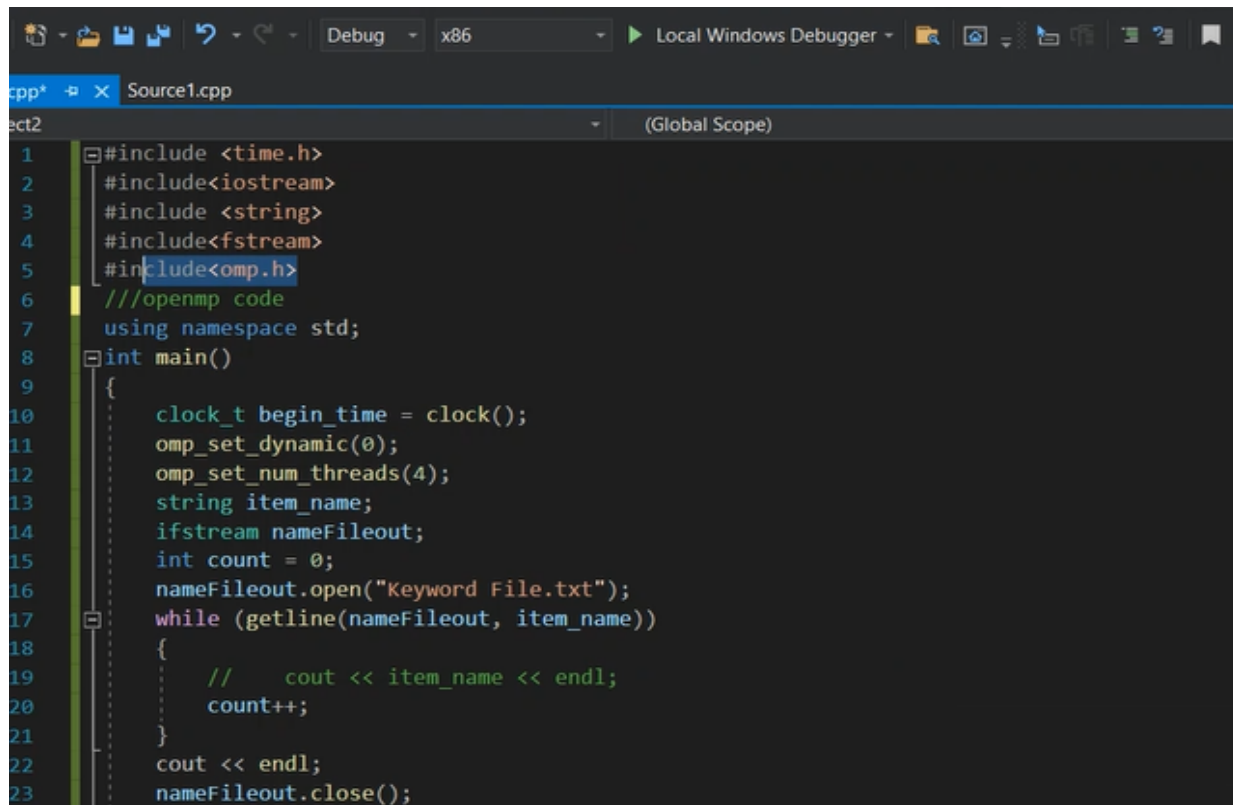
The editor interface includes a tab bar at the top with 'Source1.cpp' and 'Source2.cpp'. A dropdown menu on the left shows 'mpi_x64' and '(Global Scope)'. The bottom status bar indicates '2 %' and 'No issues found'.

The output is:

```
Select C:\Windows\system32\cmd.exe
Blaise 2
Hannah 21
Ludo 57
Bathilda 44
Katie 76
Cuthbert 1
Phineas 67
Sirius 612
Amelia 7
Susan 8
Terry 11
Lavender 98
Charity 6
Frank 54
Alecto 7
Amycus 8
Reginald 1
Cho 139
Penelope 5
Vincent 3
Colin 45
Dennis 11
Dirk 9
Barty 27
John 2
Fleur 145
Gabrielle 6
Dedalus 15
Amos 15
Cedric 198
Elphias 7
Antonin 3
Albus 135
Percival 3
Dudley 261
Marge 31
Petunia 190
Vernon 279
Time is seconds 14.052
D:\Visual Studio\Projects\mpi_x64\x64\Debug>
```

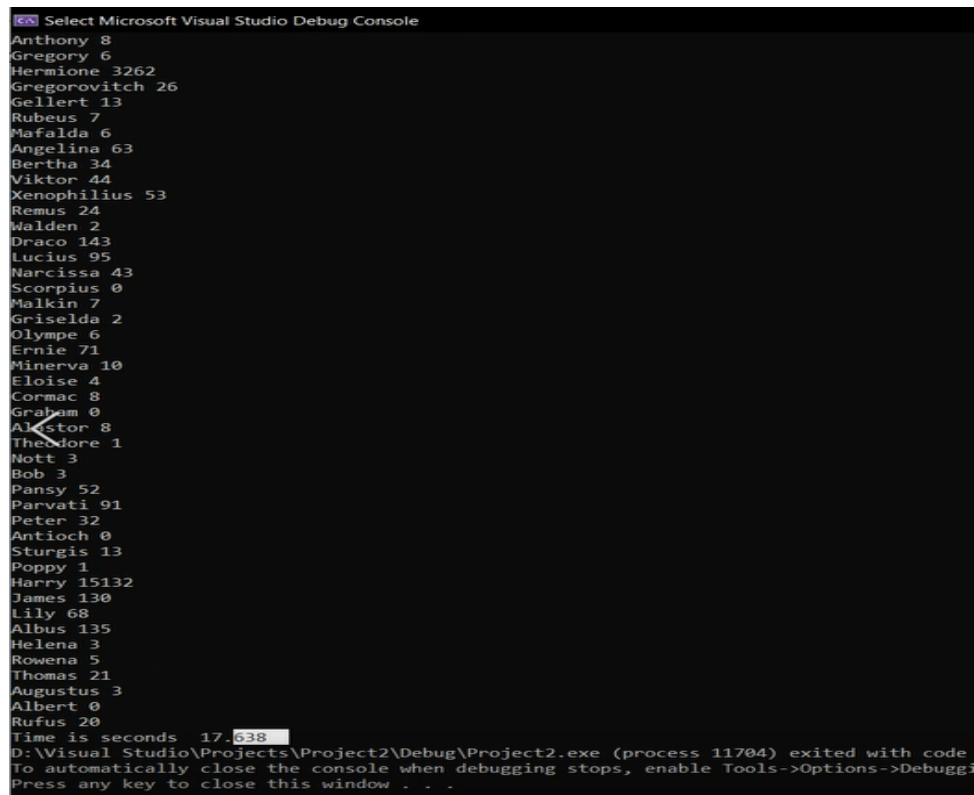
OpenMp Code

The code is then built upon OpenMP which is relatively easy as it is a compiler based language meaning the library is used for c/c++ language. The sequential code is used for this and we applied the `#pragma omp` clause to set the threads to work. The OpenMP if is used by the for clause automatically divides the chunks into their own threads if not specifically given by the schedule clause. If the schedule clause is given then the threads divide their chunks according to the schedule specified.



```
1 #include <time.h>
2 #include<iostream>
3 #include <string>
4 #include<fstream>
5 #include<omp.h>
6 ///openmp code
7 using namespace std;
8 int main()
9 {
10     clock_t begin_time = clock();
11     omp_set_dynamic(0);
12     omp_set_num_threads(4);
13     string item_name;
14     ifstream nameFileout;
15     int count = 0;
16     nameFileout.open("Keyword File.txt");
17     while (getline(nameFileout, item_name))
18     {
19         // cout << item_name << endl;
20         count++;
21     }
22     cout << endl;
23     nameFileout.close();
```

The output is:



```
Select Microsoft Visual Studio Debug Console
Anthony 8
Gregory 6
Hermione 3262
Gregorovitch 26
Gellert 13
Rubeus 7
Mafalda 6
Angelina 63
Bertha 34
Viktor 44
Xenophilus 53
Remus 24
Walden 2
Draco 143
Lucius 95
Narcissa 43
Scorpius 0
Malkin 7
Griselda 2
Olympe 6
Ernie 71
Minerva 10
Eloise 4
Cormac 8
Graham 0
Alastor 8
Theodore 1
Nott 3
Bob 3
Pansy 52
Parvati 91
Peter 32
Antioch 0
Sturgis 13
Poppy 1
Harry 15132
James 130
Lily 68
Albus 135
Helena 3
Rowena 5
Thomas 21
Augustus 3
Albert 0
Rufus 20
Time is seconds 17.638
D:\Visual Studio\Projects\Project2\Debug\Project2.exe (process 11704) exited with code
To automatically close the console when debugging stops, enable Tools->Options->Debugging
Press any key to close this window . . .
```

Methods of Parallelisation

The computing world now has many cores which is useful for parallelisation and data decomposition. First the computer was only single core but the hardware level roughly each year gets upgraded but after some level there was no upgrade level so we went on to other methods of fast and efficient computations that a computer can perform. To utilize this hardware, software needs to be written to take advantage of it, i.e. you have to go parallel. However, to really take advantage of modern computing hardware you have to write code that is targeted toward some method of parallel programming. There are 5 basic way to go parallel;

- Compiler assistance
- Library calls
- Directives
- Low level hardware targeting
- Message Passing

The first 4 of these are mostly focused on single node parallelism with vectorization, threading and offload accelerator targeting. This is very important since it is now possible to have a single workstation with dozens of CPU cores and hundreds of accelerator cores capable of handling possibly thousands of execution threads. A high end workstation can rival the compute capability of million dollar "Supercomputers" from only a few years ago.

The 5th method, Message Passing, is the traditional method of parallel computing and is still very important. Current MPI implementations can take advantage of multiple many-core CPU's on a single node with performance that is as good or sometimes better, than methods that are targeted more at multi-threading. And, for parallel computing across multiple nodes message passing is very well understood and widely used.

There are different architectures that were used to make the parallelism approaches work these are call Flynn's taxonomy.

- 1: SISD
- 2: SIMD
- 3: MISD
- 4: MIMD

SISD (Single Instruction Single Data)

By this name it is suggesting that it is a sequential program which takes only a single vector and applies a single instruction on it. This was used in single core architectures.

SIMD (Single Instruction Multiple Data)

By this name it suggests that we can have multiple vectors but the only a single operation could work on it. Suggesting that it has a touch of parallelism and interprocess communication.

MISD (Multiple Instruction Single Data)

By this name it suggests that there could be multiple instructions performed on the single data but it is not feasible as it would require ALU (Arithmetic and Logic unit) to have parallelism attach to it. But there can only be a single ALU to compute those instructions and then store them in memory thus this approach is not a feasible one.

MIMD (Multiple Instruction Multiple Data)

By this name it suggests that we can have multiple vectors and at the same time can perform various instructions on it this is basically what the threads and process achieve when they are used in the code. By using the multi core processes we can apply this approach. Also in cluster architecture the MIMD is used.

Applied Approach

The code was done based on data parallelism because we have a file that needs to be searched so the appropriate method to have is to divide the files into chunks and then have each process read their own chunk of file which saves time and we can fully grasp the power of multicore processes. This was the best way because the MPI process makes their own copy of code so we made sure that each copy of code have their own chunk of file to read from.

Performance Analysis

Both libraries have their pros and cons MPI uses interprocess communication very well but it does not have its own way of dividing the data the programmer has to do it by themselves similarly OpenMp has very good clauses that can automatically provide data decomposition and the programmer does not need to divide the chunks among processes but the interprocess communication is lacking between the processes.

Thus each have their own advantages and disadvantages. The performance is looked upon through many factors when it comes to parallelisation, mainly hardware and the optimization of code makes the computational level very much efficient.

In this approach the OpenMp worked more efficiently because there was no interprocess communication and the OpenMp have its own clause that handles data decomposition but nevertheless MPI is also very approachable but for this type of task where you need to read from a file and then search from it the OpenMp is your library to do that task. But if there is task that involves communication between processes then MPI will definitely work better for that. We can compare the performance of serial, OPENMP and MPI as:

Sequential:

Sequential code takes 20sec almost while openmp code take less then half time

OPENMP:

Openmp code with 4 threads provide best performance more then 2x (among only openmp codes)

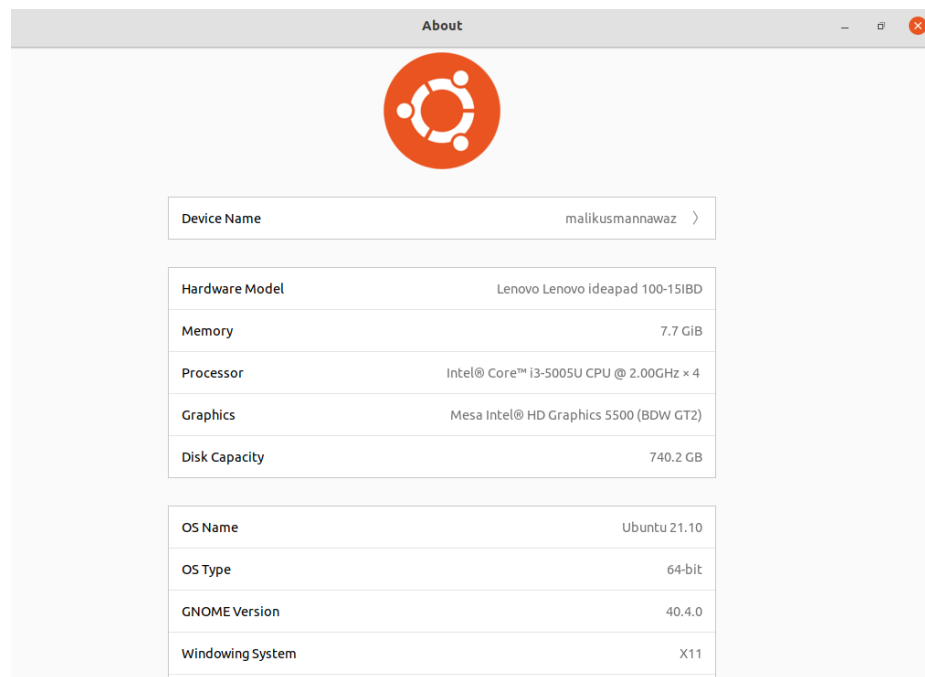
MPI:

Mpi code provides with 4 process provide round about 3x time

With 8 process most of the time waste on communication so, code run on 4 processes provide best performance among serial and openmp.

Code	Time(sec)	Time(sec)	Time(sec)
Sequential	39.4	39.5	39.4
openmp	17.1 (4 threads)	23.5(2 threads)	20.4(8 threads)
mpi	13.5(4 process)	21.9 (2 process)	17(8 process)

And mentioning our system specification is also important as well. I applied these approaches in different environment and got some different result. So, I concluded all the of above task on one main system with the specification are:



Conclusion

By looking at these 2 libraries we have found out how they differ from each other and when to use those 2 libraries. Also the types of architectures that was implemented so that parallelism could occur. We have also looked upon the fact that the task given by this was better suited for OpenMp as it has data decomposition methods that could solve problems of that magnitude very easily.

Also we looked upon what are the ways to make the code parallel. Mainly it requires good hardware support along with good optimization of code and a thought out library that can support that kind of parallelism.

GitHub Links:

Serial Code:

<https://github.com/MalikUsmanNawaz/ParallelCOMPpublicResit/blob/25d51881a72f757163ccdedc726f6e6743f572d1/Serial.cpp>

OPENMP:

<https://github.com/MalikUsmanNawaz/ParallelCOMPpublicResit/blob/25d51881a72f757163ccdedc726f6e6743f572d1/omp.cpp>

OPENMPI:

<https://github.com/MalikUsmanNawaz/ParallelCOMPpublicResit/blob/25d51881a72f757163ccdedc726f6e6743f572d1/mpi.cpp>

Keywords file:

<https://github.com/MalikUsmanNawaz/ParallelCOMPpublicResit/blob/25d51881a72f757163ccdedc726f6e6743f572d1/Keyword%20File.txt>

Textfile:

<https://github.com/MalikUsmanNawaz/ParallelCOMPpublicResit/blob/25d51881a72f757163ccdedc726f6e6743f572d1/Text%20File.txt>

Readme:

<https://github.com/MalikUsmanNawaz/ParallelCOMPpublicResit/blob/25d51881a72f757163ccdedc726f6e6743f572d1/Readme.txt>

References:

- Puget Systems. 2021. *Puget Systems | Built to Order Workstations, Custom Computers*. [online] Available at: <<https://www.pugetsystems.com/>> [Accessed 5 November 2021].
- Puget Systems. 2021. *Puget Systems | Built to Order Workstations, Custom Computers*. [online] Available at: <<https://www.pugetsystems.com/>> [Accessed 5 November 2021].