Shourya Kothari
60004190103
B2

Experiment No. 5

Aim: Implement a demonstration of balancing of workload on the MPI platform.

Theory:

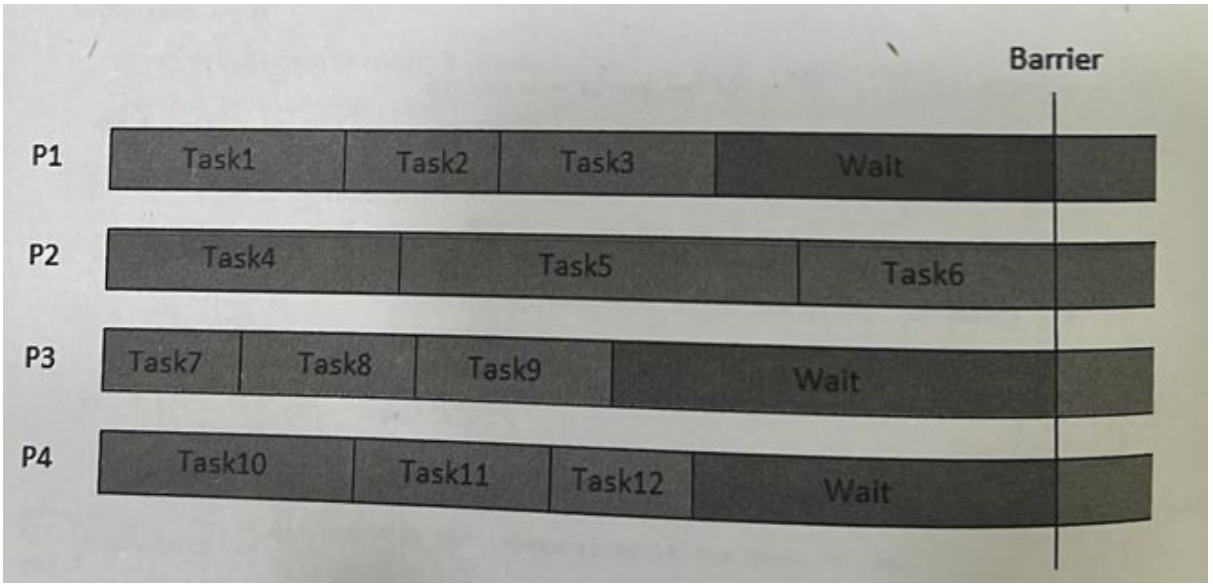
1. 1. Parallelization is used to achieve a speedup of the runtime of a program while using the available processors as efficiently as possible.

2. When splitting the workload across multiple processors, the speedup describes the achieved reduction of the runtime compared to the serial version. Obviously, the higher the speedup, the better. However, the effect of increasing the number of processors usually goes down at a certain point when they cannot be utilised optimally anymore. This parallel efficiency is defined as the achieved speedup divided by the number of processors used.

3. It usually is not possible to achieve a speedup equal to the number of processors used as most applications have strictly serial parts.

4. Load balancing is of great importance when utilising multiple processors as efficiently as possible. Adding more processors creates a noticeable amount of synchronisation overhead. Therefore, it is only beneficial if there is enough work present to keep all processors busy at the same time. This means tasks should not be assigned randomly as this might lead to serial behaviour where only one processor is working while the others have already finished their tasks.

5. e.g. at synchronisation points (e.g. barriers) or at the end of the program.



Code:

```
#include <iostream>
#include <ctime>
#include <mpi.h>

using namespace std;

double int_thetha(double E) {
   double result = 0;
   for (int k = 0; k < 20000; k++) {
      result += E * k;
      return result;
   }
}

int main() {
   int n = 3500000;
   int counter = 0;
   time_t timer;
   int start_time = time(&timer);
   int myid, numprocs;
   int k;
   double integrate, result;
   double end = 0.5;
   double start = -2.;
   double E;
   double factor = (end - start) / (n * 1.);
```

```
    integrate = 0;
    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    for (k = myid; k < n + 1; k += numprocs) {
        E = start + k * (end - start) / n;
        if ((k == 0) || (k == n))
            integrate += 0.5 * factor * int_thetha(E);
        else
            integrate += factor * int_thetha(E);
        counter++;
    }
    cout << "process " << myid << " took " << time(&timer) - start_time << "s" << endl;
    cout << "process " << myid << " performed " << counter << " computations" << endl;
    MPI_Reduce(&integrate, &result, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    if (myid == 0)
        cout << result << endl;
    MPI_Finalize();
    return 0;
}
```

Output:

```
PS C:\Users\djsce.student\source\repos\hpc_exp5\x64\debug> mpiexec -n 4 hpc_exp5.exe
process 0 took 0s
process 0 performed 875001 computations
process 2 took 0s
process 3 took 0s
process 1 performed 875000 computations
process 2 performed 875000 computations
process 3 performed 875000 computations
0
process 2 took 0s
process 0 took 0s
process 2 performed 875000 computations
process 0 performed 875001 computations
process 3 took 0s
process 3 performed 875000 computations
process 1 took 0s
process 1 performed 875000 computations
0
PS C:\Users\djsce.student\source\repos\hpc_exp5\x64\debug> mpiexec -n 3 hpc_exp5.exe
process 1 took 0s
process 2 took 0s
process 0 took 0s
process 1 performed 1166667 computations
process 2 performed 1166667 computations
process 0 performed 1166667 computations
0
PS C:\Users\djsce.student\source\repos\hpc_exp5\x64\debug> mpiexec -n 1 hpc_exp5.exe
process 0 took 0s
process 0 performed 3500001 computations
0
```

Conclusion: From this experiment, we studied how to use MPI to construct a parallel programme that computes and prints the cube of each integer in a range of numbers. We set up the MPI environment, divided the number range into sub-ranges for each process, computed the cube of each number in the sub-range, and printed the outcome using the MPI4PY module.