Parallel Computing and Programming Report

Wanghua Shi
s-whua@sjtu.edu.com
Shanghai Jiao Tong University
Shanghai, China

Abstract

This report is the projects of course "Parallel Computing and Programming". There are three projects including seven questions. The first project is MPI programming. The second project is OMP programming. The third project is a simple hadoop application.

For each question, I design experiments to verify the correctness and performance, and then analysis the result. The number of code line is about 2000(C/C++) and 1000(python). The python code is for drawing pictures. The project code can be obtained at https://github.com/shiwanghua/SharedFiles/tree/main/%E5%B9%B6%E8%A1%8C%E8%AE%A1%E7%AE%97%E4%B8%8E%E5%B9%B6%E8%A1%8C%E7%AE%97%E6%B3%95.

CCS Concepts: • **MPI** → **OpenMP**.

Keywords: MPI,OpenMP,hadoop,parallelism

ACM Reference Format:
Wanghua Shi. 2021. Parallel Computing and Programming Report. In . SJTU, Shang Hai, SH, CHINA, 6 pages.

1 Project One

"p" represents the number of processes ot threads. Without loss of generality, use randomly generated data of double type to do experiments.

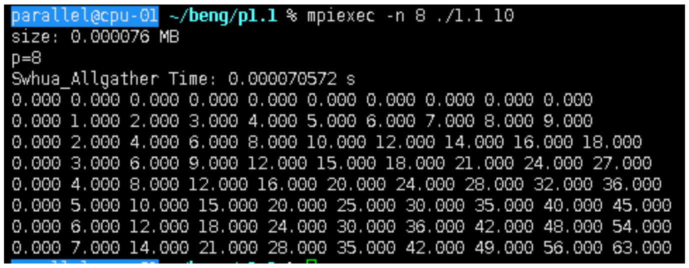
1.1 *MPI_Allgather*

1.1.1 Core Idea.

We can use the implementation of question 6.10 in howework 3 to implement customed *MPI_Allgather*. In 6.11, I realize *MPI_Bcast*. Calling the self-defined *MPI_Bcast* for p times based on each process can realize the same function of *MPI_Allgather*. In self-define *MPI_Bcast* function, the data from root should be stored in receive buffer, and then send data to other p-1 processes based on receive buffer.

1.1.2 Experiments.

1. Verification. Set p=8, each process generate 10 integer, the i-th integer is i times of the process id. After the *Swhua_Allgather* function is finished, output the receiving data in process 0:



```
parallel@cpu-01 ~/beng/p1.1 % mpiexec -n 8 ./1.1 10
size: 0.000076 MB
p=8
Swhua_Allgather Time: 0.000070572 s
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 1.000 2.000 3.000 4.000 5.000 6.000 7.000 8.000 9.000
0.000 2.000 4.000 6.000 8.000 10.000 12.000 14.000 16.000 18.000
0.000 3.000 6.000 9.000 12.000 15.000 18.000 21.000 24.000 27.000
0.000 4.000 8.000 12.000 16.000 20.000 24.000 28.000 32.000 36.000
0.000 5.000 10.000 15.000 20.000 25.000 30.000 35.000 40.000 45.000
0.000 6.000 12.000 18.000 24.000 30.000 36.000 42.000 48.000 54.000
0.000 7.000 14.000 21.000 28.000 35.000 42.000 49.000 56.000 63.000
```

Figure 1. Validate *Swhua_Bcast* function.

The "size" means the size of data transferred by each process. The running time is about 70.6 us. Line i is from process i. The j-th in i-th line is j times of number i. Hence, the function is correct.

2. Process Number Exp. Let each process send data of 256MB. Run *MPI_Allgather* and *Swhua_Allgather* with 1 to 29 processes and measure the running time. The result is shown in picture 2.

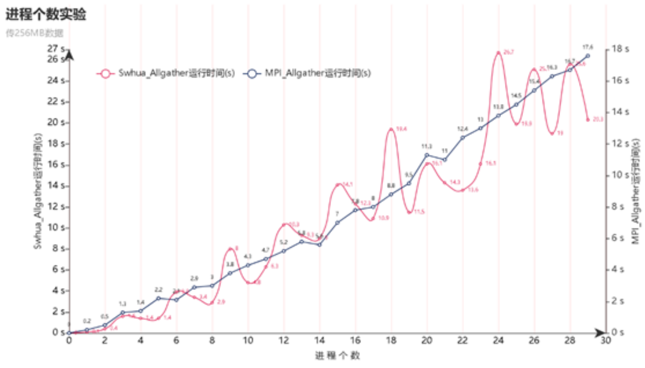


Figure 2. Running time of different process number.

The more the processes, the more the data needed to be broadcast. When p = 24, the running time of *Swhua_Bcast* reaches its maximum value 26.7 s. Overall, my version is lower than the MPI version. Besides, my version is not so stable as the data gets bigger.

3. Process Number Exp2. To ensure the data needed to transfered is a constant, I design another Process Number Exp. All processes sends 256MB data in total. So each process needs to broadcast $\frac{256MB}{p}$ data. Do experiments with p = 1 to 30. The result is as follows:

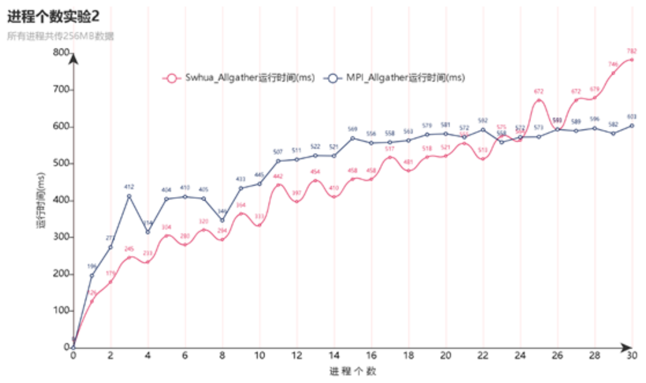


Figure 3. Running time of different process number(Fixed data size in total).

When p < 23, my Bcast function is faster. Since the data size is constant, my function becomes a little bit more stable than before.

4. DataTransmission Exp. Set p = 16. Let each process send 8B, 1KB, 1MB,64MB,256MB,512MB, the running time in millisecond is represented in figure 4.

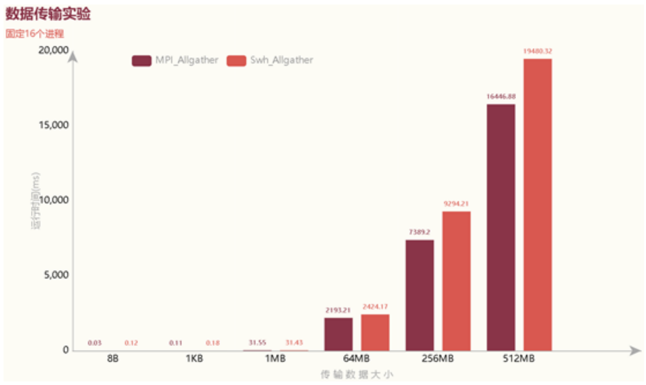


Figure 4. Running time in different data size.

The first three experiments have low running time although the data size is magnified many times, which means the growth of running time is not as fast as the growth of data volume. In the last three cases, the circumstances are the opposite. My implementation is all slower than *MPI_Allgather* in the six situations.

Moreover, nonblocking function *MPI_Isend* can be used in Bcast function. But I got slower elapsed time.

1.2 Gemm

1.2.1 Core Idea. For block matrix multiplication, use *MPI_Dims_create* [1] to get the row number and column number of grids. Use *MPI_Cart_create* to create communication domain. Use *MPI_Cart_coords* to get the grid coordinate of current process. Use *MPI_Comm_split* to create single row and single column internal communication domain. Use *MPI_Cart_rank* to get the grid ID of current process. Let process 0 be root process. Root process read or generate data and send data to relative process by *MPI_Send* or *MPI_Scatterv*. After every process get their data and finish calculating, for each row, use *MPI_Gatherv* and row communicators to collect the partial results of each column to the first process of each row. At last, use the first column communicator to collect the row results of the first column to root process and save the final result.

Convolutional operation can also be handled by this way. Set the step size to 1, the padding parameter to 0. The kernel size is marked "k" and original matrix has size ($n \times n$). The outcome matrix has size ($n-k+1$, $n-k+1$). But it is necessary to send (k-1) more rows and columns to calculate the result in the last row and the last column. For max pooling operation, the procedure is basically the same as convolutional operation but the kernel is not needed.

1.2.2 Experiments.

1. Verification. Generate three matrices a1, a2 and a3 in natural number sequence, each of which has 4 rows and 8 columns. Generate a matrix b in natural number sequence which has 4 rows and 8 columns. Generate a matrix k in natural number sequence which has 2 rows and 2 columns. Calculate $a1 \times b^T$, convolution of k on a2, (2×2) max pooling of a3. The results appear as shown in fig5.

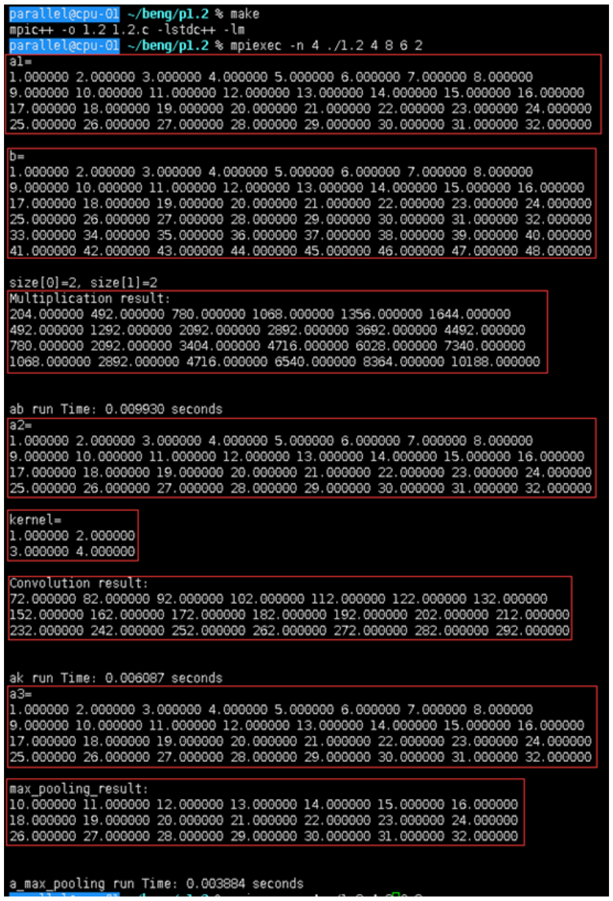
It is easy to see that every number is correct. The running time of matrix multiplication is 9.93ms, convolution is 6.087ms, and pooling is 3.884ms, which is faster and faster and logical.

2. Process Number Exp. Run experiments with 1 to 50 processes. In each time, set the shape of a and b to 1024×1024, set the kernel shape to 4×4. The running time curve is shown in fig6.

The row and column number of the grid under each process number is printed out on screen. For matrix multiplication, the fastest running time is 3.57 s when there are 10 processes. At this time, there are 5 grids in each column and 2 grids in each row; For convolution operation, situation of 17 processes is the fastest, followed by 7 processes, both less than 1.9 s; For pooling operation, the fastest time is reached at 19 processes, and the time is further reduced to 1.69 s. In general, the changing trend of the running time of the three operations is roughly the same.

1.3 Wordcount

1.3.1 Core Idea. Manager/Work mode is adopted in this question. Let process 0 be root again. Root process read text from files and send segmental text to other p-1 processes. When all the text is sent out, root process send finish message to inform other processes to do reduce operation. Once receiving a finish message from one process, root process will handle the partial result of that process. After reducing



```
parallel@cpu-01 ~/beng/p1.2 % make
mpic++ -o 1.2 1.2.c -lstdc++ -lm
parallel@cpu-01 ~/beng/p1.2 % mpiexec -n 4 ./1.2 4 8 6 2
a1=
1.000000 2.000000 3.000000 4.000000 5.000000 6.000000 7.000000 8.000000
9.000000 10.000000 11.000000 12.000000 13.000000 14.000000 15.000000 16.000000
17.000000 18.000000 19.000000 20.000000 21.000000 22.000000 23.000000 24.000000
25.000000 26.000000 27.000000 28.000000 29.000000 30.000000 31.000000 32.000000

b=
1.000000 2.000000 3.000000 4.000000 5.000000 6.000000 7.000000 8.000000
9.000000 10.000000 11.000000 12.000000 13.000000 14.000000 15.000000 16.000000
17.000000 18.000000 19.000000 20.000000 21.000000 22.000000 23.000000 24.000000
25.000000 26.000000 27.000000 28.000000 29.000000 30.000000 31.000000 32.000000
33.000000 34.000000 35.000000 36.000000 37.000000 38.000000 39.000000 40.000000
41.000000 42.000000 43.000000 44.000000 45.000000 46.000000 47.000000 48.000000

size[0]=2, size[1]=2
Multiplication result:
204.000000 492.000000 780.000000 1068.000000 1356.000000 1644.000000
492.000000 1292.000000 2092.000000 2892.000000 3692.000000 4492.000000
780.000000 2092.000000 3404.000000 4716.000000 6028.000000 7340.000000
1068.000000 2892.000000 4716.000000 6540.000000 8364.000000 10188.000000


ab run Time: 0.009930 seconds
a2=
1.000000 2.000000 3.000000 4.000000 5.000000 6.000000 7.000000 8.000000
9.000000 10.000000 11.000000 12.000000 13.000000 14.000000 15.000000 16.000000
17.000000 18.000000 19.000000 20.000000 21.000000 22.000000 23.000000 24.000000
25.000000 26.000000 27.000000 28.000000 29.000000 30.000000 31.000000 32.000000

kernel=
1.000000 2.000000
3.000000 4.000000

Convolution result:
72.000000 82.000000 92.000000 102.000000 112.000000 122.000000 132.000000
152.000000 162.000000 172.000000 182.000000 192.000000 202.000000 212.000000
232.000000 242.000000 252.000000 262.000000 272.000000 282.000000 292.000000


ak run Time: 0.006087 seconds
a3=
1.000000 2.000000 3.000000 4.000000 5.000000 6.000000 7.000000 8.000000
9.000000 10.000000 11.000000 12.000000 13.000000 14.000000 15.000000 16.000000
17.000000 18.000000 19.000000 20.000000 21.000000 22.000000 23.000000 24.000000
25.000000 26.000000 27.000000 28.000000 29.000000 30.000000 31.000000 32.000000

max_pooling_result:
10.000000 11.000000 12.000000 13.000000 14.000000 15.000000 16.000000
18.000000 19.000000 20.000000 21.000000 22.000000 23.000000 24.000000
26.000000 27.000000 28.000000 29.000000 30.000000 31.000000 32.000000


a_max_pooling run Time: 0.003884 seconds
```

Figure 5. Verification.

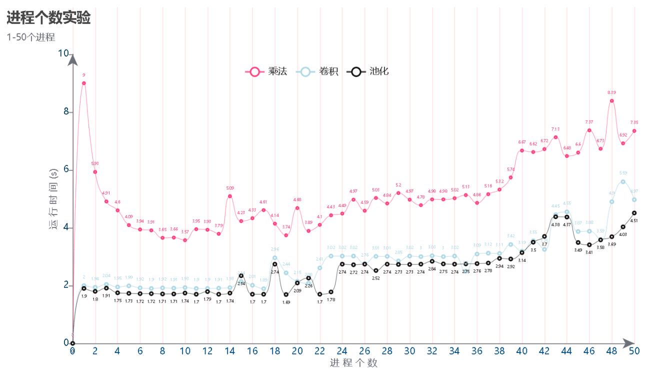


Figure 6. Runing time in different process number.

all segmental results, root process sorts and saves the word count result to local file named "*final_result.txt*".

Other p-1 processes will apply for text and find words from text and count the words continuously. Once receiving a finish message, they will stop map operation and save their segmental results to temporary files. At last, they will send a message to root process to show that they have finished their tasks.

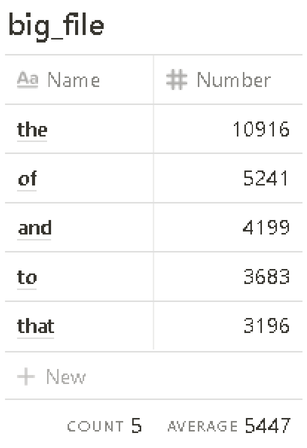
For the big file, I try two ways to distribute the text information: send a fixed length of text each time (Special treatment at the end); send one average length of text to each process, each process only needs to receive one time. Besides, there is a **space** at the end of each line in big file, and a word at the end of one line may be divided into two parts by a special **hyphens**, whose size is 2 bytes.

For small files, I also try two ways to distribute text: for each request from processes, root process send one small file based on original file sequence; arrange files in descending order according to file length, ensuring that the number of characters to be processed by each process is close. In addition, there are two **empty** files. For sorted method, once a empty file is encountered, the program can end immediately.

1.3.2 Experiments.

1. Verification. The file folder "*big_file*" and "*small_file*" are put in the same path as "1.3.c" file. The sorted word count results will be saved in "*big_file/final_result.txt*" and "*small_file/final_result.txt*".

According to my word recognition algorithm, I get the following results. In the big file, there are 147792 words, 10290 of which are different words. The top five words are shown in fig 7.



big_file

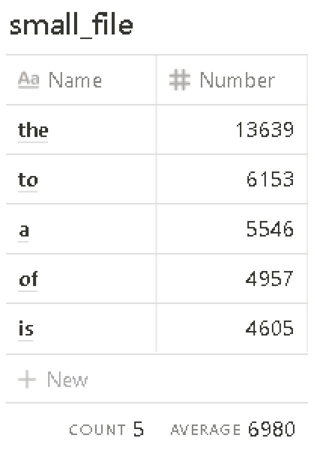
Aa Name	# Number
the	10916
of	5241
and	4199
to	3683
that	3196
+ New	
	COUNT 5 AVERAGE 5447

Figure 7. Top 5 words count in the big file.

In addition, I find there are 8 words "echo", 88 words "against" and 596 words "from" in the big file by using the search function (ctrl + F) of VSCode, which are the same as mine.

For the small files, there are 346680 words, 35386 of which are different words. The top five words are shown in fig 8.

2. Big file Process Number Exp. Set p = 1 to 50. Root process is not included in p in this experiment. For the fixed length version, the buffer size is set to 1024B. Record the running time of two word count algorithms for the big file. The result is fig 9. From the picture we can easily see that the



small_file

Aa Name	# Number
the	13639
to	6153
a	5546
of	4957
is	4605

+ New

COUNT 5 AVERAGE 6980

Figure 8. Top 5 words count in small files.

average sending version is better in most cases. This is because the number of communications has decreased. When p = 5, the running time reaches its minimum, 52 ms.

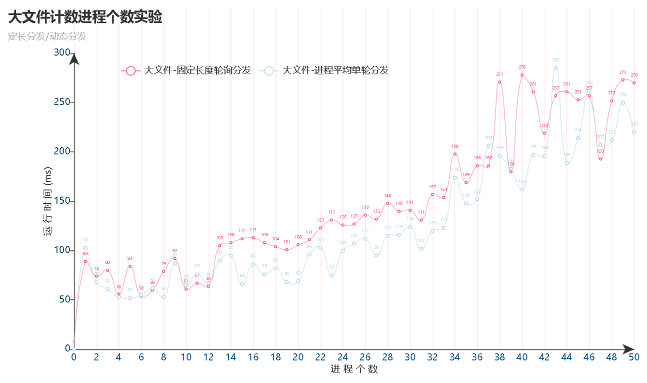


Figure 9. Runing time of two big file word count algorithms in different process number.

3. Small files Process Number Exp. Set p = 1 to 50 again. Root process is not included in p in this experiment. Record the runing time of two word count algorithms for small files. The result is fig 10. In most cases, the sort version has lower running time and better stability, which guarantees the fairness for every process. When p = 2 and 3, the running time reaches its minimum, 181 ms.

4. Stand-alone/Cluster Exp. Run fixed length big file word count algorithm in stand-alone mode and cluster mode. Set p = 1 to 50. Root process is not included in p. The experimental result is in fig 11. Cluster mode is slightly better than stand-alone mode. When p = 3, cluster mode finishes word count task in 43 ms. Nine milliseconds less than before.

5. Stand-alone/Cluster Exp2. Run four word count algorithms in stand-alone mode and cluster mode. Set p = 5.

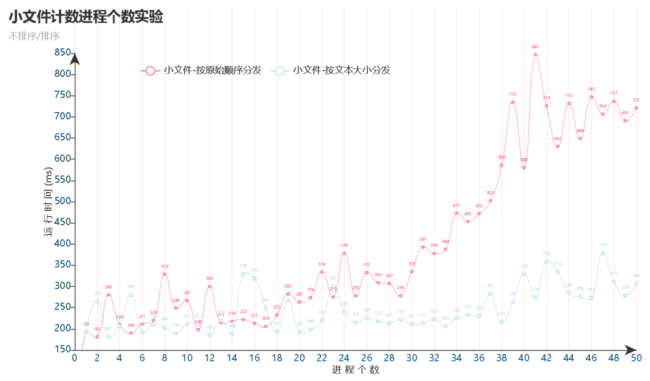


Figure 10. Runing time of two small files word count algorithms in different process number.

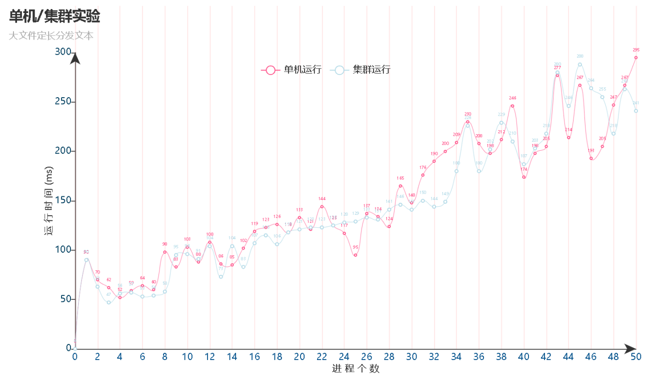


Figure 11. Runing time of big file word count task in stand-alone and cluster mode.

Four processes ask for data and do the word count recognition algorithms. The output is in fig 12. Besides the average version of big file, cluster mode shows better performance in the other three cases. Maybe the delay for sending between hosts is too long. When the root process send data to one process, those who are behind the process have to wait.

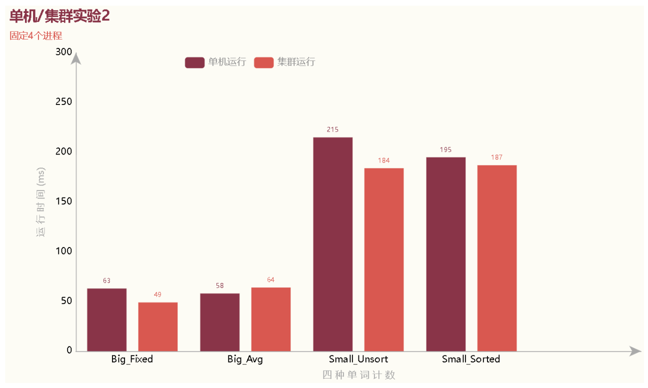


Figure 12. Runing time of four algorithms in stand-alone and cluster mode.

2 Project Two

2.1 Monte Carlo

Input the number of threads and points, count the number of points(n) in unit circle. Use "#pragma omp parallel for" to realize parallelization. Record the running time, the estimated π and the accuracy of estimated π.

Use one single thread to run the simulation program with different n: from 10, 100, 1000, ... to 10000000000. The output is shown in fig 13. When n = 1000, it is accurate enough to estimate π.

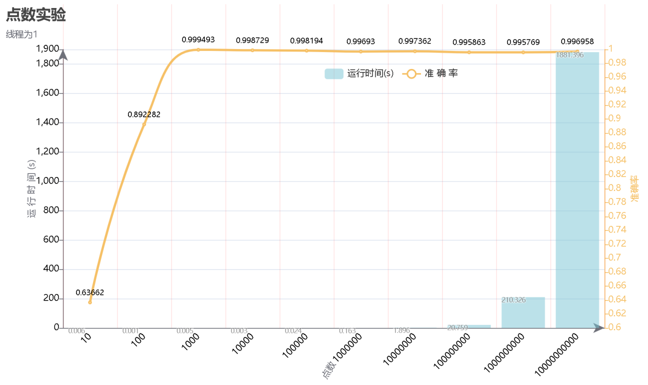


Figure 13. Runing time and accuracy in different number of points.

2.2 Quick Sort

Once determining a final position of one element, use two threads to sort the left and right subarray. When the array length is 1000000, the running time of omp version is bigger (see fig 14). This is because the required threads are too many. From the figure we can see that the generated number are different.

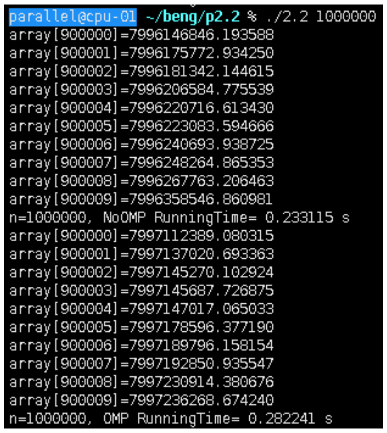

```
parallel@cpu-01 ~/beng/p2.2 % ./2.2 1000000
array[900000]=7996146846.193588
array[900001]=7996175772.934250
array[900002]=7996181342.144615
array[900003]=7996206584.775539
array[900004]=7996220716.613430
array[900005]=7996223083.594666
array[900006]=7996240693.938725
array[900007]=7996248264.865353
array[900008]=7996267763.206463
array[900009]=7996358546.860981
n=1000000, NoOMP RunningTime= 0.233115 s
array[900000]=7997112389.080315
array[900001]=7997137020.693363
array[900002]=7997145270.102924
array[900003]=7997145687.726875
array[900004]=7997147017.065033
array[900005]=7997178596.377190
array[900006]=7997189796.158154
array[900007]=7997192850.935547
array[900008]=7997230914.380676
array[900009]=7997236268.674240
n=1000000, OMP RunningTime= 0.282241 s
```

Figure 14. Output when n = 1000000.

When the array length is 10000, less threads are required, so the running time decreases (see fig 15).

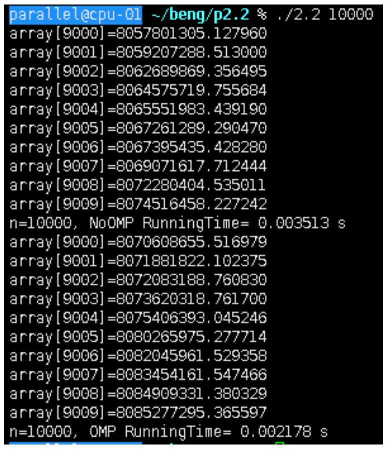

```
parallel@cpu-01 ~/beng/p2.2 % ./2.2 10000
array[9000]=8057801305.127960
array[9001]=8059207288.513000
array[9002]=8062689869.356495
array[9003]=8064575719.755684
array[9004]=8065551983.439190
array[9005]=8067261289.290470
array[9006]=8067395435.428280
array[9007]=8069071617.712444
array[9008]=8072280404.535011
array[9009]=8074516458.227242
n=10000, NoOMP RunningTime= 0.003513 s
array[9000]=8070608655.516979
array[9001]=8071881822.102375
array[9002]=8072083188.760830
array[9003]=8073620318.761700
array[9004]=8075406393.045246
array[9005]=8080265975.277714
array[9006]=8082045961.529358
array[9007]=8083454161.547466
array[9008]=8084909331.380329
array[9009]=8085277295.365597
n=10000, OMP RunningTime= 0.002178 s
```

Figure 15. Output when n = 10000.

If when the length of subarray is smaller than 50, directly use insertion sort instead, then less threads are required, the running time can also be reduced (see fig 16).

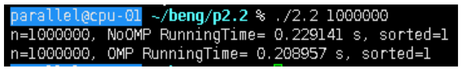

```
parallel@cpu-01 ~/beng/p2.2 % ./2.2 1000000
n=1000000, NoOMP RunningTime= 0.229141 s, sorted=1
n=1000000, OMP RunningTime= 0.208957 s, sorted=1
```

Figure 16. Runing time while using insertion sort optimization.

2.3 PageRank

Set the damping coefficient to 0.85. Use "guided" schedule mode. Use "unordered_set" to store the link number so that there are no duplicate link numbers. Run program in 1 to 50 threads. The running time is in fig 17. The minimum running time is 3735 ms when p = 35.

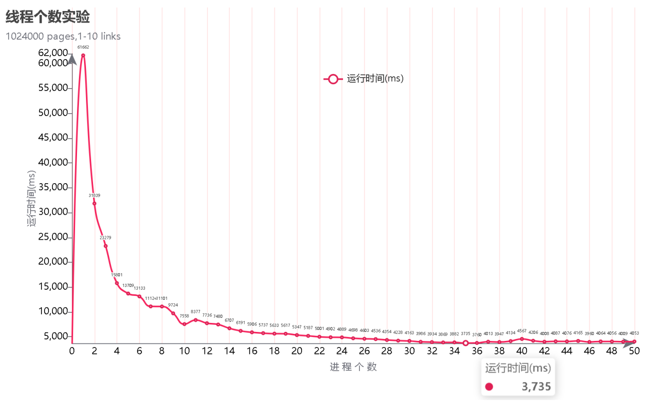
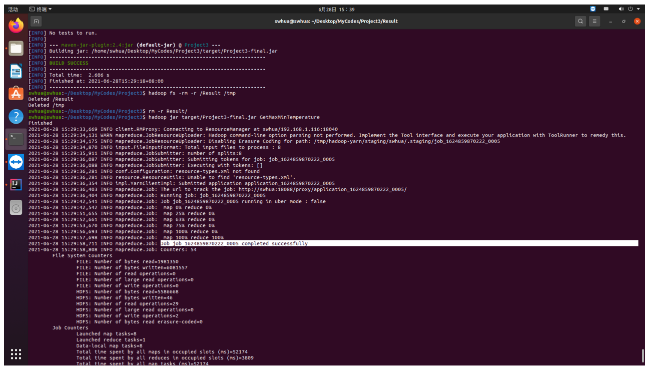


Figure 17. Runing time in different number of threads.

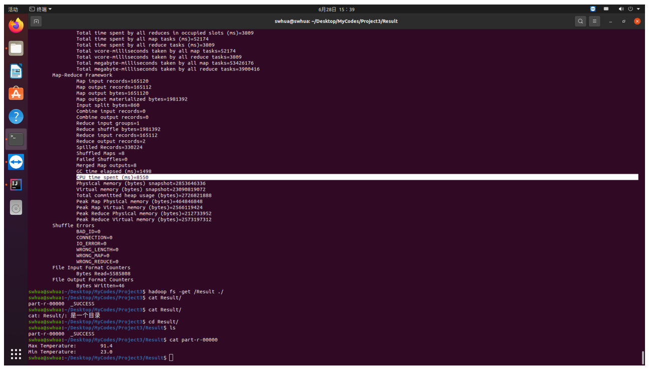
3 Project Three

Use “load” and “write.table” function of R language to convert RDA file to TXT file. In “sbin” directory of hadoop package, run command “./start-all.sh” to start the cluster. Run command “hadoop fs -mkdir /txtData” to make a directory in hdfs. Run command “hadoop fs -put Ram-N-weatherData-4326bdc/txtData /” to upload TXT files to hdfs. Run command “hadoop fs -ls /txtData” to show the files uploaded. Run command “mvn clean install package” to get the jar package. If it is not the first time to run the program, run command “hadoop fs -rm -r /Result /tmp” to remove the result of the last run of the program. Run command “hadoop jar target/Project3-final.jar GetMaxMinTemperature” to upload and run the jar package. Wait for the program to finish running. Run command “hadoop fs -get /Result ./” to download output files from hdfs. Run command “cat Result/part-r-00000” to show the final maximum and minimum temperature. All the outputs are shwon in fig 18 and fig 19.



```
[INFO] No tests to run.
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ Project3 ---
[INFO] Building jar: /home/swhua/Desktop/MyCodes/Project3/target/Project3-final.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  2.606 s
[INFO] Finished at: 2021-06-28T15:29:18+08:00
[INFO] ------------------------------------------------------------------------
swhua@swhua:~/Desktop/MyCodes/Project3$ hadoop fs -rm -r /Result /tmp
Deleted /Result
Deleted /tmp
swhua@swhua:~/Desktop/MyCodes/Project3$ rm -r Result/
swhua@swhua:~/Desktop/MyCodes/Project3$ hadoop jar target/Project3-final.jar GetMaxMinTemperature
Finished
2021-06-28 15:29:33,669 INFO client.RMProxy: Connecting to ResourceManager at swhua/192.168.1.116:18040
2021-06-28 15:29:34,131 WARN mapreduce.JobResourceUploader: Hadoop command-line option parsing not performed. Implement the Tool interface and execute your application with ToolRunner to remedy this.
2021-06-28 15:29:34,175 INFO mapreduce.JobResourceUploader: Disabling Erasure Coding for path: /tmp/hadoop-yarn/staging/swhua/.staging/job_1624859870222_0005
2021-06-28 15:29:34,870 INFO input.FileInputFormat: Total input files to process : 8
2021-06-28 15:29:35,911 INFO mapreduce.JobSubmitter: number of splits:8
2021-06-28 15:29:36,087 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1624859870222_0005
2021-06-28 15:29:36,088 INFO mapreduce.JobSubmitter: Executing with tokens: []
2021-06-28 15:29:36,281 INFO conf.Configuration: resource-types.xml not found
2021-06-28 15:29:36,281 INFO resource.ResourceUtils: Unable to find 'resource-types.xml'.
2021-06-28 15:29:36,354 INFO impl.YarnClientImpl: Submitted application application_1624859870222_0005
2021-06-28 15:29:36,403 INFO mapreduce.Job: The url to track the job: http://swhua:18088/proxy/application_1624859870222_0005/
2021-06-28 15:29:36,404 INFO mapreduce.Job: Running job: job_1624859870222_0005
2021-06-28 15:29:42,541 INFO mapreduce.Job: Job job_1624859870222_0005 running in uber mode : false
2021-06-28 15:29:42,542 INFO mapreduce.Job:  map 0% reduce 0%
2021-06-28 15:29:51,655 INFO mapreduce.Job:  map 25% reduce 0%
2021-06-28 15:29:52,661 INFO mapreduce.Job:  map 63% reduce 0%
2021-06-28 15:29:53,670 INFO mapreduce.Job:  map 75% reduce 0%
2021-06-28 15:29:56,693 INFO mapreduce.Job:  map 100% reduce 0%
2021-06-28 15:29:57,698 INFO mapreduce.Job:  map 100% reduce 100%
2021-06-28 15:29:58,711 INFO mapreduce.Job: Job job_1624859870222_0005 completed successfully
2021-06-28 15:29:58,808 INFO mapreduce.Job: Counters: 54
        File System Counters
                FILE: Number of bytes read=1981350
                FILE: Number of bytes written=6081557
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=5586668
                HDFS: Number of bytes written=46
                HDFS: Number of read operations=29
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=2
                HDFS: Number of bytes read erasure-coded=0
        Job Counters
                Launched map tasks=8
                Launched reduce tasks=1
                Data-local map tasks=8
                Total time spent by all maps in occupied slots (ms)=52174
                Total time spent by all reduces in occupied slots (ms)=3809
                Total time spent by all map tasks (ms)=52174
```

Figure 18. Output1



```
                Total time spent by all reduces in occupied slots (ms)=3809
                Total time spent by all map tasks (ms)=52174
                Total time spent by all reduce tasks (ms)=3809
                Total vcore-milliseconds taken by all map tasks=52174
                Total vcore-milliseconds taken by all reduce tasks=3809
                Total megabyte-milliseconds taken by all map tasks=53426176
                Total megabyte-milliseconds taken by all reduce tasks=3900416
        Map-Reduce Framework
                Map input records=165120
                Map output records=165112
                Map output bytes=1651120
                Map output materialized bytes=1981392
                Input split bytes=860
                Combine input records=0
                Combine output records=0
                Reduce input groups=1
                Reduce shuffle bytes=1981392
                Reduce input records=165112
                Reduce output records=2
                Spilled Records=330224
                Shuffled Maps =8
                Failed Shuffles=0
                Merged Map outputs=8
                GC time elapsed (ms)=1498
                CPU time spent (ms)=8550
                Physical memory (bytes) snapshot=2853646336
                Virtual memory (bytes) snapshot=23090819072
                Total committed heap usage (bytes)=2726821888
                Peak Map Physical memory (bytes)=464846848
                Peak Map Virtual memory (bytes)=2566119424
                Peak Reduce Physical memory (bytes)=212733952
                Peak Reduce Virtual memory (bytes)=2573197312
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=5585808
        File Output Format Counters
                Bytes Written=46
swhua@swhua:~/Desktop/MyCodes/Project3$ hadoop fs -get /Result ./
swhua@swhua:~/Desktop/MyCodes/Project3$ cat Result/
part-r-00000  _SUCCESS
swhua@swhua:~/Desktop/MyCodes/Project3$ cat Result/
cat: Result/: 是一个目录
swhua@swhua:~/Desktop/MyCodes/Project3$ cd Result/
swhua@swhua:~/Desktop/MyCodes/Project3/Result$ ls
part-r-00000  _SUCCESS
swhua@swhua:~/Desktop/MyCodes/Project3/Result$ cat part-r-00000
Max Temperature:        91.4
Min Temperature:        23.0
swhua@swhua:~/Desktop/MyCodes/Project3/Result$ 
```

Figure 19. Output2

The maximum temperature is 91.4. The minimum temperature is 23.0. The running time is 8550 ms.

Acknowledgments

To teachers and assistants, for providing cluster experimental environment and answering the questions and doubts.

References

[1] Michael J. Quinn. 2003. *Parallel Programming in C with MPI and OpenMP* (1st. ed.). McGraw-Hill Education, New York, NY.