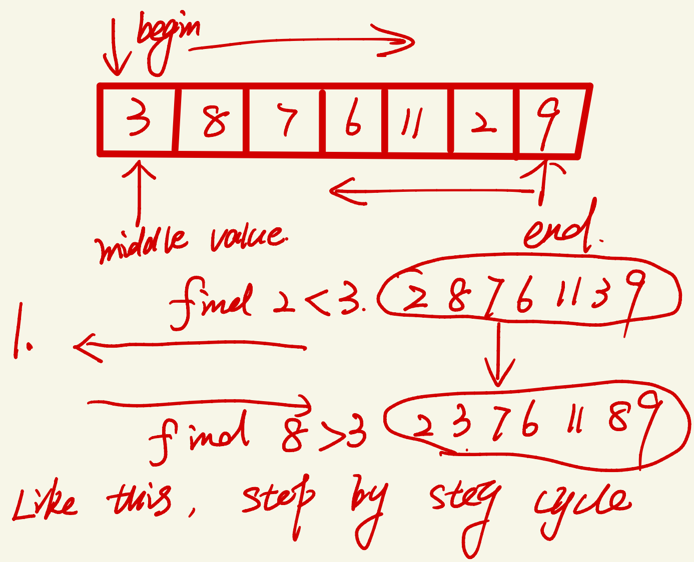
5507 Project1
Ame Liu 22910358

1.Structure

[1] QuickSort

Principle： The main principle of quick sorting is left and right exchange sorting, first, we determine a middle value, the array is divided into left and right two parts, from right to left to find smaller than the middle value, from left to right to find larger than the middle value, exchange each other's position, and so on, and so on, finally get a sequence that the middle value to the left is all smaller than the middle value, the middle value to the right is larger than the middle value. At this point, the left and right sides can be seen as two new arrays, continue to use the above principle until the end of the loop.

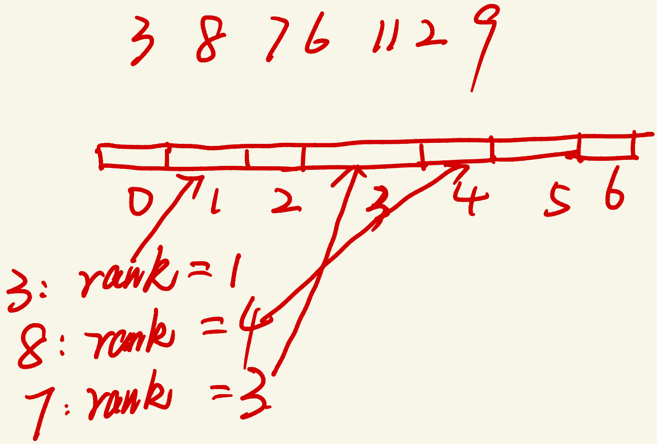
Diagram:



[2]EnumerationSort

Principle：Enumeration sorting is to compare the size of all elements with all other elements and find out how many elements are smaller than themselves, so as to obtain the position (rank) of the elements. The position of each element is unique.

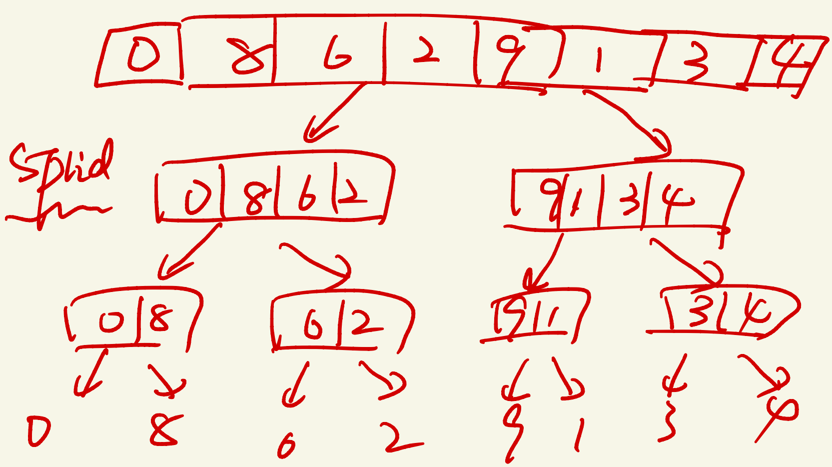
Diagram:



[3]MergeSort

Principle: The array is split into individual elements, so that relative to itself is ordered, two adjacent elements compare size, so that the loop, and finally merged into a whole orderly sequence.

Diagram:



2.Requirement

(1).Random generation of an array with double-precision floating-point numbers;

```
std::random_device rd;
std::mt19937 gen(rd());
std::uniform_real_distribution<>random(1.0,100000.0);
for(int i=0;i<n;i++) {
    a[i]=random(gen);
    //std::cout<<a[i]<<' ';
    }
```

All of my code uses the same random function as shown in the figure to display double random numbers.

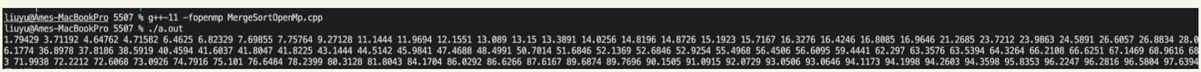
```
liuyu@Ames-MacBookPro 5507 % ./a.out
3.14555 1.57566 2.67936 4.57623 3.99636 2.41742 8.83001 7.27742 3.27268 7.91091
```

(2).A serial solution for each sorting algorithm to sort the generated array;
[1] QuickSort

```
liuyu@Ames-MacBookPro 5507 % g++-11 -fopenmp QuickSortOpenMp2.cpp
liuyu@Ames-MacBookPro 5507 % ./a.out
1.54195 3.08108 3.417 4.60761 5.46417 7.97852 8.7639 8.84981 9.66999 9.84632
```

[2]EnumerationSort

[3]MergeSort



All code arrangement functions are complete.

(3) A parallel solution for each sorting algorithm to sort the array with OpenMP on CPU;

The parallel solution for each arrangement algorithm will be described in detail in the Report section.

3.Report

a. the pseudocodes of your serial and parallel solutions.

[1] QuickSort

serial codes:

void QuickSort; (double *arr , int low, int high) *;*

To define and this function, we need two auxiliary pointers low and high,

double temp = arr[low] The base value is temp

int i = low Used to traverse from left to right

int j = high Used to traverse from right to left

while(i<j && arr[j] > temp); j—;arr[i] = arr[j]. When the number on the right is greater than the number of baselines, skip and continue looking to the left, fill in the number of reference elements less than or equal to the right into the corresponding position on the right.

while(i<j && arr[i] <= temp). i++;

arr[j] = arr[i]

arr[i] = temp

QuickSort(arr,low,i-1); When the number on the left is less than or equal to the base number, skip and continue looking to the right, fill in the number of elements on the left that are larger than the baseline element in the corresponding position on the left.

QuickSort(arr,i+1,high)

Once all arranged, the left and right of temp can be thought of as two new sequences.

And so on.

parallel codes:

My parallel idea is to divide the original array into two and sort them quickly respectively. After the quick sorting, the merging sorting of the two arrays can make the algorithm with the lowest time complexity and the most efficient algorithm.

#pragma omp parallel sections

{

#pragma omp section Quick sorting of two arrays d,e at the same time.

MergeSort (d,0,m-1);

#pragma omp section.

MergeSort (e,0,m-1);

}

while(left<m && right<m). When the left and right arrays are not finished.

{

if(d[left]<=e[right]). if the small on the left row is left, the pointer moves one unit to the right. If the small on the right row is right, the pointer moves one unit to the right.

b[k++]=d[left++];

else

b[k++]=e[right++];

}

while(left<m). If the left row is finished, copy all the rest on the right. If the right row is finished, copy all the rest on the left.

{

b[k++]=d[left++];

[2]EnumerationSort

serial codes:

void enumSort (double *a, double *b, int n) Define two arrays, one is itself and the other is responsible for passing.

Enter the loop

```
for (int i = 0; i < n; i++)
  {
     int k = 0;
     for (int j = 0; j < n; j++)
     {
        if (a[i] > a[j])
           k++;
     }
     while (b[k] != 0)
        k++;
     b[k] = a[i];
  }
```

Create two arrays. Array A is the sequence to be sorted, and array B is used to store the sorted sequence.

Two for loops are used to compare enumeration sorting. The external for loop i = 0... i + +. Each time, we will select a value to compare with each number in the sequence. The purpose of this loop is to change this value.

The purpose of the second for loop j = 0... j + + is to select a value to compare with each number in the sequence, which is "each number"

After executing the for loop twice, we can compare each number in the sequence with these numbers in the sequence. If a [i] > a [j] performs the k + + operation, it means to move the position of a [i] to the right. For example, if the leftmost number a [0] in the sequence is larger than the five numbers in the sequence, execute k + + five times.

Start with the first number [i] and compare with the numbers in the sequence. If [i] is smaller than the numbers in the sequence, put [i] in the position of the corresponding array b,

The meaning of while loop is to judge whether the position of array b has a value at this time. If so (for example, if two numbers in array a are equal, their k + + times are also equal. At this time, if you do not use while loop judgment, the old original value will be replaced by the new value. For example, 33325 may be arranged as 23 empty 5).

Finally, b [k] = a [i] is to put the compared number in the position that should be placed in array b.

parallel codes:

```
void paEnumSort(double *a, double *b, int n)
{

#pragma omp parallel.
 {
    #pragma omp for
   for (int i = 0; i < n; i++)
   {
      int k = 0;
      for (int j = 0; j < n; j++)
      {
         if (a[i] > a[j])
            k++;
      }
      while (b[k] != 0)
         k++;
      b[k] = a[i];
   }
 }

}
```

[3]MergeSort

serial codes:

```
void MergeSort (double arr [], int low,int high)        //incise
if(low>=high) { return; } // Stop                          when all sequence lengths are 1.
   int mid =  low + (high - low)/2;                        //Gets the element in the middle of the sequence.

   MergeSort(arr,low,mid);                                // Recursive to the left half.

   MergeSort(arr,mid+1,high);                            //  Recursive to the right half.



   Merge(arr,low,mid,high);                             // merge.
```

```
void Merge(double arr[],int low,int mid,int high){
                                                    //Low is the first element
of the 1st ordered zone, i points to the 1st element, and mid is the last element of the 1st
ordered zone.
   int i=low,j=mid+1,k=0;                           //Mid + 1 is the first
element of the second ordered region, and j points to the first element.
   double *temp=new double[hig h-low+1];            //Temp array temporarily
stores the merged ordered sequence.
   while(i<=mid&&j<=high){
      if(arr[i]<=arr[j])                            //Which side is smaller, which
side is placed directly into the temporarily.
         temp[k++]=arr[i++];
      else
         temp[k++]=arr[j++];
   }
   while(i<=mid)                                    //After the comparison, if there is an
array left that indicates that the rest is already in order of sequence, copy directly.
      temp[k++]=arr[i++];
   while(j<=high)
      temp[k++]=arr[j++];
   for(i=low,k=0;i<=high;i++,k++)                   //Save the sequence back to the
range of low to high in arr.
   arr[i]=temp[k];
   delete []temp;                                   //Free up memory.
}
```

parallel codes:

The parallel idea of merge sort is similar to that of quicksort. I choose to divide the array into two, and then run merge sort twice in parallel, and then sort and merge the two arrays after sorting,

```
#pragma omp parallel sections
{
 #pragma omp section
 MergeSort (d,0,m-1);      //MergeSorting of two arrays d,e at the same time.
 #pragma omp section
 MergeSort (e,0,m-1);
}
```

```
while(left<m && right<m)      //When the left and right arrays are not finished.
{
    if(d[left]<=e[right])
    b[k++]=d[left++];       // if the small on the left row is left, the pointer moves one unit to the right.
If the small on the right row is right, the pointer moves one unit to the right.
    else
    b[k++]=e[right++];
}
while(left<m)
{
    b[k++]=d[left++];

}
while(right<m)
{
    b[k++]=e[right++];
}
```

b. the description of your experimental environment (such as CPU, RAM, and operating system)

compiler: gcc-11

```
liuyu@Ames-MacBookPro ~ % which gcc
gcc: aliased to gcc-11
```

CPU:

处理器 2.6 GHz 六核Intel Core i7

RAM:16GB

BANK 0/ChannelA-DIMM0	8 GB	DDR4	2667 MHz
BANK 2/ChannelB-DIMM0	8 GB	DDR4	2667 MHz

OS:

macOS Big Sur
版本 11.5.2

MacBook Pro (16-inch, 2019)

c&d a. how you compile your source code, and the experimental results with speedup analysis of your parallel implementations over the serial ones.

Statement: the parallel and serial sequences used in all experiments are the same random sequences, and the computer runs different kinds of codes for the same time, which has reduced the contingency and unreliability as much as possible.

n is set to 100’000 random numbers.

[1] QuickSort

```
int main()
{  int k=0;
   int i=0;
   int n=100000;
   int left=0;
   int right=0;
   int m=n/2;
   double a[n];
   double b[n];
   double c[n];
   double e[m];
   double d[m];
   std::random_device rd;
   std::mt19937 gen(rd());
   std::uniform_real_distribution<>random(1.0,100000.0);
```

```
s1 = omp_get_wtime();
quickSort(a,0,n-1);
//for (int i = 0;i<n;i++)
//    {
//    cout<<a[i]<<' ';
//        }
e1 = omp_get_wtime();

cout << "Serial operation time:" << e1 - s1 << endl;
cout << endl;

s2 = omp_get_wtime();
#pragma omp parallel sections
{
 #pragma omp section
 quickSort(d,0,m-1);      //quickSorting of two arrays d,e at the same time.
 #pragma omp section
 quickSort(e,0,m-1);
}
while(left<m && right<m)      //When the left and right arrays are not finished.
{
    if(d[left]<=e[right])
    b[k++]=d[left++];
    else
    b[k++]=e[right++];          // if the small on the left row is left, the po
}
while(left<m)
{
    b[k++]=d[left++];

}                               //If the left row is finished, copy all the rest
while(right<m)
{
    b[k++]=e[right++];
}

e2 = omp_get_wtime();
cout << "Parallel operation time:" << e2 - s2 << endl;
cout << endl;

return 0;
}
```

```
liuyu@Ames-MacBookPro 5507 % g++-11 -fopenmp QuickSortOpenMp2.cpp
liuyu@Ames-MacBookPro 5507 % ./a.out
Serial operation time:0.0155

Parallel operation time:0.008414

liuyu@Ames-MacBookPro 5507 % ./a.out
Serial operation time:0.015897

Parallel operation time:0.008668

liuyu@Ames-MacBookPro 5507 % ./a.out
Serial operation time:0.016199

Parallel operation time:0.0085

liuyu@Ames-MacBookPro 5507 % ./a.out
Serial operation time:0.015743

Parallel operation time:0.008568

liuyu@Ames-MacBookPro 5507 % ./a.out
Serial operation time:0.01579

Parallel operation time:0.008609
```

I also explored how to perform multiple quick sorting at the same time, and the efficiency is greatly improved by using OpenMP.

```
   s1 = omp_get_wtime();
   QuickSort(a1,0,n-1); //(a,0,n-1)
   QuickSort(a2,0,n-1);
   QuickSort(a3,0,n-1);
   QuickSort(a4,0,n-1);
   e1 = omp_get_wtime();
   cout << "Serial operation time:" << e1 - s1 << endl;
    //for (int i = 0;i<10000;i++)
       {
       //cout<<a1[i]<<' ';
       }
   cout << endl;
   s2 = omp_get_wtime();
#pragma omp parallel sections
{
   #pragma omp section
    {
  QuickSort(b1,0,n-1); //(a,0,n-1)
    }
   #pragma omp section
   {
   QuickSort(b2,0,n-1);
   }
   #pragma omp section
   {
   QuickSort(b3,0,n-1);
   }
   #pragma omp section
   {
   QuickSort(b4,0,n-1);
   }


}
   e2 = omp_get_wtime();
   cout << "Parallel operation time:" << e2 - s2 << endl;
```

```
liuyu@Ames-MacBookPro 5507 % g++-11 -fopenmp QuickSortOpenMp1.cpp
liuyu@Ames-MacBookPro 5507 % ./a.out
Serial operation time:0.053777

Parallel operation time:0.013002

liuyu@Ames-MacBookPro 5507 % ./a.out
Serial operation time:0.053715

Parallel operation time:0.012753

liuyu@Ames-MacBookPro 5507 % ./a.out
Serial operation time:0.05167

Parallel operation time:0.012951

liuyu@Ames-MacBookPro 5507 % ./a.out
Serial operation time:0.051198

Parallel operation time:0.012969

liuyu@Ames-MacBookPro 5507 % ./a.out
Serial operation time:0.055104

Parallel operation time:0.012121

liuyu@Ames-MacBookPro 5507 % ./a.out
Serial operation time:0.054489

Parallel operation time:0.013448
```

[2]EnumerationSort

```
}
  s1 = omp_get_wtime();
  enumSort(a1, b1, n);
  e1 = omp_get_wtime();
  cout << "Serial operation time:" << e1 - s1 << endl;

  s2 = omp_get_wtime();
  paEnumSort(a2, b2, n);
  e2 = omp_get_wtime();
  cout << "Parallel operation time:" << e2 - s2 << endl;
  cout << endl;
```

```
liuyu@Ames-MacBookPro 5507 % ./a.out
liuyu@Ames-MacBookPro 5507 % g++-11 -fopenmp EnumerationSortOpenMp.cpp
liuyu@Ames-MacBookPro 5507 % ./a.out
Serial operation time:45.8007
Parallel operation time:5.55875

liuyu@Ames-MacBookPro 5507 % ./a.out
Serial operation time:42.3381
Parallel operation time:4.86783

liuyu@Ames-MacBookPro 5507 % ./a.out
Serial operation time:42.559
Parallel operation time:4.85028

liuyu@Ames-MacBookPro 5507 % ./a.out
Serial operation time:42.4556
Parallel operation time:4.90004

liuyu@Ames-MacBookPro 5507 % ./a.out
Serial operation time:42.3383
Parallel operation time:4.88937
```

[3]MergeSort

```
e1 = omp_get_wtime();

cout << "Serial operation time:" << e1 - s1 << endl;
cout << endl;

s2 = omp_get_wtime();
#pragma omp parallel sections
{
 #pragma omp section
 MergeSort (d,0,m-1);       //MergeSorting of two arrays d,e at the same time.
 #pragma omp section
 MergeSort (e,0,m-1);
}
while(left<m && right<m)       //When the left and right arrays are not finished.
{
    if(d[left]<=e[right])
    b[k++]=d[left++];        // if the small on the left row is left, the pointer moves one unit
    else
    b[k++]=e[right++];
}
while(left<m)
{
    b[k++]=d[left++];       //If the left row is finished, copy all the rest on the right. If the

}
while(right<m)
{
    b[k++]=e[right++];
}

e2 = omp_get_wtime();
cout << "Parallel operation time:" << e2 - s2 << endl;
cout << endl;

return 0;
}
```

```
liuyu@Ames-MacBookPro 5507 % g++-11 -fopenmp MergeSortOpenMp.cpp
liuyu@Ames-MacBookPro 5507 % ./a.out
Serial operation time:0.029143

Parallel operation time:0.015022

liuyu@Ames-MacBookPro 5507 % ./a.out
Serial operation time:0.034414

Parallel operation time:0.017207

liuyu@Ames-MacBookPro 5507 % ./a.out
Serial operation time:0.034067

Parallel operation time:0.017556

liuyu@Ames-MacBookPro 5507 % ./a.out
Serial operation time:0.034101

Parallel operation time:0.017265

liuyu@Ames-MacBookPro 5507 % ./a.out
Serial operation time:0.034393

Parallel operation time:0.017152
```

Experimental Data

Time	Sort	Types	Sequences	Rating
0.0155	QuickSort	Serial	1	1.8452381
0.0159	QuickSort	Serial	2	1.82758621
0.0162	QuickSort	Serial	3	1.90588235
0.0157	QuickSort	Serial	4	1.8045977
0.0158	QuickSort	Serial	5	1.8372093
0.0084	QuickSort	Parallel	1	
0.0087	QuickSort	Parallel	2	
0.0085	QuickSort	Parallel	3	
0.0087	QuickSort	Parallel	4	
0.0086	QuickSort	Parallel	5	
45.8	EnumerationSort	Serial	1	8.23889189
42.338	EnumerationSort	Serial	2	8.69720624
42.559	EnumerationSort	Serial	3	8.77505155
42.456	EnumerationSort	Serial	4	8.6644898
42.3383	EnumerationSort	Serial	5	8.65991
5.559	EnumerationSort	Parallel	1	
4.868	EnumerationSort	Parallel	2	
4.85	EnumerationSort	Parallel	3	
4.9	EnumerationSort	Parallel	4	
4.889	EnumerationSort	Parallel	5	
0.0291	MergeSort	Serial	1	1.94
0.0344	MergeSort	Serial	2	2.02352941
0.034	MergeSort	Serial	3	1.93181818
0.0341	MergeSort	Serial	4	1.97109827
0.0344	MergeSort	Serial	5	2
0.015	MergeSort	Parallel	1	
0.017	MergeSort	Parallel	2	
0.0176	MergeSort	Parallel	3	
0.0173	MergeSort	Parallel	4	
0.0172	MergeSort	Parallel	5	

To sum up:
The improvement ratio of parallel quicksort and parallel mergesort is about 100%, doubling the speed; The parallel enumeration sort is improved by about 750%. We find that concurrent programs greatly improve the enumeration sort of serial programs; Finally, on the whole, quick sort is the best sort;