Note: All programs should be run with 4 threads in order to function corrrectly

Question 01:

```cpp
#include <iostream>
#include <map>
#include <omp.h>
#include <fstream>
#include <bits/stdc++.h>
using namespace std;

int main()
{
    vector<string> wordsmap;
    fstream file;
    string word, filename, output_;
    fstream result;
    result.open("Q1_ans.txt", ios_base::out);
    int count = 0, temp_count = 0, parallel_count = 0;
    filename = "test.txt";
    file.open(filename.c_str());
    while (file >> word)
    {
        wordsmap.push_back(word);
        count++;
    }
    file.close();
    int chunk = count / 4;
    vector<string> text_of_article[4];
    file.open(filename.c_str());

    while (file >> word)
    {
        if (temp_count < chunk * 1)
        {
            text_of_article[0].push_back(word);
            temp_count++;
            continue;
        }
        else if (temp_count < chunk * 2)
        {
            text_of_article[1].push_back(word);
            temp_count++;
            continue;
        }
```

```
        else if (temp_count < chunk * 3)
        {
            text_of_article[2].push_back(word);
            temp_count++;
            continue;
        }
        else if (temp_count < chunk * 4)
        {
            text_of_article[3].push_back(word);
            temp_count++;
            continue;
        }
    }
    omp_set_num_threads(4);
#pragma omp parallel reduction(+ \
                               : parallel_count)
    {
        int local_sum = 0;
        int id = omp_get_thread_num();
        for (int i = 0; i < text_of_article[id].size(); i++)
        {
            parallel_count++;
            local_sum++;
        }
#pragma omp critical
        {
            output_ += "Words counted by thread ";
            output_ += to_string(id);
            output_ += " are ";
            output_ += to_string(local_sum);
            output_ += ".\n";
            cout << "Words counted by thread " << id << " are " << local_sum <<
"." << endl;
        }
    }

    cout << "The total number of words in the given document when counted
serially is " << count << "." << endl;
    cout << "The total number of words is counted via parallel threads is " <<
parallel_count << "." << endl;

    output_ += "The total number of words in the given document when counted
serially is  ";
    output_ += to_string(count);
    output_ += ".\n";
```

```
    output_ += "The total number of words is counted via parallel threads is ";
    output_ += to_string(parallel_count);
    output_ += ".\n";
    result << output_;
    result.close();
}
```

Question 02:

```
#include <iostream>
#include <map>
#include <fstream>
#include <bits/stdc++.h>
#include <mpi.h>
// Use 4 threads to run this program!
using namespace std;

int main(int argc, char *argv[])
{
    int npes, rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &npes);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    vector<string> wordsmap;
    vector<string> partition[4];
    fstream file;
    string word, output_;
    int count = 0, temp_count = 0, parallel_count = 0, chunk = 0;
    file.open("test.txt");
    while (file >> word)
    {
        wordsmap.push_back(word);
        count++;
    }
    chunk = count / 4;
    file.clear();
    file.seekg(0);
    while (file >> word)
    {
        if (temp_count < chunk * 1)
        {
            partition[0].push_back(word);
            temp_count++;
            continue;
```

```
        }
        else if (temp_count < chunk * 2)
        {
            partition[1].push_back(word);
            temp_count++;
            continue;
        }
        else if (temp_count < chunk * 3)
        {
            partition[2].push_back(word);
            temp_count++;
            continue;
        }
        else if (temp_count < chunk * 4)
        {
            partition[3].push_back(word);
            temp_count++;
            continue;
        }
    }

    temp_count = 0;
    for (int i = 0; i < partition[rank].size(); i++)
    {
        temp_count++;
    }
    MPI_Send(&temp_count, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    cout << "The total number of words in the current thread is " << temp_count
<< ".\n";

    MPI_Barrier(MPI_COMM_WORLD);
    if (rank == 0)
    {
        fstream result;
        result.open("Q2_ans.txt", ios_base::out);
        if (!result)
            cout << "Answer file couldn't be created" << endl;
        string output_;
        int global_sum = 0;
        for (int i = 0; i < 4; i++)
        {
            MPI_Recv(&temp_count, 1, MPI_INT, i, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
            global_sum += temp_count;
        }
```

```
        cout << "The global sum  of all the threads is " << global_sum << "." <<
endl;
        output_ += "The global sum  of all the threads is ";
        output_ += to_string(global_sum);
        output_ += ".\n";
        result << output_;
        result.close();
    }
    MPI_Finalize();
    file.close();
};
```

Question 03:

```
#include <iostream>
#include <map>
#include <omp.h>
#include <fstream>
#include <bits/stdc++.h>
using namespace std;

int main()
{
    map<string, int> wordsmap;
    vector<string> text_of_article[4];
    fstream file;
    string word, filename, output_;
    int count = 0, temp_count = 0;
    fstream result;
    result.open("Q3_ans.txt", ios_base::out);
    filename = "test.txt";
    file.open(filename.c_str());
    while (file >> word)
    {
        wordsmap.insert({word, 0});
        count++;
    }
    file.close();
    int chunk = count / 4;
    file.open(filename.c_str());

    while (file >> word)
```

```
    {
        if (temp_count < chunk * 1)
        {
            text_of_article[0].push_back(word);
            temp_count++;
            continue;
        }
        else if (temp_count < chunk * 2)
        {
            text_of_article[1].push_back(word);
            temp_count++;
            continue;
        }
        else if (temp_count < chunk * 3)
        {
            text_of_article[2].push_back(word);
            temp_count++;
            continue;
        }
        else if (temp_count < chunk * 4)
        {
            text_of_article[3].push_back(word);
            temp_count++;
            continue;
        }
    }
    omp_set_num_threads(4);
#pragma omp parallel
    {
        int local_sum = 0;
        int id = omp_get_thread_num();
        for (int i = 0; i < text_of_article[id].size(); i++)
        {
#pragma omp critical
            {
                local_sum++;
                wordsmap[text_of_article[id][i]]++;
            }
        }
        cout << "Words counted by thread " << id << " are " << local_sum << "."
<< endl;
#pragma omp critical
        {
            output_ += "Words counted by thread ";
            output_ += to_string(id);
```

```
            output_ += " are ";
            output_ += to_string(local_sum);
            output_ += ".\n";
        }
    }

    temp_count = 0;
    result << "The frequency map of the text is given below: \nThe count of each
thread and the total words will be listed at the end of the map (its quite big
map so you'll have to scroll for a bit or you can use Ctrl+End to reach the end
of the file)\n";
    string entry;
    for (auto it = wordsmap.cbegin(); it != wordsmap.cend(); it++)
    {
        entry = "";
        temp_count += it->second;
        cout << it->first << ":" << it->second << endl;
        entry += it->first;
        entry += ":";
        entry += to_string(it->second);
        entry += "\n";
        result << entry;
    }

    output_ += "The size of the map (unique words) is ";
    output_ += to_string(wordsmap.size());
    output_ += " and the total number of words is ";
    output_ += to_string(count);
    output_ += ".\n";
    output_ += "The total number of words is counted via parallel threads is ";
    output_ += to_string(temp_count);
    output_ += ".\n";
    result << output_;
    cout << output_;
    result.close();
}
```

Question 04:

```cpp
#include <iostream>
#include <map>
#include <omp.h>
#include <fstream>
#include <mpi.h>
#include <bits/stdc++.h>
using namespace std;

int main(int argc, char *argv[])
{
    int npes, rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &npes);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    map<string, int> wordsmap;
    vector<string> text_of_article[4];
    fstream file;
    string word, filename, output_;
    int count = 0, temp_count = 0;
    fstream result;
    result.open("Q3_ans.txt", ios_base::out);
    filename = "test.txt";
    file.open(filename.c_str());
    while (file >> word)
    {
        wordsmap.insert({word, 0});
        count++;
    }
    file.close();
    int chunk = count / 4;
    file.open(filename.c_str());

    while (file >> word)
    {
        if (temp_count < chunk * 1)
        {
            text_of_article[0].push_back(word);
            temp_count++;
            continue;
        }
        else if (temp_count < chunk * 2)
        {
            text_of_article[1].push_back(word);
```

```
            temp_count++;
            continue;
        }
        else if (temp_count < chunk * 3)
        {
            text_of_article[2].push_back(word);
            temp_count++;
            continue;
        }
        else if (temp_count < chunk * 4)
        {
            text_of_article[3].push_back(word);
            temp_count++;
            continue;
        }
    }

    int local_sum = 0;
    for (int i = 0; i < text_of_article[rank].size(); i++)
    {
        wordsmap[text_of_article[rank][i]]++;
        local_sum++;
    }
    MPI_Send(&local_sum, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);

    MPI_Barrier(MPI_COMM_WORLD);
    result.close();
    if (rank == 0)
    {
        fstream result;
        result.open("Q4_ans.txt", ios_base::out);
        if (!result)
            cout << "Answer file couldn't be created" << endl;
        string output_;
        int global_sum = 0;
        for (int i = 0; i < 4; i++)
        {
            MPI_Recv(&local_sum, 1, MPI_INT, i, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
            global_sum += local_sum;
        }
        cout << "The global sum of all the words in threads is " << global_sum <<
"." << endl;
        output_ += "The global sum of all the words in threads is ";
        output_ += to_string(global_sum);
```

```
        output_ += ".\n";
        result << output_;
        result.close();
    }

    MPI_Finalize();
}
```

Question 05:

a) Static Schedule vs Dynamic Schedule in OpenMP :

i) Static Scheduling:

In static scheduling each thread is assigned a fixed number of iterations by the compiler
Its syntax is:
#pragma omp parallel for schedule (static, chunk-size)
Where static tells the manner of mapping and chunk-size tells the size of each chunk mapped to an execution thread.

```
1   #include <iostream>
2   #include <map>
3   #include <omp.h>
4   #include <fstream>
5   #include <bits/stdc++.h>
6   using namespace std;
7
8   int main()
9   {
10  #pragma omp parallel for schedule(static, 5)
11      for (int i = 0; i < 100; i++)
12      {
13          cout << "This" << endl;
14      }
15  }
```

ii) Dynamic Scheduling:

In Dynamic scheduling iterations are dynamically mapped on each execution thread by the compiler. In this paradigm chunks are assigned to each execution threads as the become idle.

The syntax for dynamic scheduling of the for loop is given by:

#pragma omp parallel for schedule (dynamic, chunk-size)

Where “dynamic” represents the manner of mapping and “chunk-size” represents the size of each chunk dynamically mapped on each execution thread.

```
#include <iostream>
#include <map>
#include <omp.h>
#include <fstream>
#include <bits/stdc++.h>
using namespace std;

int main()
{
#pragma omp parallel for schedule(dynamic, 5)
    for (int i = 0; i < 100; i++)
    {
        cout << "This" << endl;
    }
}
```

b) Barrier Vs NoWait in OpenMP

i) Barrier:

The barrier directive is used for synchronizing threads at a given point of execution during the program. The program doesn’t proceed further till all threads in the program have reached at a particular point.

The syntax for barrier clause is given below:

#pragma omp parallel barrier

```
#include <iostream>
#include <map>
#include <omp.h>
#include <fstream>
#include <bits/stdc++.h>
using namespace std;

int main()
{
    omp_set_num_threads(4);
#pragma omp parallel
    {
        cout << "This will be printed without barrier" << endl;
        if (omp_get_thread_num() == 0)
            cout << "This will print when thread number =0" << endl;
#pragma omp barrier
        cout << "This will wait for all threads to reach this place." << endl;
    }
```

ii) Nowait:

The nowait clause is used to remove implicit barriers, it is used to override any synchronization that would otherwise occur at the end of the construct.

The syntax for nowait clause is:

#pragma omp parallel for nowait

```
#include <iostream>
#include <map>
#include <omp.h>
#include <fstream>
#include <bits/stdc++.h>
using namespace std;

int main()
{
    omp_set_num_threads(4);
#pragma omp parallel
    {
        cout << "This will be printed without barrier" << endl;
        if (omp_get_thread_num() == 0)
            cout << "This will print when thread number =0" << endl;
#pragma omp barrier
        cout << "This will wait for all threads to reach this place." << endl;
#pragma omp nowait
        cout << "This line will be printed before the barrier line because of the no wait clause." << endl;
    }
```

c) Broadcast() vs Scatter() in MPI

i) Broadcast:

The Broadcast function is used to send the date from one member of a group to all members of the other group.

```
#include <iostream>
#include <map>
#include <omp.h>
#include <fstream>
#include <bits/stdc++.h>
#include <mpi.h>
using namespace std;

int main(int agrc, char *argv[])
{
    int npes, rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &npes);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int array[100] = {0};

    MPI_Bcast(&arr, 25, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Finalize();
}
```

ii) Scatter:

In scatter each node sends a unique message of size m to every other node.

```
#include <iostream>
#include <map>
#include <omp.h>
#include <fstream>
#include <bits/stdc++.h>
#include <mpi.h>
using namespace std;

int main(int agrc, char *argv[])
{
    int npes, rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &npes);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int array[100] = {0};

    MPI_Scatter(&arr, 25, MPI_INT, &arr_, 25, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Finalize();
}
```

d) Gather() vs Reduce() in MPI

i. Gather:

The gather function gathers the different data sent to it by the other processes of the group.

```
#include <iostream>
#include <map>
#include <omp.h>
#include <fstream>
#include <bits/stdc++.h>
#include <mpi.h>
using namespace std;

int main(int agrc, char *argv[])
{
    int npes, rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &npes);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int array[100] = {0};

    MPI_Gather(&arr, 25, MPI_INT, &arr_, 25, MPI_INT, 0,
    MPI_COMM_WORLD);
    MPI_Finalize();
}
```

ii. Reduce:

Reduce function gathers all the data sent to it by the other processes of the group and applies the reduction function to it such as MIP_MIN, MPI_MAX,MPI_SUM etc.

```
#include <iostream>
#include <map>
#include <omp.h>
#include <fstream>
#include <bits/stdc++.h>
#include <mpi.h>
using namespace std;

int main(int agrc, char *argv[])
{
    int npes, rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &npes);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int array[100] = {0};

    MPI_Gather(&arr, &arr_, 25, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    MPI_Finalize();
}
```

e) Broadcast() vs Alltoall() in MPI

i) Broadcast:

The Broadcast function is used to send the date from one member of a group to all members of the other group.

```
#include <iostream>
#include <map>
#include <omp.h>
#include <fstream>
#include <bits/stdc++.h>
#include <mpi.h>
using namespace std;

int main(int agrc, char *argv[])
{
    int npes, rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &npes);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int array[100] = {0};

    MPI_Bcast(&arr, 25, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Finalize();
}
```

ii) AlltoAll:

The all to all function enables the user to send a different portion of the data to all processes including itself.

```
#include <iostream>
#include <map>
#include <omp.h>
#include <fstream>
#include <bits/stdc++.h>
#include <mpi.h>
using namespace std;

int main(int agrc, char *argv[])
{
    int npes, rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &npes);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int array[100] = {0};

    MPI_Alltoall(&arr, 25, MPI_INT, &arr_, 25, MPI_INT, MPI_COMM_WORLD);
    MPI_Finalize();
}
```