Distributed Computing

Lab – I

K. Manoj Kumar

2022BCS0184.

PART – A:

Q1. A). Write a simple C/C++ to print "Hello World".

Code :

```
#include<iostream>
int main() {
   std::cout << "Hello World";
}
```

Output :

```
PS C:\Users\manoj\Desktop\CPP\llel comp> ./a
Hello World
```

B). Write an openMP program to print "Hello World" with parallel computation

Program :

```
#include<iostream>
#include<omp.h>
int main() {
   #pragma omp parallel
   {
      std::cout << "Hello World\n";
   }
```

}

Output :

```
PS C:\Users\manoj\Desktop\CPP\llel comp> g++ -fopenmp q1.cpp
PS C:\Users\manoj\Desktop\CPP\llel comp> ./a
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
```

Q2. A). Write an openMP program to print Hello World with parallel computation along with the corresponding thread id.

Program :

```
#include<iostream>

#include<omp.h>

int main() {

  #pragma omp parallel

  {

    std::cout << "Hello World from Thread:" << omp_get_thread_num() << std::endl;

  }

}
```

Output :

```
PS C:\Users\manoj\Desktop\CPP\llel comp> g++ -fopenmp q1.cpp
PS C:\Users\manoj\Desktop\CPP\llel comp> ./a
Hello World from Thread:0
Hello World from Thread:7
Hello World from Thread:5
Hello World from Thread:2
Hello World from Thread:3
Hello World from Thread:9
Hello World from Thread:1
Hello World from Thread:1
Hello World from Thread:Hello World from Thread:11Hello World from Thread:10
Hello World from Thread:6
Hello World from Thread:8

4
```

B). Write an openMP program to print number **of threads.**

Program :

```
#include<iostream>
#include<omp.h>
int main() {
  #pragma omp parallel
  {
    #pragma omp single
    {
      std::cout << "No. of threads: " << omp_get_num_threads() << std::endl;
    }
  }
}
```

Output :

```
PS C:\Users\manoj\Desktop\CPP\llel comp> ./a
No. of threads: 12
```

Q3. Write an openMP to print "Hello World" for k times using for loop and compute the execution time of sequential and parallel run.

Program:

```
#include<iostream>
#include<omp.h>
#include<chrono>

void sequential() {
   for(int i = 0;i < 10;i++) {
      std::cout << "Hello world\n";
   }
}

void parallel() {
   #pragma omp parallel for

        for(int i = 0;i < 10;i++)
        {
           std::cout << "Hello world from iteration :" << i << " thread:" <<
omp_get_thread_num() << std::endl;
        }

}
```

```
int main() {
    auto s = std::chrono::high_resolution_clock::now();
    sequential();
    auto e = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> dur = e - s;
    std::cout << "Execution time for sequential:" << dur.count() << std::endl;
    auto s_p = std::chrono::high_resolution_clock::now();
    parallel();
    auto e_p = std::chrono::high_resolution_clock::now();
    dur = e_p - s_p;
    std::cout << "Execution time for parallel:" << dur.count() << std::endl;

}
```

Output:

```
Hello world
Hello world
Hello world
Hello world
Execution time for sequential:0.0023916
Hello world from iteration :Hello world from iteration :5thr
Hello world from iteration :8thread:8
Hello world from iteration :1thread:1
0thread:0
0thread:0
0thread:0
0thread:0
0thread:0
Hello world from iteration :7thread:Hello world from iterati
7
Hello world from iteration :2thread:2
Hello world from iteration :6thread:6
Hello world from iteration :3thread:3

Execution time for parallel:0.0043695
```

PART - B

Q4. Write an openMP program (C++) for matrix multiplication with serial and parallel computation and note the execution time of the same. Test the execution time with different dimension of the matrix (D). For example N=10, 100, 1000…… (Any Larger Number).

Program:

```
#include<bits/stdc++.h>


using namespace std;
```

```
void print(std::vector<std::vector<int>> arr) {
   int row = arr.size() ,column = arr[0].size();
   for(int i = 0;i < row;i++) {
      for(int j = 0;j < column;j++) {
         std::cout << arr[i][j] << " ";
      }
      std::cout << std::endl;
   }
}


void p_matmul(std::vector<std::vector<int>>
&arr1,std::vector<std::vector<int>> &arr2,std::vector<std::vector<int>>
&result,size_t dim) {
   #pragma omp parallel for collapse(3) shared(result)
   for(size_t i = 0;i < dim;i++) {
      for(size_t j = 0;j < dim;j++) {
         result[i][j] = 0;
         for(size_t k = 0;k < dim;k++) {


            result[i][j] += arr1[i][k] * arr2[k][j];
         }
      }
   }
}
void matmul(std::vector<std::vector<int>>
&arr1,std::vector<std::vector<int>> &arr2,std::vector<std::vector<int>>
&result,size_t dim) {
```

```
    for(size_t i = 0;i < dim;i++) {
        for(size_t j = 0;j < dim;j++) {
            result[i][j] = 0;
            for(size_t k = 0;k < dim;k++) {

                result[i][j] += arr1[i][k] * arr2[k][j];
            }
        }
    }
}

void fillMatrix(vector<vector<int>> &arr,size_t dimension) {
    std::random_device rd;
    unsigned seed =
std::chrono::system_clock::now().time_since_epoch().count();
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> uni(0, 100);
    for(int i = 0;i < dimension;i++) {
        for(int j = 0;j < dimension;j++) {
            arr[i][j] = uni(gen);
        }
    }
}

void run(const size_t &dim) {
```

```
 std::vector<std::vector<int>> arr1 (dim,std::vector<int>(dim));
   std::vector<std::vector<int>> arr2 (dim,std::vector<int>(dim));
   std::vector<std::vector<int>> result(dim,std::vector<int>(dim));
   fillMatrix(arr1,dim);
   fillMatrix(arr2,dim);
   auto start = std::chrono::high_resolution_clock::now();
   matmul(arr1,arr2,result,dim);
   auto end = std::chrono::high_resolution_clock::now();
   std::chrono::duration<double> duration = end - start;

   auto start_p = std::chrono::high_resolution_clock::now();
   p_matmul(arr1,arr2,result,dim);
   auto end_p = std::chrono::high_resolution_clock::now();
   std::chrono::duration<double> duration_p = end_p - start_p;

   std::cout << "\nExecution time for Multiplication(serial) of matrix dimension
" << dim << " is :" << duration.count() ;
   std::cout << "\nExecution time for Multiplication(parallel) : of matrix
dimension " << dim << "is : " << duration_p.count() ;

}

int main() {

   for(int i : {10,100,1000}) {
      run(i);
```

```
    }


    return 0;
}
```

Output:

```
PS C:\Users\manoj\Desktop\CPP\llel comp> g++ -fopenmp mat_mul.cpp
PS C:\Users\manoj\Desktop\CPP\llel comp> ./a

Execution time for Multiplication(serial) of matrix dimension 10 is :7.1e-06
Execution time for Multiplication(parallel) : of matrix dimension 10is : 0.0023208
Execution time for Multiplication(serial) of matrix dimension 100 is :0.0065619
Execution time for Multiplication(parallel) : of matrix dimension 100is : 0.0027097
Execution time for Multiplication(serial) of matrix dimension 1000 is :5.22352
Execution time for Multiplication(parallel) : of matrix dimension 1000is : 1.90577
```

Q5. Write an openMP program (C++) for the following sorting algorithms with serial and parallel computation and note the execution time of the same. Analyse the output with different high N values, where N is the number of elements to be sorted. (Like above table) a) Merge sort b) Quick sort

Program(Merge Sort) :

```
#include<bits/stdc++.h>


using namespace std;

#define THRESHOLD 50


void merge(vector<int> &arr, int left, int mid, int right) {
    int l = mid - left + 1;
    int r = right - mid;
    vector<int> left_array(l);
    vector<int> right_array(r);
```

```
for (int i = 0; i < l; i++) {
    left_array[i] = arr[left + i];
}
for (int i = 0; i < r; i++) {
    right_array[i] = arr[mid + 1 + i];
}

int i = 0, j = 0, k = left;
while (i < l && j < r) {
    if (left_array[i] <= right_array[j]) {
        arr[k] = left_array[i];
        i++;
    } else {
        arr[k] = right_array[j];
        j++;
    }
    k++;
}

while (i < l) {
    arr[k] = left_array[i];
    i++;
    k++;
}

while (j < r) {
```

```
        arr[k] = right_array[j];
        j++;
        k++;
    }
}
void print(vector<int>& arr) {
    std::cout << "[";
    for (int i : arr) {
        cout << i << ", ";
    }
    std::cout << "]";
}

void mergeSort(vector<int> &arr, const int &left, const int &right) {
    if (left >= right)
        return;
    int mid = left + (right - left) / 2;
    mergeSort(arr, left, mid);
    mergeSort(arr, mid + 1, right);
    merge(arr, left, mid, right);
}

void mergeParallelSort(vector<int> &arr, const int &left, const int &right) {
    if(left >= right) {
        return;
    }
```

```
    int mid = left + (right - left) / 2;
    if(right - left < THRESHOLD) {
        mergeSort(arr, left, right);
        return;
    }
    #pragma omp parallel
    {
        #pragma omp single
        {
            #pragma omp taskgroup
            {
                #pragma omp task shared(arr)
                mergeParallelSort(arr, left, mid);

                #pragma omp task shared(arr)
                mergeParallelSort(arr, mid + 1, right);
            }
        }

    }
    #pragma omp taskwait
    merge(arr, left, mid,  right);
}

void fill_vector(vector<int> &arr) {
    std::random_device rd;
```

```
    unsigned seed =
std::chrono::system_clock::now().time_since_epoch().count();
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> uni(0, 100);
    for(int i = 0;i < arr.size();i++) {
        arr[i] = uni(gen);
    }
}

void run(const size_t &array_length) {
    vector<int> arr(array_length);

    fill_vector(arr);
    auto s = std::chrono::high_resolution_clock::now();
    mergeSort(arr, 0, array_length - 1);
    auto e = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> run_time = e - s;
    std::cout << "size [" << std::setw(4)<< std::left << array_length <<"] runtime
for serial merge sort " << run_time.count() << "s" << std::endl;

    fill_vector(arr);
    auto ps = std::chrono::high_resolution_clock::now();
    mergeParallelSort(arr, 0, array_length - 1);
    auto pe = std::chrono::high_resolution_clock::now();
    run_time = pe - ps;
    std::cout << "size [" << std::setw(4) << array_length << "] runtime for parall
merge sort " << run_time.count() << "s" << std::endl;
```

```
}

int main() {
   for(const auto& i : {10,100,1000}) {
      run(i);
   }
}
```

Output:

```
PS C:\Users\manoj\Desktop\CPP\llel comp> g++ -fopenmp .\merge_sort.cpp
PS C:\Users\manoj\Desktop\CPP\llel comp> ./a
size [10  ] runtime for serial merge sort 6.3e-06s
size [10  ] runtime for parall merge sort 3.8e-06s
size [100 ] runtime for serial merge sort 4.69e-05s
size [100 ] runtime for parall merge sort 0.0036892s
size [1000] runtime for serial merge sort 0.000344s
size [1000] runtime for parall merge sort 0.0007356s
```

Program (Quick Sort) :

```
#include<bits/stdc++.h>

using namespace std;

int partition (vector<int>& arr,int left,int right) {
   int piv = arr[right];
   int l = left - 1;

   for(int i = left;i < right;i++) {
      if(arr[i] < piv) {
         l++;
```

```
            swap(arr[l],arr[i]);
        }
    }
    swap(arr[l + 1],arr[right]);
    return l + 1;
}

void quick(vector<int>& arr,int left,int right) {
    if(left < right) {
        int pivot = partition(arr,left,right);
        quick(arr,left,pivot - 1);
        quick(arr,pivot + 1,right);
    }
}

void print(vector<int>& arr) {
    std::cout << "[";
    for (int i : arr) {
        cout << i << ", ";
    }
    std::cout << "]";
}

void p_quick(vector<int>& arr,int left,int right) {
        if (left < right) {
        int pivot = partition(arr, left, right);
```

```
    // Parallelize recursive calls if the partition is large enough
    #pragma omp parallel sections if(right - left > 50) // Limit small partitions from spawning too many threads
    {
      #pragma omp section
      {
        p_quick(arr, left, pivot - 1);
      }
      #pragma omp section
      {
        p_quick(arr, pivot + 1, right);
      }
    }
  }
}

void randomize(vector<int> &arr) {
  std::random_device rd;
  int n = arr.size();
  unsigned seed = std::chrono::system_clock::now().time_since_epoch().count();
  std::mt19937 gen(rd());
  std::uniform_int_distribution<> uni(INT16_MIN, INT16_MAX);
  for(int i = 0;i < n;i++) {
    arr[i] = uni(gen);
  }
```

```
}

void run(const size_t &length) {
   vector<int> arr(length);

   randomize(arr);
   auto start = std::chrono::high_resolution_clock::now();
   quick(arr,0,arr.size() - 1);
   auto end = std::chrono::high_resolution_clock::now();
   std::chrono::duration<double> duration = end - start;
   std::cout << "\nexecution time for serial of array length "<< length << " is : "
<< duration.count() << endl;

   randomize(arr);
   auto start_p = std::chrono::high_resolution_clock::now();

   #pragma omp parallel
   {
      #pragma omp single
      {
         p_quick(arr, 0, arr.size() - 1);
      }
   }
   auto end_p = std::chrono::high_resolution_clock::now();

   std::chrono::duration<double> duration_p = end_p - start_p;
```

```
    std::cout << "\nExecution time for parallel  of array length "<< length << " is :" << duration_p.count() << endl;
}

int main() {
    for(int len : {10,100,1000}) {
        run(len);
    }
    return 0;
}
```

Output: