CITS5507 Project1 Report

Hanlin Zhang

22541459

Sep 2 2021

Content

1. Experimental design

1. Pseudocode of serial enum sort algorithm

```
enum_sort(array, array_new, array_length)
for i <– 0 to array_length
    count_smaller <– 0
    for j <– 0 to array_length
        if array[j] < array[i]
            then count_smaller += 1
    new_array[count_smaller] <– array[i]

for i <– 0 to array_length
    if array_new[i] == 0
        then array_new[i] <– array_new[i–1]

return array_new
```

For this algorithm, we need to pick each element from the array, then compare other elements in that array, to see how many elements is larger than it (suppose number is x). Place current selected element and place it at position x.

But this algorithm will meet a problem, if there are two or more elements have same value, this might occurs the different elements with same value place at the same position. To solve this, loop the whole new array, if empty cell has been found, copy the last cell value to it.

2. Pseudocode of merge quick sort algorithm

```
merge_sort(array, start, end)
if start < end
    then half <– (start + end) / 2
        merge_sort(array, start, half)
        merge_sort(array, half + 1, end)
        merge_sort_looping(array, start, end)

merge_sort_looping(array, start, end)
half <– (start + end) / 2
pointer <– 0
create array temp[0, end – start + 1]
left <– start
right <– half + 1
```

```
while (left <= half) and (right <= end)
        do if array[left] < array[right]
                then temp[pointer++] <– array[left++]
          else do temp[pointer++] <– array[right++]

while(left <= half)
    do temp[pointer++] <– array[left++]

while(right <= end)
    do temp[pointer++] <– array[right++]

for (i <– start to end)
    array[i] <– temp[i – start]
```

Generate a new array to get the result after sorting.split the array to be two parts which called A and B. Split A and B separately by using recursion.repeat the step, until only 1 element inside, then compare each. Put the smallest in two elements at the left, the biggest on the right. After repeat the step above, we get multiple arrays which contains 2 elements inside. Then re-compare each array from left to right, sort them and merge two arrays times and times. Finally, we will get a sorted array.

3. Pseudocode of serial quick sort algorithm

```
quicksort(array, left, right)
    if(left < right)
        then pointer <– quick_sort_looping(array, left, right)
             quicksort(array, left, pointer – 1)
             quicksort(array, pointer + 1, right)

quick_sort_looping(array, left, right)
    if(left > right)
        then return 0
    temp <– array[left]
    while(left < right)
        while(left < right && array[right] >= temp)
            then right––
        array[left] <– array[right]
        while(left < right && array[left] <= temp)
            then left++
        array[right] <– array[left]

    array[left] <– temp
    return left
```

Loop the array. In general, quick sort pick the first element, then compare each element with it. If element less than it, put at the left hand side. Otherwise put at the right hand side. Repeat the step above to sort the left array and right array by using recursion. Finally, we will get a sorted array.

4. Pseudocode of parallel enumerate sort algorithm using omp

```
omp_enum(array, array_new, value){

    int count_smaller
    #omp parallel block

            #omp for avoid data race
            for i <– 0 to value
                count_smaller <– 0;
                for j <– 0 to value
                    if array[j] < array[i]
                        then do count_smaller + 1

                array_new[count_smaller] <– array[i]
            #omp for end


            #omp for
            for i <– 0 to value
                if array_new[i] = 0
                    array_new[i] <– array_new[i – 1]
            #omp for end

    #omp parallel block end
```

This part is really similar with the pseudocode of serial enumerate sort algorithm, just add several omp blocks inside. These blocks can increasing the speed of the algorithm by using multiple thread to run the for loop. One important thing is, this might cause data race, so we have to avoid it, make sure each time only one thread read and re-write the variable.

5. Pseudocode of parallel merge sort algorithm using omp

```
omp_merge(array, start, end)
    if start < end
        then do:
        if (end – start) > 100000
            then do:
            middle = (start + end) / 2;
            #omp parallel block

                #single thread nowait

                    task firstprivate(array, start, middle)
                    omp_merge(array, start, middle);    //array1 (left)
                    task firstprivate(array, middle, end)
                    omp_merge(array, middle + 1, end);  //array2 (right)
                #single thread nowait end

            //wait for the task finished, then sort the array
            taskwait
            then do merge_sort_looping(array, start, end)

             #omp parallel block end
        else
             // if the distance between start and end is small using serial
            merge_sort(array, start, end);
```

In this algorithm, we will split the whole task to be two parts (tasks), then solve them separately by split them again and again. Then pass the tasks result to loop them (compare each) and sort them.

Here, we might find an interesting thing, the running time if we only use parallel way will be multi times slower than the serial solution (around three times). To improve it, we can let the algorithm combine the advantages of serial solution and parallel solution. After testing, 100000 might be a good critical value. It means when the array size is ≤ 100000, we will start to use serial solution. Otherwise, using parallel solution.

6. Pseudocode of parallel quick sort algorithm using omp

```
omp_quick(array, left, right){
    pointer = quick_sort_looping(array, left, right – 1)
    #parallel block start with 2 threads

        sections block

            section block1
            quicksort(array, 0, pointer – 1)

            section block2
            quicksort(array, pointer, right – 1)

        sections block end

    #parallel block end
```

This parallel solution is easier to understand and implement. Two threads execute the two sections separately. More info, in the serial solution of quick sort.

7. What platform this program testing

Mac M1 (gcc version:10.0)
&
virtual machine environment on Mac Intel (Intel(R) Core(TM) i5-8257U CPU @1.40GHz)

8. How to run this program

———————————————— Serial —————————————————

To compile the program, use:

On Mac M1 (gcc-x depands on the version of your gcc):

```
gcc-10 -fopenmp project_file.c -o project_file.out
```

On virtual machine environment with Mac Intel

```
gcc -fopenmp project_file.c -o project_file.out
```

To run the serial enumerate, merge or quick sort algorithm separately, we can use command below (./FILE_NAME ARRAY_SIZE ALGORITHM):

```
./project_file.out 50000 enum
```

```
./project_file.out 10000000 merge
```

```
./project_file.out 10000000 quick
```

Important! Do not set a large array size when you are using enumerate sort algorithm, since the running speed will be too slow. For an array with 100000 size large might need nearly one minute to finish sorting.

To view the output, add command “print_array(array_quick, array_length)” in function start_quick, start_enum or start_merge, after they call quick, enum or merge sort algorithm function. Did not print the result due to the large number testing issue.

If you want to run them together, you can use:

```
./project_file.out 100000 all
```

All algorithm will using array size 100000 in the command above, including enumerate sort!

—————————————— Parallel ——————————————

To run the parallel enumerate, merge or quick sort algorithm separately, we can use command below (./FILE_NAME ARRAY_SIZE ALGORITHM -omp):

```
./project_file.out 50000 enum -omp
```

```
./project_file.out 10000000 merge -omp
```

```
./project_file.out 10000000 quick -omp
```

Important! Do not set a large array size when you are using enumerate sort algorithm, since the running speed will be too slow. For an array with 100000 size large might need nearly one minute to finish sorting. For quick sort using omp, need array size larger than 40000 can show the improve. Then for merge sort using omp, need array size larger than 100000 can show the improve.

To view the output, add command “print_array(array_quick, array_length)” in function start_omp_quick, start_omp_enum or start_omp_merge, after they call omp_quick, omp_enum or omp_merge sort algorithm function. Did not print the result due to the large number testing issue.

Using compare_result(array_1, array_2, array_3, array_length) to compare the result.

If you want to run them together, you can use:

```
./project_file.out 100000 all -omp
```

All algorithm will using array size 100000 in the command above, including enumerate sort!

For the parallel solution, it will also run the serial solution together and compare how many times parallel solution faster than serial solution.

2. Experimental result analysis

Array length: 200,000 (200 k) for both Group A and B

Group A: On Mac platform (M1 chip)

Group A	enum	enum omp	merge	merge omp	quick	quick omp
test 1	195.618187	36.047353	0.053235	0.026388	0.027241	0.023274
test 2	202.589828	38.493807	0.055101	0.026302	0.027787	0.023673
test 3	202.742286	40.756201	0.054948	0.029628	0.027848	0.024222
test 4	204.08035	42.615962	0.054433	0.03214	0.028338	0.024716
test 5	202.968463	39.513847	0.055592	0.025841	0.028127	0.025394
Average	201.599823	39.485434	0.054662	0.028060	0.027868	0.024256
Average Rate	**0.195860**		**0.513335**		**0.870376**	

Group B: On virtual machine environment with using Mac Intel (Intel(R) Core(TM) i5-8257U CPU @ 1.40GHz)

Group B	enum	enum omp	merge	merge omp	quick	quick omp
test 1	193.655498	63.127677	0.044642	0.024274	0.029665	0.025374
test 2	243.996778	80.445499	0.044882	0.025364	0.033567	0.025946
test 3	221.277733	62.083233	0.046585	0.025891	0.037324	0.030555
test 4	191.056434	57.982002	0.044739	0.024074	0.030514	0.023346
test 5	200.12945	62.304247	0.04335	0.024716	0.030454	0.024355
Average	210.023179	65.188532	0.044840	0.024864	0.032305	0.025915
Average Rate	**0.310387**		**0.554505**		**0.802209**	

To control variable, in each group, each algorithm in the same test using the same array to sort. For instance, in Group A test 1, all algorithm using the same array whatever using OpenMP or not.

The variable “Average Rate” calculate by Average time / Average time using OpenMP.

Here, we can increase the number of array length to see quick sort and merge sort which is faster:

Array length: 5,000,000 (5 m) for both Group C and D

Group C: On Mac platform (M1 chip)

Group C	enum	enum omp	merge	merge omp	quick	quick omp
test 1	–	–	1.521067	0.703963	0.835471	0.432534
test 2	–	–	1.512747	0.713458	0.840361	0.432362
test 3	–	–	1.489361	0.700635	0.834955	0.424965
test 4	–	–	1.506970	0.693225	0.831194	0.421740
test 5	–	–	1.498213	0.687684	0.828703	0.420106
Average	–	–	1.505672	0.699793	0.834137	0.426341
Average Rate	–		**0.464771**		**0.511117**	

Group D: On virtual machine environment with using Mac Intel (Intel(R) Core(TM) i5-8257U CPU @ 1.40GHz)

Group D	enum	enum omp	merge	merge omp	quick	quick omp
test 1	–	–	1.392141	0.751875	0.946755	0.720371
test 2	–	–	1.318632	0.741339	0.924295	0.887917
test 3	–	–	1.326845	0.736106	0.925113	0.922990
test 4	–	–	1.313538	0.750556	0.924147	0.929483
test 5	–	–	1.319364	0.719340	0.935519	0.883676
Average	–	–	1.334104	0.739843	0.931166	0.868887
Average Rate	–		**0.554562**		**0.933118**	

Array length: 20,000,000 (20 m) for both Group C and D

Group C: On Mac platform (M1 chip)

Group C	enum	enum omp	merge	merge omp	quick	quick omp
test 1	–	–	6.727785	3.078425	3.612686	1.850466
test 2	–	–	6.704744	2.969015	3.609417	1.814439
test 3	–	–	6.690637	3.049508	3.629180	1.874938
test 4	–	–	6.768092	3.104300	3.542231	1.811846
test 5	–	–	6.717507	3.072843	3.594551	1.829782
Average	–	–	6.721753	3.054818	3.597613	1.836294
Average Rate	–		0.454467		0.510420	

Group D: On virtual machine environment with using Mac Intel (Intel(R) Core(TM) i5-8257U CPU @ 1.40GHz)

Group D	enum	enum omp	merge	merge omp	quick	quick omp
test 1	–	–	6.043571	3.289074	3.999978	3.387359
test 2	–	–	5.718616	3.122319	3.940162	3.556851
test 3	–	–	5.798393	3.086951	3.916789	2.596928
test 4	–	–	5.750836	3.076208	3.875419	3.319530
test 5	–	–	5.736199	3.113520	3.072843	2.429681
Average	–	–	5.809523	3.137614	3.761038	3.058070
Average Rate	–		0.540081		0.813092	

3. Explanations of the results

The performance in Group A and B are similar, different due to the different test environment / platform. Now we use Group which using Mac M1 platform to explain, due to I mainly use and programming on this platform (another group is test only).

For the enum sort algorithm, after we improve the enum parallel solution, the enum parallel solution is 5 times faster than enum serial solution. Even only using 200000 array length to test enum sort algorithm, it still spend 3.5 min / 0.7 min for enum serial / enum parallel, so it is hard to discover the result of enum sort algorithm with larger array length.

For the merge sort algorithm, the runtime of merge parallel solution is nearly 40% - 50% shorter than merge serial solution. Merge sort parallel solution does not get influence a lot by changing the array size, due to we make an improvement in the original parallel merge sort algorithm (use both parallel and serial). When array length ≤ 100000, use serial. Otherwise, use parallel.

For the quick sort algorithm, the runtime of quick parallel solution is around 15% - 50% shorter than quick serial solution. The parallel solution will performance better if the array has a larger array length.

In general, the parallel solution of quick sort algorithm is the fastest in all algorithms. Next is merge sort parallel solution. There is no doubt whatever enumerate serial or parallel solution is the slowest in all algorithms, since enumerate algorithm is using time complexity of O(n ^ 2), much larger than others.