Parallel Computing and Parallel algorithms Final Projects

Xinyuan Miao
020033910009
mxinyuan@sjtu.edu.cn

1 PROJECT1: PARALLEL PROGRAMMING WITH MPI

In this project, I complete three programming tasks using MPI:

- **MPI_ALLGATHER**: Use *MPI_SEND* and *MPI_RECV* to implement the *MPI_ALLGATHER* function, and compare the performance of your implementation and the original MPI implementation.
- **Gemm**: Parallelize the matrix multiplication, convolution operations and pooling operations in the tile unit.
- **Wordcout**: Implement the wordcount algorithm respectively for the two situations (small files and one big file) and print the results to the screen.

1.1 MPI_ALLGATHER

The Idea of the re-implementation of *MPI_Allgather* is simple. For the partial data that is already in the process, just copy it to the target location of the global receive buffer. Then, send the partial data to all the other processors using *MPI_Isend* and collect the other partial data from other processors using *MPI_Irecv*.

The comparison between the two implementation is showed in Figure. 1. The program randomly generated 1000000 characters and then benchmark the performance.

```
(base) matthewxy@Master:~/WorkSpaceParallel/project1$ mpirun -np 4 ./allgather benchmark MPI Al
lgather 1000000
Testing MPI Allgather
Total elapsed time:   0.000708
(base) matthewxy@Master:~/WorkSpaceParallel/project1$ mpirun -np 4 ./allgather benchmark MY All
gather 1000000
Testing MY Allgather
Total elapsed time:   0.008848
```

Figure 1: The comparison between *MPI_Allgather* and my re-implementation version.

1.2 Gemm

As mentioned in the description of this task, we should parallelize the program in the tile unit. Therefore, the segmentation is done according to the result matrix. For simplicity, all the three subtasks, we assume the number of processors p is **perfect square** so that it is easy to split the result matrix into p parts (both the column and row are divided evenly into p parts).

CS7344, June, 2021

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1.2.1 Matrix Multiplication. The segmentation is done according to the resulting matrix, we calculate the needed rows of matrix A and columns of matrix B for each processor and the root processor sends the input data to others (including the root processor itself). In the Fig. 2, we present a tiny example to validate the correctness of the program and then Fig. 3 shows the performance on the matrix multiplication of two random 1024 *1024 matrices. In general, the more processors you have, the faster your computing speed will be. But there are still examples of this phenomenon not being followed (as $p = 9$ and $p = 16$).

```
parallel@cpu-01 ~/020033910009
 % echo -e "Correctness validation: "
Correctness validation:
parallel@cpu-01 ~/020033910009
 % mpirun -np 4 ./matMulti 4 "true"
A:
3       6       7       5
3       5       6       2
9       1       2       7
0       9       3       6

B:
0       6       2       6
1       8       7       9
2       0       2       3
7       5       9       2

C:
55      91      107     103
31      68      71      85
54      97      92      83
57      102     123     102

Total elapsed time:   0.000118
```

Figure 2: The tiny example of the matrix multiplication.

1.2.2 Covcolution Operations & Pooling Operations. Similar as the matrix multiplication, the segmentation is done according to the result. We calculate the input required for each processor and allocate the part of the matrix from the root processor to all the other

```
parallel@cpu-01 ~/020033910009
 % mpirun -np 4 ./matMulti 1024 "false"
Total elapsed time:   1.710747

parallel@cpu-01 ~/020033910009
 % mpirun -np 9 ./matMulti 1024 "false"
Total elapsed time:   0.813500

parallel@cpu-01 ~/020033910009
 % mpirun -np 16 ./matMulti 1024 "false"
Total elapsed time:   0.898463

parallel@cpu-01 ~/020033910009
 % mpirun -np 25 ./matMulti 1024 "false"
Total elapsed time:   0.671079

parallel@cpu-01 ~/020033910009
 % mpirun -np 36 ./matMulti 1024 "false"
Total elapsed time:   0.545114
```

Figure 3: Performance of matrix multiplication with different p.

(including the root itself). Note that for the convolution operation, the kernel matrix will be sent to each processor intact. We keep the same assumption as mentioned in the matrix multiplication that the number of processors p is perfect square. In addition, we set the stride of the convolution and pooling as 1 and *the size of the kernel* respectively.

The benchmark result is showed in Fig. 4 and Fig. 5. From the figures, it seems that more processors brings negative impact on the performance. For this kind of counterintuitive result, there are two possible explanations. One is that the matrix is not big enough to show the advantages of multiprocessors as the computational overhead is relatively lower than communication. Another explanation is that the design of our algorithm has some flaws. For a processors that have been assigned a small input, we padded the input to the same size as the others have for programming convenience and this incurs additional computing overhead.

1.2.3 WordCount. There are two scenarios: WordCount on a bunch of files and WordCount on one big file. The algorithm is similar. As a classic Map-Reduce task, there are typically two phases, say the map phase and the reduce phase. In the map phase, each processor perform take some files or lines as input and output some intermediate key-value pairs. For the WordCount task, the form of intermediate pairs is like <*word*, 1>, which is the input of the reduce phase.

One key point is how to properly/evenly allocate this intermediate pairs to multiple processors. In my implementation, for each word w, I calculate its hash and mod p, the result is the ID of the processor that will fetch the pair the key of which is w in the reduce phase. This approach ensure that in the reduce phase, the same key

```
parallel@cpu-01 ~/020033910009
 % mpirun -np 4 ./convOP 1024 4 "false"
Total elapsed time:   0.061078

parallel@cpu-01 ~/020033910009
 % mpirun -np 16 ./convOP 1024 4 "false"
Total elapsed time:   0.110448

parallel@cpu-01 ~/020033910009
 % mpirun -np 25 ./convOP 1024 4 "false"
Total elapsed time:   0.152758

parallel@cpu-01 ~/020033910009
 % mpirun -np 36 ./convOP 1024 4 "false"
Total elapsed time:   0.221827
```

Figure 4: Performance of convolution operation with different p.

```
parallel@cpu-01 ~/020033910009
 % mpirun -np 4 ./poolingOP 1024 4 "false"
Total elapsed time:   0.030767

parallel@cpu-01 ~/020033910009
 % mpirun -np 16 ./poolingOP 1024 4 "false"
Total elapsed time:   0.037073

parallel@cpu-01 ~/020033910009
 % mpirun -np 25 ./poolingOP 1024 4 "false"
Total elapsed time:   0.038643

parallel@cpu-01 ~/020033910009
 % mpirun -np 36 ./poolingOP 1024 4 "false"
Total elapsed time:   0.068522
```

Figure 5: Performance of pooling operation with different p.

can only be processed by one processor. At the end of the reduce phase, all partial reusults will be sent into the root processor.

The WordCount result is showed in the Fig. 6, we pick top 10 words that appear the most frequently for the presentation.

2 PROJECT2: PARALLEL PROGRAMMING WITH OPENMP

In this project, I complete three programming tasks using OPENMP:

- **Monte Carlo algorithm**: Use OpenMP to implement the Monte Carlo algorithm.
- **Quick Sort**: Use OpenMP to implement a quick sorting algorithm with large data volume which contains 1000000 number.

```
parallel@cpu-01 ~/020033910009
 % sort -n -t ' ' -k 2 -r WordCountBig_results.txt | head -n 10
the 10399
of 5042
and 4164
to 3676
a 3115
that 3070
in 2879
was 2370
he 2205
had 2174
parallel@cpu-01 ~/020033910009
 % sort -n -t ' ' -k 2 -r WordCountSmall_results.txt | head -n 10
the 11653
to 6002
a 5203
of 4913
is 4595
n 4410
and 3937
in 3334
for 2297
that 2099
```

Figure 6: Top 10 frequent words.

- **PageRank**: Initialize a graph has 1, 024, 000 nodes and the edge count of different nodes ranges from 1 to 10. Implement the PageRank algorithm and run for 100 iterations.

2.1 Monte Carlo algorithm

According to the Monte Carlo algorithm to estimate the value of π, we generate 1000000 random coordinates (x, y) both of the x and y are in the range 0 to 1 and then calculate the proportions that meet $(x - 0.5) * (x - 0.5) + (y - 0.5) * (y - 0.5) < 0.25$. The calculated proportions is the estimated value of π. OpenMP can help us split the for-loop and allocate the calculation to multiple threads by the command **#pragma omp for**.

The Fig. 7 shows the result of the program with the 4 threads.

```
parallel@cpu-01 ~/020033910009/project2
 % ./MonteCarlo 4 1000000
Thread 3 takes 0.0089 s.
Thread 0 takes 0.008914 s.
Thread 1 takes 0.0089 s.
Thread 2 takes 0.008907 s.
After the calculation, total time cost: 0.009053 s.
After 1000000 trials, Pi is estimated to be 3.141684
```

Figure 7: The execution result of the Monte Carlo algorithm.

2.2 Quick Sort

To implement multithreaded calls to recursive functions, we use the **#pragma omp task** command in the **QuickSort** function, the detail of the program is showed in Fig. 8. The Fig. 9 shows a tiny example and the performance with different number of the threads is showed in Fig. 10. As the result showed in the figure, more threads do help to save the time.

2.3 PageRank

Similar to the Monte Carlo algorithm. we use **#pragma omp for** to split the for-loop to multiple threads. Each thread take charge for the update of some vertex in the graph. Fig. 12. I first give a tiny example that the number of iterations is 2 and then benchmark the program on a big graph with 102400 iterations' update.

```
void QuickSort(int *&array, int len, int l, int r, int id){
    if(len <= 1)return;
    int pivot = array[len/2];
    int lptr = 0;
    int rptr = len-1;
    while(lptr<=rptr){
        while(array[lptr]<pivot)lptr++;
        while(array[rptr]>pivot)rptr--;
        if(lptr<=rptr)
        {
            swap(array[lptr], array[rptr]);
            lptr++;
            rptr--;
        }
    }
    int *subarray[] = {array, &(array[lptr])};
    int sublen[] = {rptr+1, len-lptr};
    #pragma omp task default(none) firstprivate(subarray, sublen) shared(std::cout) firstprivate(l, rptr)
    {
        QuickSort(subarray[0], sublen[0], l, l+rptr, omp_get_thread_num());
    }
    #pragma omp task default(none) firstprivate(subarray, sublen) shared(std::cout) firstprivate(l, lptr, r)
    {
        QuickSort(subarray[1], sublen[1], l+lptr, r, omp_get_thread_num());
    }
}
```

Figure 8: The code snippet of the quick sort.

```
# Quick Sort

Correctness validation:
7 6 9 19 17 31 10 12 9 13 26 11 18 27 3 6
3 6 6 7 9 9 10 11 12 13 17 18 19 26 27 31
The sorting takes 0.000123 s.
```

Figure 9: The tiny example of quick sort.

```
parallel@cpu-01 ~/020033910009/project2
 % ./QuickSort 1 1000000 false
The sorting takes 0.285326 s.
parallel@cpu-01 ~/020033910009/project2
 % ./QuickSort 2 1000000 false
The sorting takes 0.153456 s.
parallel@cpu-01 ~/020033910009/project2
 % ./QuickSort 3 1000000 false
The sorting takes 0.103341 s.
parallel@cpu-01 ~/020033910009/project2
 % ./QuickSort 4 1000000 false
The sorting takes 0.099313 s.
parallel@cpu-01 ~/020033910009/project2
```

Figure 10: Performance of quick sort with different numbers of threads.

3 PROJECT3: BIG DATA ANALYSIS IN HADOOP SYSTEM

In this project, I implement weatherdata program using MapReduce. Referring to the official WordCount example, I modified the map function and reduce function to apply to the weatherdata program. The code is showed in Fig. 13.

```java
public class TempStatistics {
    public static class MaxTempMapper extends Mapper<LongWritable, Text, Text, FloatWritable>{
        String csvSplitBy = ",";
        public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException{
            InputSplit inputSplit = context.getInputSplit();
            String fileName = ((FileSplit) inputSplit).getPath().getName().split( regex: "\\.")[0];
            String[] entry = value.toString().split(csvSplitBy);
            if (entry[1].startsWith("\"Time\"")) return;
            Float temp = Float.parseFloat(entry[2]);
            context.write(new Text(fileName), new FloatWritable(temp));
        }
    }
    public static class MaxTempReducer extends Reducer<Text, FloatWritable, Text, FloatWritable>{
        public void reduce (Text key, Iterable<FloatWritable> values, Context context) throws IOException, InterruptedException {
            Float maxValues = Float.MIN_VALUE;
            Float minValues = Float.MAX_VALUE;
            for(FloatWritable value:values){
                maxValues = Math.max(maxValues, value.get());
                minValues = Math.min(minValues, value.get());
            }

            context.write(new Text( string: key.toString()+"_MAX\t"), new FloatWritable(maxValues));
            context.write(new Text( string: key.toString()+"_MIN\t"), new FloatWritable(minValues));
        }
    }

    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, jobName: "Temperature Statistics");
        job.setJarByClass(TempStatistics.class);
        job.setMapperClass(MaxTempMapper.class);
//        job.setCombinerCass()
        job.setReducerClass(MaxTempReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(FloatWritable.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion( verbose: true)? 0 : 1);
    }
}
```

Figure 13: The code snippet of the weatherdata program.

```
London2013_MAX          91.4
London2013_MIN          23.0
Mumbai2013_MAX          102.2
Mumbai2013_MIN          53.0
NewYork2013_MAX         99.0
NewYork2013_MIN         12.0
SFO2012_MAX             91.9
SFO2012_MIN             36.0
SFO2013_MAX             88.0
SFO2013_MIN             35.1
```

Figure 14: The execution result of the weatherdata program.


```
London2013_MAX          91.4
London2013_MIN          23.0
Mumbai2013_MAX          102.2
Mumbai2013_MIN          53.0
NewYork2013_MAX         99.0
NewYork2013_MIN         12.0
SFO2012_MAX             91.9
SFO2012_MIN             36.0
SFO2013_MAX             88.0
SFO2013_MIN             35.1
```


Figure 11: The execution result of the weatherdata program.

```
# PageRank

Correctness validation:
node id: 0 inDegree: 2 outDegree: 2 rank: 1 pointed from 3 7
node id: 1 inDegree: 3 outDegree: 1 rank: 1 pointed from 12 5 18
node id: 2 inDegree: 1 outDegree: 4 rank: 1 pointed from 11
node id: 3 inDegree: 3 outDegree: 4 rank: 1 pointed from 6 9 3
node id: 4 inDegree: 2 outDegree: 1 rank: 1 pointed from 16 17
node id: 5 inDegree: 2 outDegree: 2 rank: 1 pointed from 17 1
node id: 6 inDegree: 1 outDegree: 3 rank: 1 pointed from 11
node id: 7 inDegree: 1 outDegree: 2 rank: 1 pointed from 9
node id: 8 inDegree: 1 outDegree: 0 rank: 1 pointed from 2
node id: 9 inDegree: 2 outDegree: 5 rank: 1 pointed from 2 0
node id: 10 inDegree: 2 outDegree: 2 rank: 1 pointed from 4 10
node id: 11 inDegree: 1 outDegree: 3 rank: 1 pointed from 10
node id: 12 inDegree: 3 outDegree: 2 rank: 1 pointed from 0 12 5
node id: 13 inDegree: 2 outDegree: 0 rank: 1 pointed from 9 3
node id: 14 inDegree: 3 outDegree: 1 rank: 1 pointed from 7 19 2
node id: 15 inDegree: 1 outDegree: 0 rank: 1 pointed from 2
node id: 16 inDegree: 2 outDegree: 2 rank: 1 pointed from 6 9
node id: 17 inDegree: 3 outDegree: 3 rank: 1 pointed from 6 9 3
node id: 18 inDegree: 1 outDegree: 1 rank: 1 pointed from 11
node id: 19 inDegree: 3 outDegree: 1 rank: 1 pointed from 14 16 17
After 1 iterations:
node id: 0 rank: 0.75
node id: 1 rank: 2
node id: 2 rank: 0.333333
node id: 3 rank: 0.783333
node id: 4 rank: 0.833333
node id: 5 rank: 1.33333
node id: 6 rank: 0.333333
node id: 7 rank: 0.2
node id: 8 rank: 0.25
node id: 9 rank: 0.75
node id: 10 rank: 1.5
node id: 11 rank: 0.5
node id: 12 rank: 1.5
node id: 13 rank: 0.45
node id: 14 rank: 1.75
node id: 15 rank: 0.25
node id: 16 rank: 0.533333
node id: 17 rank: 0.783333
node id: 18 rank: 0.333333
node id: 19 rank: 1.83333
The calculation takes 0.007372 s.

1024000 nodes 100 iterations 4 threads:
The calculation takes 4.07496 s.
```

Figure 12: The execution result of the PageRank program.

The environment configuration and compilation process is trivial, here we just present the final execution result as showed in the Fig. 11