HW1 Report

Shengyuan Wang (09/26/2022)

COMP 445

Profiling Sequential Version of the Code

I create a shell script file (`profile_seq.sh`) to run the sequential version of the code automatically. In the script file, I use a for loop to set different problem size from 2^20 to 2^30. In each loop, I use the command below to make a table.

```
./countSort_seq -n $problem_size -e
```

```
#!/bin/bash\
# Name: profile_seq.sh
# Author: Shengyuan Wang
printf "problem_size \t input \t counts \t output \n"
for problem_size in 1048576 2097152 4194304 8388608 16777216 33664432 67108864 134217728 268435456 536870912 1073741824
do
    printf "$problem_size \t"
    command="./countSort_seq -n $problem_size -e"
    $command
done
printf "\n"
```

And I create a tsv file (`profile_seq.tsv`) to store the table output from the shell script by running the code below.

```
bash profile_seq.sh > profile_seq.tsv
```

This is the table created by `profile_seq.sh`.

```
problem_size   input   counts    output
1048576   0.021218  0.008054  0.007540
2097152   0.029506  0.011185  0.010444
4194304   0.041667  0.015781  0.014792
8388608   0.061725  0.022982  0.021383
16777216  0.099042  0.042132  0.041939
33664432  0.198561  0.084535  0.084075
67108864  0.398626  0.169057  0.167617
134217728   0.795822  0.337028  0.335250
268435456   1.590190  0.696946  0.701125
536870912   3.176097  1.450171  1.378579
1073741824  6.351734  2.846343  2.817639
```

From the table above, we can find that input generation is the most time-consuming part of the program. And the counting part and output part consumes about the same time.

When we have a look at the code in input generation part of the sequential version, we can find that it generate the random input string by creating single char character at one time, which is the reason why it is time consuming up to 6 seconds. For counting part, I notice that in the sequential version, it se for loop to count the how many times each ascii character occurs. My decision to speed up this part is to use leapfrog method to parallel the for loop. As for the output part, it also appears the nested for loop, which is a part we can use parallel programming to improve the program performance.

Methodology Used For Parallelize

I will illustrate my methodology in this section from three parts: random word generation, count generation, and sorted word generation.

Random Word Generation

Instead of `rand_r` function used in sequential version, which is not thread safe, I use `Trng` library to generate random numbers in uniform distributions. And I create a generator to repeatedly feed the generator to a distribution to create the next random number. I use data decomposition algorithm strategy for parallel algorithm strategy pattern; Single Program Multiple Data, Fork-Join, Parallel For loop for the program structure; Shared Data for Data Structure; ThreadPool for Concurrent Execution Patterns in generating random string.

Since we will use the uniform distribution function in `Trng` library, we need to initialize `min` and `max`, which in this program is 32(`ASCII_START`) to 126(`ASCII_END`). Then we use openMP to fork threads to generate the random characters in parallel and splits the work in the for loop using leap frog method and assign random value into string. Since the race condition will not take place here, so we do not need to do reduction. Thus, we have `tid`, `i`, `nextRandVal` here as private variable, and `size`, `nThreads`, `res`, `min`, `max`, `seed` as shared variable here.

```
char* generateRandomString(int size, int nThreads) {
    long unsigned int seed = time(NULL);
    int tmp = 0;
    char *res = (char *)malloc(size + 1);
    unsigned int tid; // thread id when forking threads
    unsigned min = ASCII_START;
    unsigned max = ASCII_END;
    int i;
    char nextRandVal;
    // fork threads to generate random character in parallel
    #pragma omp parallel default(none) \
    private(tid, i, nextRandVal) \
    shared(size, nThreads, res, min, max, seed)
    {
      trng::lcg64 rand;
      trng::uniform_dist<> uniform(min, max);
      rand.seed(seed);
      tid = omp_get_thread_num();
      if (nThreads > 1){
        // thread will be substream tid from numThread separate substreams
        rand.split((unsigned)nThreads, tid);
      }
      // openMP splits the work
      #pragma omp for
      for (i=0; i<size; i++) {
        nextRandVal = uniform(rand);
        res[i] = nextRandVal;
      }
    }
    res[size] = '\0';  //so it is a string we can print when debugging
    return res;
}
```

Count Generation

We use data decomposition algorithm strategy for parallel algorithm strategy pattern; Single Program, Multiple Date, Fork-Join, Parallel For loop for program structure; Shared Data for Data Structure; ThreadPool and Reduction for Concurrent Execution Patterns in generating counts for the random string.

The race condition takes place here. When multi-threads count the characters with the same ascii number, some problems will happen on the update to the variable `counts` without reduction. Thus, we have `N`, `input` as shared variables here, and make reduction action on `counts`.

```
void countEachLetter(int * counts, char * input, int N) {
    // Count occurences of each key, which are characters in this case
    #pragma omp parallel for default(none) \
    shared(N, input)  \
    reduction(+:counts[:COUNTS_SIZE])
    for ( int k = 0; k < N ; ++ k ) {
        counts [ input [ k ] ] += 1;
    }
}
```

Sorted Word Generation

We use data decomposition algorithm strategy for parallel algorithm strategy pattern; Single Program, Multiple Date, Fork-Join, Parallel For loop for program structure; Shared Data for Data Structure; ThreadPool for Concurrent Execution Patterns in creating sorted Array.

Since we want each thread to know what small portion of output it can fill independently of the other threads, we introduce a prefix sum array from the counts array. In the array `prefixArr`, `prefixArr[i]` means how many characters has been counted until the character with ascii number `i` . After completing the prefix sum array, we can set `start`, `end`, `v` as private variable, and `output`, `prefixArr` as shared variable to sort in parallel.

```
void createSortedStr(int * counts, char * output, int N) {
    // prefix sum array
    int prefixArr[COUNTS_SIZE] = {0};

    for ( int i = 0; i < COUNTS_SIZE; i++) {
      if (i == 0) {
        prefixArr[0] = counts[0];
      } else {
        prefixArr[i] = prefixArr[i-1] + counts[i];
      }
    }
    // Construct output array from counts
    char v;
    int start, end;

    #pragma omp parallel for default(none) \
    private(v, start, end)\
    shared(output, prefixArr)
    for ( v = 0; v <= ASCII_END; ++ v ) {
      if (v == 0) {
        start = 0;
      } else {
        start = prefixArr[v-1];
      }
      end = prefixArr[v];
      for ( int k = start; k < end; ++ k ) {
        output[ k ] = v ;
      }
    }
    output[N+1] = '\0';   // so it is a string we could print for debugging
}
```

To compile the code, we can manually type in the following code in terminal

```
g++ -o countSort_omp getCommandLine.o countSort_omp.c -fopenmp -I /usr/local/include/trng -l trng4
```

Or just using makefile to compile the code

```
make
#or
make countSort_omp
```

The lines following are the usage of argument.

```
Usage: ./countSort_omp [-h] [-v] [-n numElements] [-t numThreads] [-e printPart]
   -h shows this message and exits.
   -v provides verbose output.
   -n indicates the number of elements to create that will be sorted
   -t indicates the number of threads to use. Default is 1 without this flag.
   -e indicates which part of the experiment results to print: 0(all results) 1(generate part)  2(sort part)
```

For example, when we want to find out how much time it takes for the program to generate and sort a random-generated string with 100000 characters in 4 threads, we can use this line

```
./countSort_omp -n 100000 -t 4
```

Data Collection From Parallel Version

In the program, I use the function `omp_get_wtime()` to get the start and end time for each part in order to find out how much time used for string generation and sorting part. Also, to print out the time for each case and condition in format, I added a new variable to `getArgument` to help us print out the experiment results. Besides the modifications to the parallel version of the program, I write two shell script files to help myself collect the data.

```
#!/bin/bash
# Name: run_strong_tests.sh
# Author: Shengyuan Wang
num_times=$1
test_obj=$2

printf "trial \t#th \t134217728 \t268435456 \t536870912 \t1073741824 \t2147483616 \n"
# each trial will run num_times using a certain number of threads
for num_threads in 1 2 4 6 8 12 16
do
    # run the series of problem sizes with the current value of num_threads
    counter=1
    while [ $counter -le $num_times ]
    do
        # $counter is the trial number
        printf "$counter\t$num_threads\t"

        # run each problem size once
        for problem_size in 134217728 268435456 536870912 1073741824 2147483616
        do
            if [ "$num_threads" == "1" ]; then
                command="./countSort_omp -n $problem_size -t 1 -e $test_obj"
            else
                command="./countSort_omp -n $problem_size -t $num_threads -e $test_obj"
            fi
            $command
        done
        printf "\n"
        ((counter++))
    done
    printf "\n"
done
```

```
#!/bin/bash
# Name: run_weak_tests.sh
# Author: Shengyuan Wang
num_times=$1
initial_size=$2
weak_scale_lines=$3
test_obj=$4
printf "trial \tproblem size \tthreads \ttime\n"
problem_size=$initial_size
double=0

for line in $(seq 1 $weak_scale_lines)
do
    echo "line: " $line

    for num_threads in 1 2 4 8 16
    do
        counter=1
        while [ $counter -le $num_times ]
        do
            printf "$counter\t$problem_size\t$num_threads\t"
            command="./countSort_omp -n $problem_size -t $num_threads -e $test_obj"
            $command
            printf "\n"
            ((counter++))
        done
        problem_size=$(($problem_size*2))
    done
    printf "\n"
    double=$((2**$line))
    problem_size=$(($initial_size*$double))
done
```

Our next goal is to try some experiments to show how this program behaves in terms of scalability for various problem sizes. With these two linux shell scripts, it can format output data into tab-separated lines that can be written to a file.

```
bash ./run_strong_tests.sh 5 1 > strong_count.tsv
bash ./run_weak_tests.sh 10 33554431 3 1 > weak_count.tsv
bash ./run_strong_tests.sh 5 2 > strong_sort.tsv
bash ./run_weak_tests.sh 10 33554431 3 2 > weak_sort.tsv
```

For strong test, the problem sizes (the length of the random-generated string)vary from 134217728 to 2147483616 and the number of threads used is 1, 2, 4, 6, 8, 12, 16. And I run the full strong test experiment by trying 5 replicates and test the word generation part and word sorting part separately.

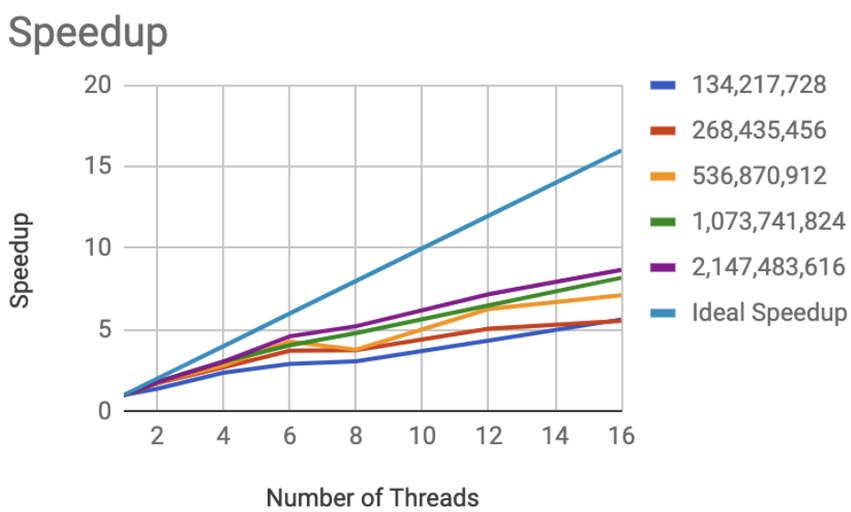
For weak scalability test, this script created data designed for an existing spreadsheet. The number 10 is for the number of replications to run, and the number 33554431 is a starting problem size. The number 3 is the number of weak scalability lines to be graphed on the spreadsheet.

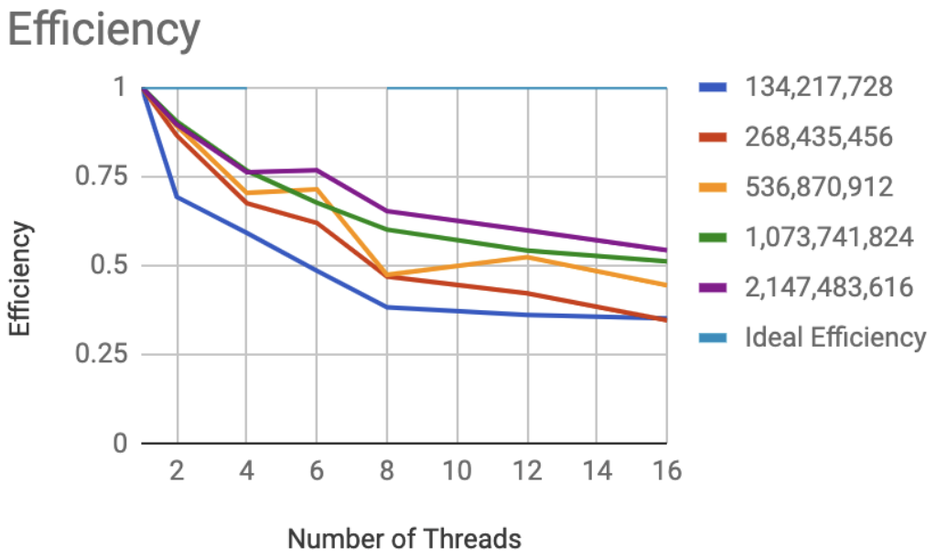
Analysis of Scalability and Efficiency

For strong scalability and efficiency tests, I tried five problem sizes from 134217728 to 2147483616 and for each problem size, the number of threads used for each problem size is 1, 2, 4, 6, 8, 12, 16 with 5 replicates for each strong test experiment.

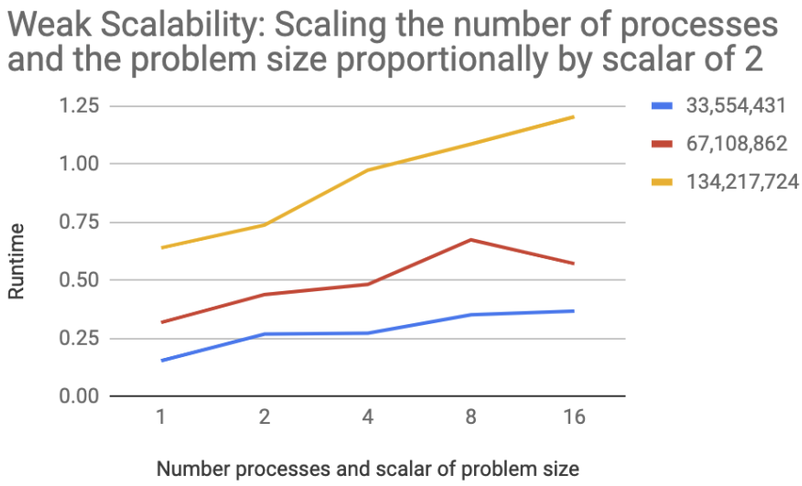
For weak scalability tests, the three starting problem size tried is 33554431, 67108862, 134217724. And we scaling the number of processes and the problem size proportionally by scalar of 2, and for each test case, we have 10 replications to run.

We can make several graphs using the data gained from strong scalability and efficiency tests and weak scalability tests of word sorting.





When the problem size (the random word size) is 134217728, the efficiency is below 0.75 from 2 to 16 threads, meaning that for this problem size, we should better not parallel it, instead we can compile it in sequential version. As for the problem sizes 268435456 and 536870912, the efficiency is clearly higher than 0.75 at 2 threads but lower than 0.75 at 4 threads. Thus, facing these two problem sizes, it demonstrate strong salability for 2 threads, thus paralleling the program in 2 threads works better here. For the other two problem size, 1073741824 and 2147483616 , it is not hard to find that the efficiency of the program is still higher than 0.75 for 4 threads, and the problem size 2147483616 even have a strong scalability for 6 threads. Thus, for the problem size of 1073741824, I would use 4 threads to parallel since it demonstrates strong scalability from 1 thread to 4 threads, and for the problem size of 2147483616, I would use 6 threads to parallel since it demonstrates strong scalability from 1 thread to 6 threads.



For the problem size of 67108862 and 134217724 , the run time does not change too much, so they both demonstrate weak scalability. However, for the problem size of 134217724, the program fails to demonstrate weak scalability.