Report

1. Description

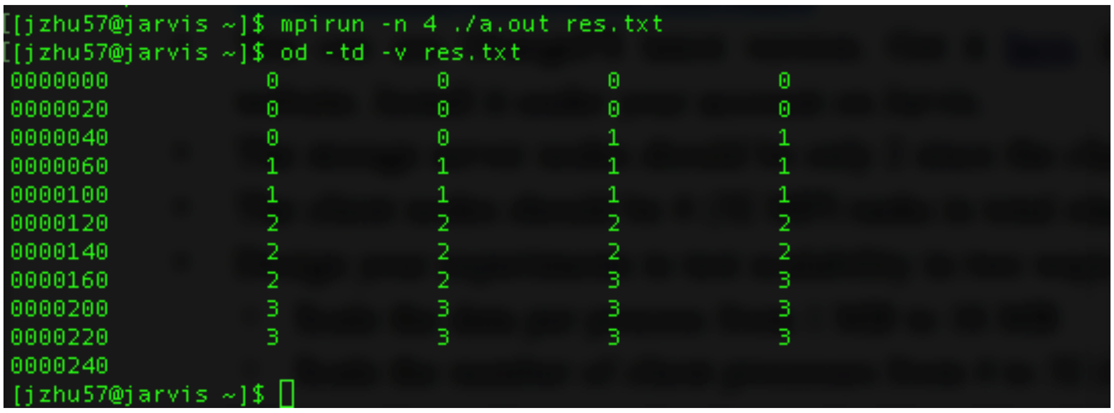
   In this assignment, I implemented a simple MPI-IO program. There are two parts in my assignment, the first is to gain some familiarity with MPI-IO and we need to let each MPI process to write its rank to a common file. Then check the correctness the output result. The second part is to build a parallel I/O benchmark using MPI-IO. The process is to open a temporary file, write random data to file and close the file. Then open it again, read data and close file in the end.

   Following is the two parts:
   Part 1:
   Firstly, for each rank, assign the rank value to a buffer array, which size is 10, then use MPI_FILE_OPEN() a common for each rank and set the write position, which is equal to "rank*sizeof(int)*10", for each rank in this common file. Finally, each rank writes the value of buffer array on their position. In order to check the correctness of the output file, I use the command "od -td –v filename" to check.

   Part 2:
   ①Environment configuration:
   Follow the orangefs documentation to build the orangefs environment. In Jarvis, it still need a new version of mpi, so I download mpich-3.2, then build it in Jarvis.
   I built 2 servers, and keep the client number equals to 4.
   (the details for configuration can see on readme.txt file).
   ②Code Design
   As we need to test the bandwidth from the underlying parallel file system, so I tested write bandwidth and read bandwidth.
   <1>First, start to record the overall_start_time after init the MPI, then each rank writes random data to a write buffer, which size from 1MB to 16MB/
   <2>use MPI_BARRIER() to make sure that each rank stop at this point and start at the same time and record the start time.
   <3> The writing method is the same as we talked in part1. When write finished, record the end time and use MPI_REDUCE() to get the maximum running time in ranks.
   <4> Write bandwidth= (number of process×data_size) / maximum time.
   <5> Close the file. Then, the open the common file again to read. The method to get view position is the same as in part1 and the method of reading time calculation is also the same as calculating the writing time. then we can get the read bandwidth.
   <6> Finally, record the overall_end time to get the overall time and write all the time we get to an output file.

2. My optimization and problems encountered.
   The biggest problem for me is to build the orangeFS in Jarvis, I tried a lot of times, and fortunately, it is successful finally. Also, at the beginning, I cannot control the client number. When I run my code, sometimes, I can get more than 4 node , but sometimes less. Afterwards, I tried to use the specific nodes by adding the name of node on my running command line: following is the example I use only two nodes:
   qsub –l h=" gpu-compute-0-0.local|gpu-compute-0-1.local|" -pe mpich 32 run1.sh.

3. Performance Evaluation
   For part1, I checked the correctness of my MPI_IO, following is a result of an example that I use 4 processes and each write their rank on the common file.



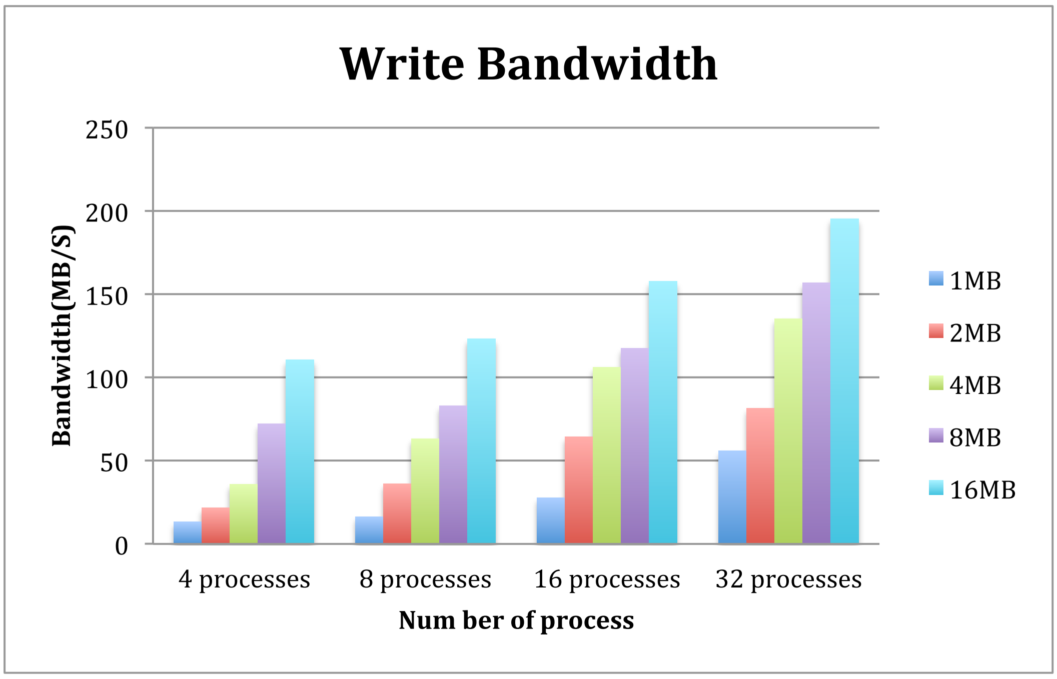
```
[jzhu57@jarvis ~]$ mpirun -n 4 ./a.out res.txt
[jzhu57@jarvis ~]$ od -td -v res.txt
0000000          0          0          0          0
0000020          0          0          0          0
0000040          0          0          1          1
0000060          1          1          1          1
0000100          1          1          1          1
0000120          2          2          2          2
0000140          2          2          2          2
0000160          2          2          3          3
0000200          3          3          3          3
0000220          3          3          3          3
0000240
[jzhu57@jarvis ~]$ 
```

Figure 1. The correctness check for MPI_FILE_WRITE()

For part 2, I tested different size of data for each rank, and the same data size for different number of process.
I scale the data per process from 1MB to 16MB, also scale the number of client processes form 4 to 32.
Following is the result:

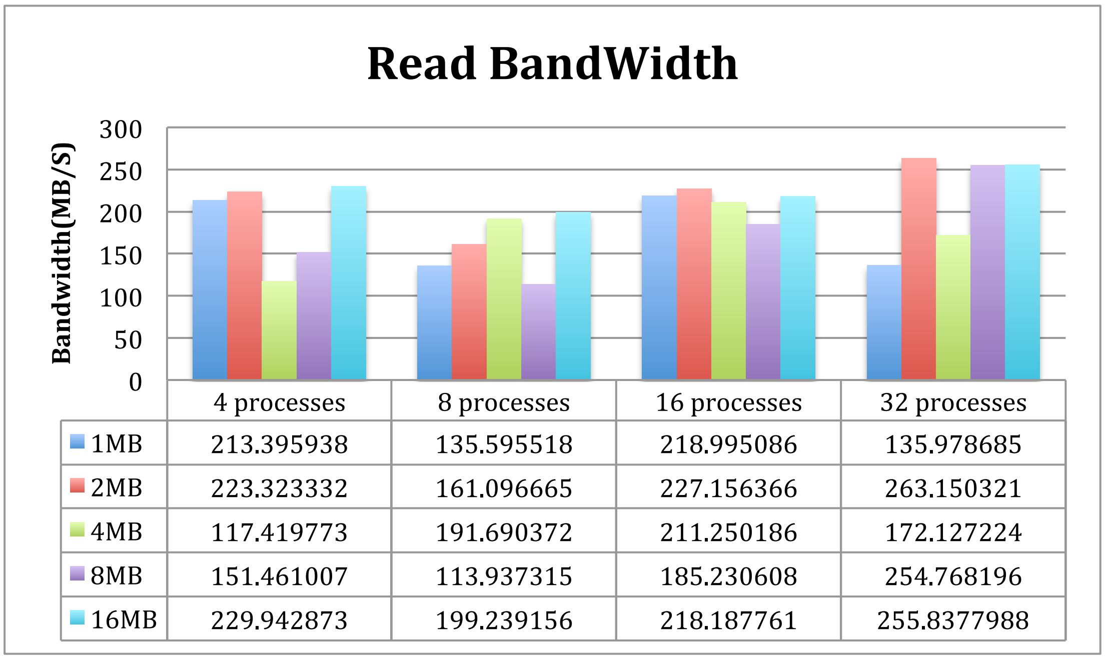
	4processes	8processes	16processes	32processes
1MB	13. 183134	16. 414077	27. 656922	55. 849053
2MB	21. 645373	36. 200395	64. 408656	81. 359206
4MB	35. 729458	63. 081408	106. 231835	135. 399772
8MB	72. 074304	82. 943241	117. 630706	156. 849770
16MB	110. 518513	123. 134228	157. 767898	195. 444837

Table 1. the write bandwidth with different number of process and different size of data



Plot 1. the write bandwidth with different number of process and different size of data

We can see from the above plot, at the same data size, with the increasing number of process, the bandwidth becomes larger. It is because that more cores can execute faster, also we use 4 nodes to run the program, since each node has 8 cores, totally we can have 32 MPI ranks, so the bandwidth is the largest with 32 processes. Also, at the same number of process, with more data, the bandwidth becomes bigger. We know there is latency in system, when the data size becomes bigger, the influence of latency will become less, so the bandwidth become larger.

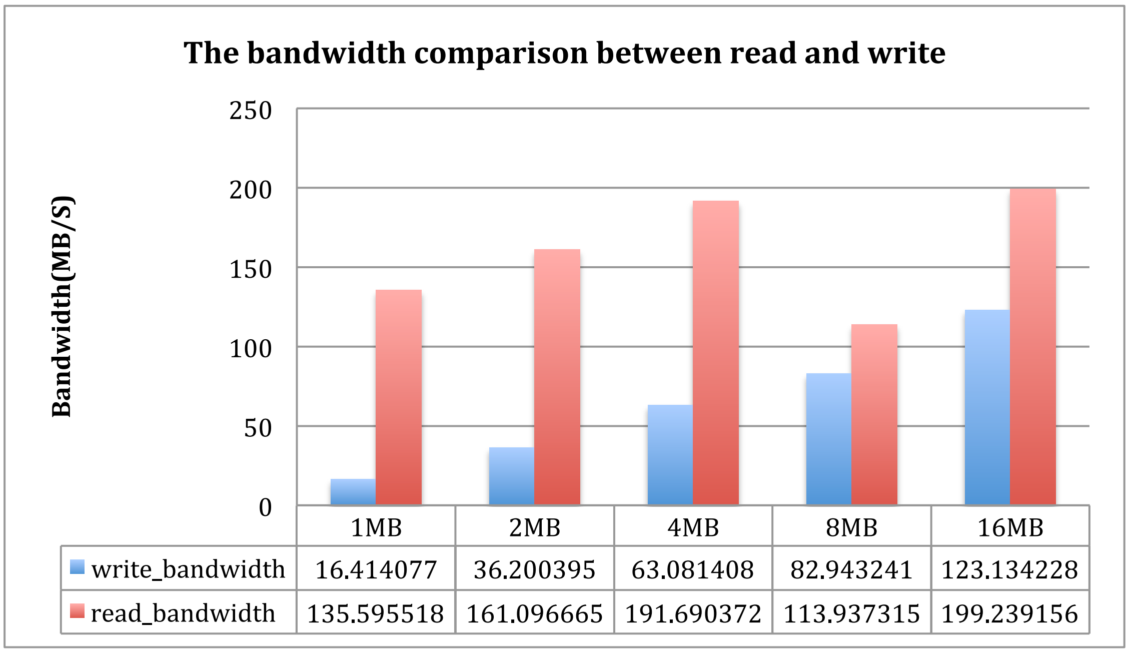


	4 processes	8 processes	16 processes	32 processes
🟦 1MB	213.395938	135.595518	218.995086	135.978685
🟥 2MB	223.323332	161.096665	227.156366	263.150321
🟩 4MB	117.419773	191.690372	211.250186	172.127224
🟪 8MB	151.461007	113.937315	185.230608	254.768196
🟦 16MB	229.942873	199.239156	218.187761	255.8377988

PLOT 2. the read bandwidth with different number of process and different size of data

Also we can see from the above plot, which the tendency is similar to write bandwidth. However, there are some measurement errors for same data because numerous users run in Jarvis and the cluster is not stable. In fact, the reading bandwidth is larger than writing bandwidth.

And I compared them with process number =8.
Following is the result:



	1MB	2MB	4MB	8MB	16MB
🟦 write_bandwidth	16.414077	36.200395	63.081408	82.943241	123.134228
🟥 read_bandwidth	135.595518	161.096665	191.690372	113.937315	199.239156

Figure 3. The bandwidth comparison between read and write with number of process=8

From above figure, we can see apparently that the reading bandwidth is larger than writing bandwidth, which is general rule we all know that.