LP1 Assignment HPC H3

Parallelize Sorting Algorithms using OpenMP

Date - 3rd September, 2020.

Assignment Number - HPC H3

Title

Parallel Sorting Algorithms

Problem Definition

For Bubble Sort and Merge Sort, based on existing sequential algorithms, design and implement parallel algorithm utilizing all resources available

Learning Objectives

- Learn parallel decomposition of sorting algorithms.
- Learn parallel computing using OpenMP

Learning Outcomes

I will be able to decompose sorting algorithms into subproblems, to solve sub problems using threads in OpenMP

Software Packages and Hardware Apparatus Used

- Operating System : 64-bit Ubuntu 18.04
- Browser : Google Chrome
- Programming Language : C++ (OpenMP header file included), Python 3
- Jupyter Notebook Environment : Google Colaboratory

Related Mathematics

Mathematical Model

Let S be the system set:

S = {s; e; X; Y; Fme; Ff; DD; NDD; Fc; Sc}

s=start state

e=end state

X=set of inputs

X = {X1}

where X1 = Array of arbitrary size

Y= Output Set

Y = {Y1,Y2,Y3,Y4} where

Y1 = Sorted Array X1 using Serial Merge Sort

Y2 = Sorted Array X1 using Parallel Merge Sort

Y3 = Sorted Array X1 using Serial Bubble Sort

Y4 = Sorted Array X1 using Parallel Bubble Sort

Fme is the set of main functions

Fme = {f0} where

f0 = output display function

Ff is the set of friend functions

Ff = {f1,f2,f3,f4,f5,f6,f7,f8,f9} where

f1 = Copy Constructor for class Array

f2 = Function to initialize array with random values

f3 = Overloaded insertion operator

f4 = Merge Sort Serial (Overloaded)

f5 = Merge Sort Parallel (Overloaded)

f6 = Bubble Sort Serial

f7 = Bubble Sort Parallel

f8 = Merge Sub Function for Merge Sort Algorithms

f9 = Swap Sub Function for Bubble Sort Algorithms

DD = Deterministic Data

Input Array X1

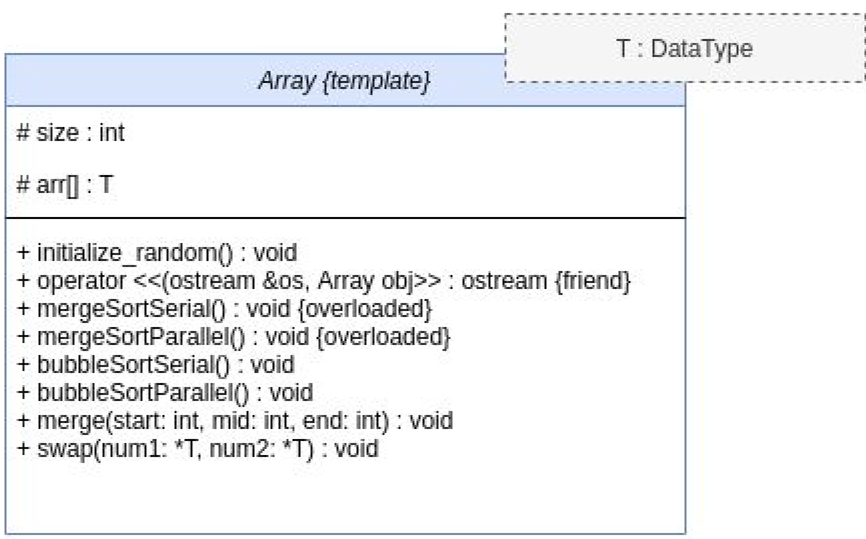
NDD=Non-deterministic data

No non deterministic data

Fc =failure case:

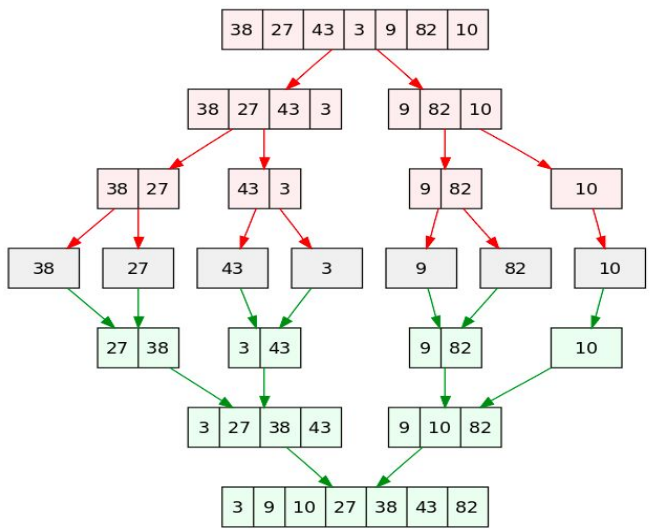
No failure case identified for this application

Class Diagram

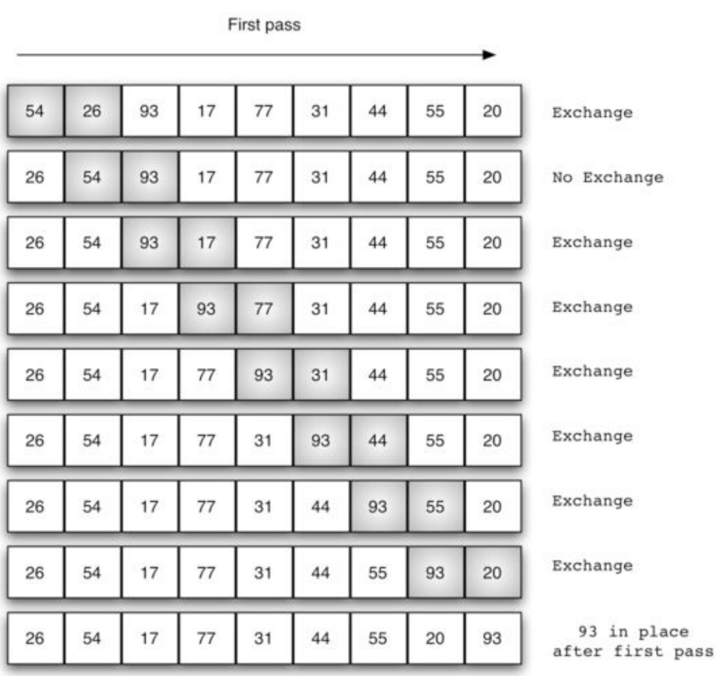


Diagrams

Merge Sort



Bubble Sort



Concepts related Theory

OpenMP is an implementation of multithreading, a method of parallelizing whereby a *primary* thread (a series of instructions executed consecutively) *forks* a specified number of *sub*-threads and the system divides a task among them. The threads then run concurrently, with the runtime environment allocating threads to different processors.

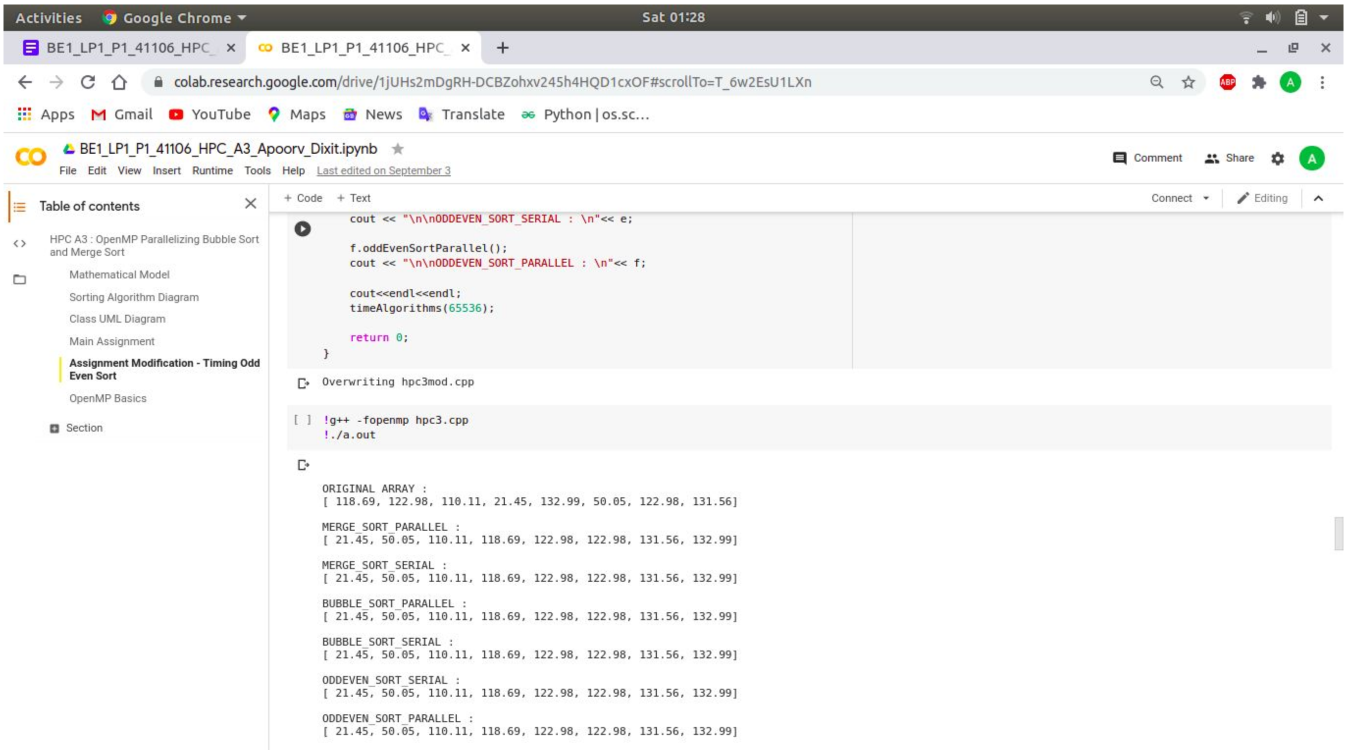
The section of code that is meant to run in parallel is marked accordingly, with a compiler directive that will cause the threads to form before the section is executed. Each thread has an *id* attached to it which can be obtained using a function (called `omp_get_thread_num()`). The thread id is an integer, and the primary thread has an id of *0*. After the execution of the parallelized code, the threads *join* back into the primary thread, which continues onward to the end of the program.

By default, each thread executes the parallelized section of code independently. *Work-sharing constructs* can be used to divide a task among the threads so that each thread executes its allocated part of the code. Both task parallelism and data parallelism can be achieved using OpenMP in this way.

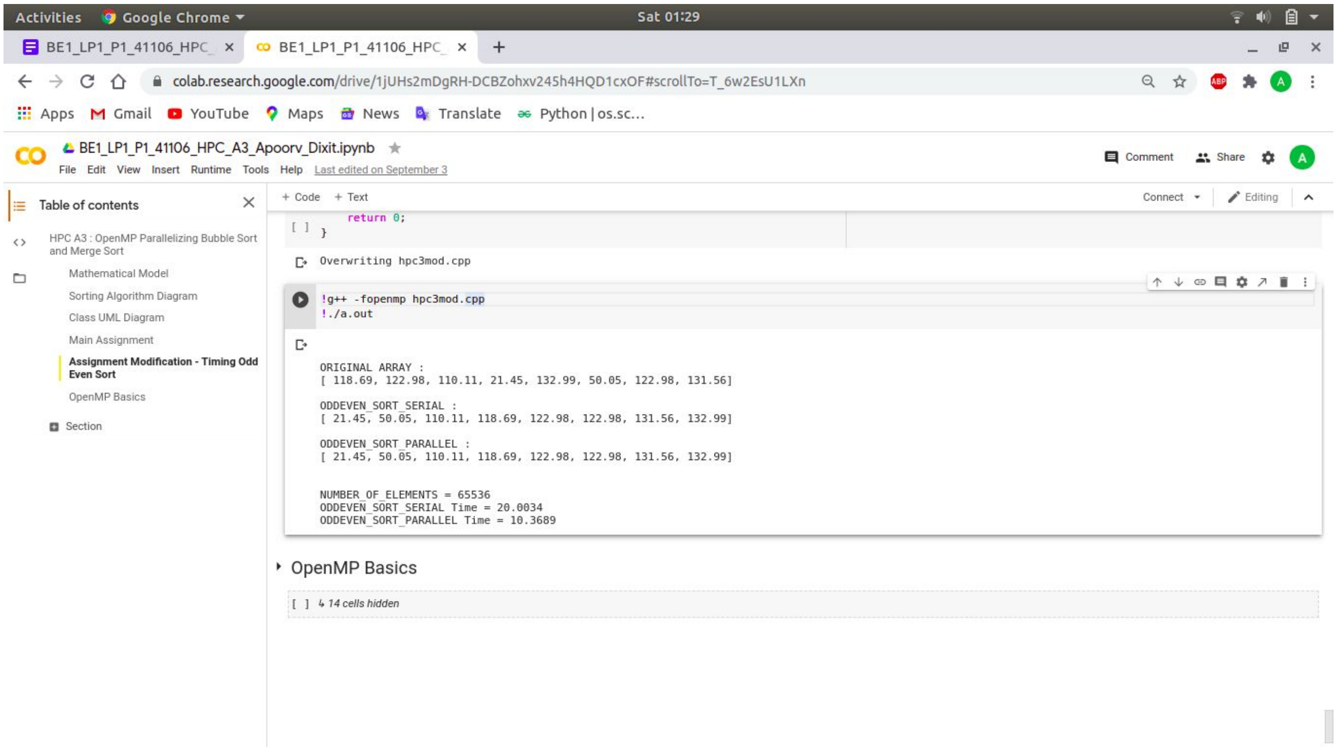
The runtime environment allocates threads to processors depending on usage, machine load and other factors. The runtime environment can assign the number of threads based on environment variables, or the code can do so using functions. The OpenMP functions are included in a header file labelled `omp.h` in C/C++.

Output Screenshots

Serial and Parallel Merge Sort and Bubble Sort



Modification Odd Even Parallel Sort (Timing the Algorithms)



Source Code

```
#include<iostream>
#include<omp.h>
using namespace std;

#define MERGE_SORT_PARALLEL 0
#define MERGE_SORT_SERIAL 1
#define BUBBLE_SORT_PARALLEL 2
#define BUBBLE_SORT_SERIAL 3
#define ODDEVEN_SORT_SERIAL 4
#define ODDEVEN_SORT_PARALLEL 5

template<class T>
```

```
class Array {

 public:
 int size;
 T* arr;

 //Parameterized Constructer
 Array(int n, bool init_rand=true){
   size = n;
   arr = (T*)malloc(n * sizeof(T));
   if(init_rand==true){
     initialize_random();
   }
 }

 //Copy Constructor
 Array(const Array &arg){
     size = arg.size;
     arr = (T*)malloc(size * sizeof(T));
     for(int i=0;i<size;i++){
         arr[i] = arg.arr[i];
     }
 }

 void initialize_random() {
   for(int i=0; i<size; i++) {
     arr[i] = rand()%100 * 1.43;
   }
 }

 //Overloading Insertion Operator
 template <typename U>
 friend ostream& operator<<(ostream& os, const Array<U> &obj);


 void merge(int l, int mid, int h, T* temp) {
```

```
  int cur = l;
  int i = l;
  int j = mid+1;

  while(i <= mid && j <= h){
    if(arr[i] < arr[j]) {
      temp[cur] = arr[i];
      cur++;
      i++;
    }
    else {
      temp[cur] = arr[j];
      cur++;
      j++;
    }
  }

  if(i <= mid) {
    while(i <= mid) {
      temp[cur] = arr[i];
      i++;
      cur++;
    }
  }
  else if(j <= h) {
    while(j <= h) {
      temp[cur] = arr[j];
      j++;
      cur++;
    }
  }

  for(int arrIndex=l;arrIndex<=h;arrIndex++){
      arr[arrIndex] = temp[arrIndex];
  }

} // end of merge()
```

```
void mergeSortParallel(int l, int h, T* tmp)
{
  if (h <= l)
    return;

  int mid = l + (h-l)/2;
  int mid2 = mid+1;

  #pragma omp task firstprivate (l, mid, tmp)
  mergeSortParallel(l, mid, tmp);

  #pragma omp task firstprivate (mid2, h, tmp)
  mergeSortParallel(mid2, h, tmp);

  #pragma omp taskwait
  merge(l,mid, h, tmp);

}


void mergeSortParallel(){
  T* tmp = (T*)malloc(size * sizeof(T));
  omp_set_num_threads(2);
  #pragma omp parallel
  {
    #pragma omp single
    mergeSortParallel(0, size-1, tmp);
  }
}

void mergeSortSerial(int l, int h, T* tmp)
{
  if (h <= l)
    return;
  int mid = l + (h-l)/2;
  mergeSortSerial(l, mid, tmp);
```

```
  mergeSortSerial(mid+1, h, tmp);
  merge(l,mid, h, tmp);

}



void mergeSortSerial(){
  T* tmp = (T*)malloc(size * sizeof(T));
  mergeSortSerial(0, size-1, tmp);
}

void swap(T *num1, T *num2) {
    T temp = *num1;
    *num1 = *num2;
    *num2 = temp;
}

void bubbleSortParallel(){
    for(int i=size-1; i>=0; i--) {
        #pragma omp parallel for
        for(int j=0; j<i; j++) {
            if(arr[j] > arr[j+1])
              swap(&arr[j], &arr[j+1]);
        }
    }

}

void oddEvenSortSerial(){

  for(int i=0;i<size;i++) {


    // Perform Bubble sort on odd indexed element

    for (int j=1; j<=size-2; j=j+2){
      if (arr[j] > arr[j+1]) {
```

```
        swap(&arr[j], &arr[j+1]);
      }
    }

    // Perform Bubble sort on even indexed element
    for (int j=0; j<=size-2; j=j+2) {
      if (arr[j] > arr[j+1]) {
        swap(&arr[j], &arr[j+1]);
      }
    }
  }
}

void oddEvenSortParallel(){
  int i,j;
  for(i=0;i<size;i++){
    if(i%2==0){
      #pragma omp parallel for private(j) shared(arr)
      for(j=0;j<size-1;j+=2){
        if(arr[j]> arr[j+1]){
          swap(&arr[j],&arr[j+1]);
        }
      }
    }
    else{
      #pragma omp parallel for private(j) shared(arr)
      for(j=1;j<size-1;j+=2){
        if(arr[j]> arr[j+1]){
          swap(&arr[j],&arr[j+1]);
        }
      }
    }
  }
}


void bubbleSortSerial(){
```

```
    for(int i=size-1; i>=0; i--) {
        for(int j=0; j<i; j++) {
            if(arr[j]>arr[j+1]){
                swap(&arr[j],&arr[j+1]);
            }
        }
    }
}

double timeit(int choice){

    double start, end;

    switch(choice){
        case MERGE_SORT_PARALLEL:
            start = omp_get_wtime();
            mergeSortParallel();
            end = omp_get_wtime();
            break;
        case MERGE_SORT_SERIAL:
            start = omp_get_wtime();
            mergeSortSerial();
            end = omp_get_wtime();
            break;
        case BUBBLE_SORT_PARALLEL:
            start = omp_get_wtime();
            bubbleSortParallel();
            end = omp_get_wtime();
            break;
        case BUBBLE_SORT_SERIAL:
            start = omp_get_wtime();
            bubbleSortSerial();
            end = omp_get_wtime();
            break;
        case ODDEVEN_SORT_SERIAL:
            start = omp_get_wtime();
            oddEvenSortSerial();
```

```
            end = omp_get_wtime();
            break;
        case ODDEVEN_SORT_PARALLEL:
            start = omp_get_wtime();
            oddEvenSortParallel();
            end = omp_get_wtime();
            break;
        default:
            return 0;

    }

    return (end-start);

 }

};


int timeAlgorithms(int n){

   Array <float> obj(n), a=obj, b=obj, c=obj, d=obj, e=obj, f=obj;

   cout<<"\nNUMBER_OF_ELEMENTS = "<<n;

   cout<<"\nMERGE_SORT_SERIAL Time = "      <<a.timeit(MERGE_SORT_SERIAL);
   cout<<"\nMERGE_SORT_PARALLEL Time = "
<<b.timeit(MERGE_SORT_PARALLEL);
   cout<<"\nBUBBLE_SORT_SERIAL Time = "    <<c.timeit(BUBBLE_SORT_SERIAL);
   cout<<"\nBUBBLE_SORT_PARALLEL Time = "
<<d.timeit(BUBBLE_SORT_PARALLEL);
   cout<<"\nODDEVEN_SORT_SERIAL Time = "
<<e.timeit(ODDEVEN_SORT_SERIAL);
   cout<<"\nODDEVEN_SORT_PARALLEL Time = "
<<f.timeit(ODDEVEN_SORT_PARALLEL);
}
```

```
template <typename T>
ostream& operator<<(ostream& os, const Array<T> &obj){
 if(obj.size==0){
   os<<"[ ]";
   return os;
 }
 os<<"[ "<<obj.arr[0];
 for(int i=1; i<obj.size; i++) {
   os<<", "<<obj.arr[i];
 }
 os<<"]";
 return os;
}

int main(){

   Array <float> obj(8), a=obj, b=obj, c=obj, d=obj, e=obj, f=obj;

   cout << "\n\nORIGINAL ARRAY : \n"<< obj;

   a.mergeSortParallel();
   cout << "\n\nMERGE_SORT_PARALLEL : \n"<< a;

   b.mergeSortSerial();
   cout << "\n\nMERGE_SORT_SERIAL : \n"<< b;

   c.bubbleSortParallel();
   cout << "\n\nBUBBLE_SORT_PARALLEL : \n"<< c;

   d.bubbleSortSerial();
   cout << "\n\nBUBBLE_SORT_SERIAL : \n"<< d;

   e.oddEvenSortSerial();
   cout << "\n\nODDEVEN_SORT_SERIAL : \n"<< e;

   f.oddEvenSortParallel();
   cout << "\n\nODDEVEN_SORT_PARALLEL : \n"<< f;
```

```
    cout<<endl<<endl;
    timeAlgorithms(65536);

    return 0;
}
```

Google Colab Notebook Link

https://colab.research.google.com/drive/1jUHs2mDgRH-DCBZohxv245h4HQD1cxOF?usp=sharing

Conclusion

I have successfully parallelized sorting algorithms like bubble sort, merge sort and odd-even sort, learnt how to do so using OpenMP in C++.