Shrikrushna Zirape :

LP5 : Assignment2

41283 (BE2)

---

1. **Bubble Sort**

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <omp.h>

using namespace std;

void bubble_sort(int* arr, int n)
{
  for(int i=0; i<n; i++)
  {
    bool swapped = false;
    for(int j=0; j<n-i-1; j++)
    {
      if(arr[j] > arr[j+1])
      {
        swap(arr[j], arr[j+1]);
        swapped = true;
      }
    }
    if(!swapped) break;
  }
}

void parallel_bubble_sort(int* arr, int n)
{
  for(int i=0; i<n; i++)
  {
    bool swapped = false;
    #pragma omp parallel for shared(arr,swapped)
    for(int j=0; j<n-i-1; j++)
```

```
      {
        if(arr[j] > arr[j+1])
        {
          swap(arr[j], arr[j+1]);
          swapped = true;
        }
      }
      if(!swapped) break;
    }
}

int main()
{
    srand(time(0));

    const int N = 1000;
    int arr[N];
    for(int i=0; i<N; i++)
      arr[i] = rand()%N;

    int seq_start = clock();
    bubble_sort(arr, N);
    int seq_end = clock();

    int par_start = clock();
    parallel_bubble_sort(arr, N);
    int par_end = clock();

    double seq_time = (seq_end - seq_start) / (double) CLOCKS_PER_SEC;
    double par_time = (par_end - par_start) / (double) CLOCKS_PER_SEC;

    cout << "Sequential Time: " << seq_time << " seconds" << endl;
    cout << "Parallel Time: " << par_time << " seconds" << endl;

    double speedup = seq_time / par_time;
    double efficiency = speedup / omp_get_max_threads();
    double throughput = N / par_time;
```

```
    cout << "Speedup: " << speedup << endl;
    cout << "Efficiency: " << efficiency << endl;
    cout << "Throughput: " << throughput << endl;

    return 0;

}



/*

to run :
g++ -fopenmp bubblesort.cpp -o bubblesort
./bubblesort



output :

Sequential Time: 0.003 seconds
Parallel Time: 0.001 seconds
Speedup: 3
Efficiency: 0.375
Throughput: 1e+06


*/
```

2. **Merge Sort**

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <omp.h>

#define N 10000

using namespace std;

void merge(int arr[], int left, int middle, int right)
{
  int i, j, k;
  int n1 = middle - left + 1;
  int n2 = right - middle;

  int L[n1], R[n2];

  for (i = 0; i < n1; i++)
    L[i] = arr[left + i];
  for (j = 0; j < n2; j++)
    R[j] = arr[middle + 1 + j];

  i = 0;
  j = 0;
  k = left;

  while (i < n1 && j < n2) {
    if (L[i] <= R[j]) {
      arr[k] = L[i];
      i++;
    }
    else {
      arr[k] = R[j];
      j++;
    }
    k++;
```

```
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void merge_sort(int arr[], int left, int right)
{
    if (left < right) {
        int middle = left + (right - left) / 2;

        merge_sort(arr, left, middle);
        merge_sort(arr, middle + 1, right);

        merge(arr, left, middle, right);
    }
}

void parallel_merge_sort(int arr[], int left, int right)
{
    if (left < right) {
        int middle = left + (right - left) / 2;

        #pragma omp parallel sections
        {
            #pragma omp section
            {
                parallel_merge_sort(arr, left, middle);
```

```
        }

        #pragma omp section
        {
            parallel_merge_sort(arr, middle + 1, right);
        }
      }

      merge(arr, left, middle, right);
   }
}

int main()
{

   int arr[N];
   int i;

   // Initialize array with random values
   srand((unsigned)time(NULL));
   for (i = 0; i < N; i++) {
      arr[i] = rand() % 10000;
   }

   int seq_start = clock();
   merge_sort(arr, 0, N - 1);
   int seq_end = clock();

   int par_start = clock();
   parallel_merge_sort(arr, 0, N - 1);
   int par_end = clock();

  double seq_time = (seq_end - seq_start) / (double) CLOCKS_PER_SEC;
   double par_time = (par_end - par_start) / (double) CLOCKS_PER_SEC;

   cout << "Sequential Time: " << seq_time << " seconds" << endl;
   cout << "Parallel Time: " << par_time << " seconds" << endl;
```

```
    double speedup = seq_time / par_time;
    double efficiency = speedup / omp_get_max_threads();
    double throughput = N / par_time;

    cout << "Speedup: " << speedup << endl;
    cout << "Efficiency: " << efficiency << endl;
    cout << "Throughput: " << throughput << endl;

    return 0;
}



/*

output :

Sequential Time: 0.002 seconds
Parallel Time: 0.073 seconds
Speedup: 0.0273973
Efficiency: 0.00342466
Throughput: 136986
*/
```