CS546 HOMEWORK 6

SERGIO PENAVADES SUAREZ
CWID: A20387929

1. Introduction

The purpose of this document is to show how I've implemented my code using MPI-IO to perform operations over a parallel file system. Besides, this homework is composed by 2 parts. The first part consists on implementing a simple code in which a file is written and read. The second one consists on codding a benchmark based on the first code, calculate the maximum achieved bandwidth and present the output file.

2. Process of implementation

In this homework I analyzed how to implementing a parallel program using MPI-IO, identifying the complexity of them and how different instructions improve the application performance. With the analysis I acquired the enough knowledge to fulfill the requirements presented in the homework statements.

With a global vision of MPI-OI, I studied some internet code examples to understand everything inside and I developed the parallel code, making tests, measuring execution times and ensuring that the code worked well.

3. Part 1

3.1. Develop process

This first part of the homework, I used MPI_WRITE() and MPI_READ() functions to write the file and check the correctness.

During the coding, the principal problem presented was to define if I was check the output file correctness using a sequential way or using the MPI_READ() function. Finally, y decided to check the file parallel using MPI.

Introducing how I check different set of possible errors that can appear in the code, I used the MPI_CHECK() function provided to check handle errors in all the MPI Statements. Additionally, I check if the input provided has the file where write the results.

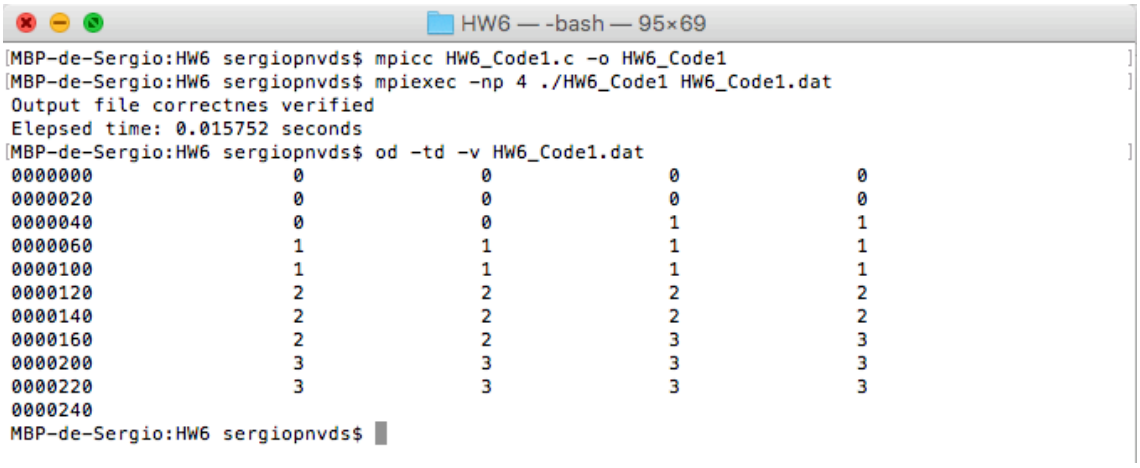
Describing step by step how the code:

- Check if the the output file is provided
- Share with all process if there is a file error input
- If there is not error
    - Take initial time
    - Calculate the offset
    - Open the output file for writing
    - Each process writes its rank ten times
    - Close the file
    - Open the file for reading
    - Get the output pile size
    - Calculate the buffer size for each process
    - Read the output file
    - Check the correctness of each process
    - Join all process correctness in one variable(“totalCorrectness”)
    - Close file
    - Take final time
    - MPI finalize
    - Return if correct or not
    - Return elapsed time
- If and error
    - Return “Input error”
    - Return elapsed time
- Exit

3.2. Output files

3.2.1. *Personal computer*

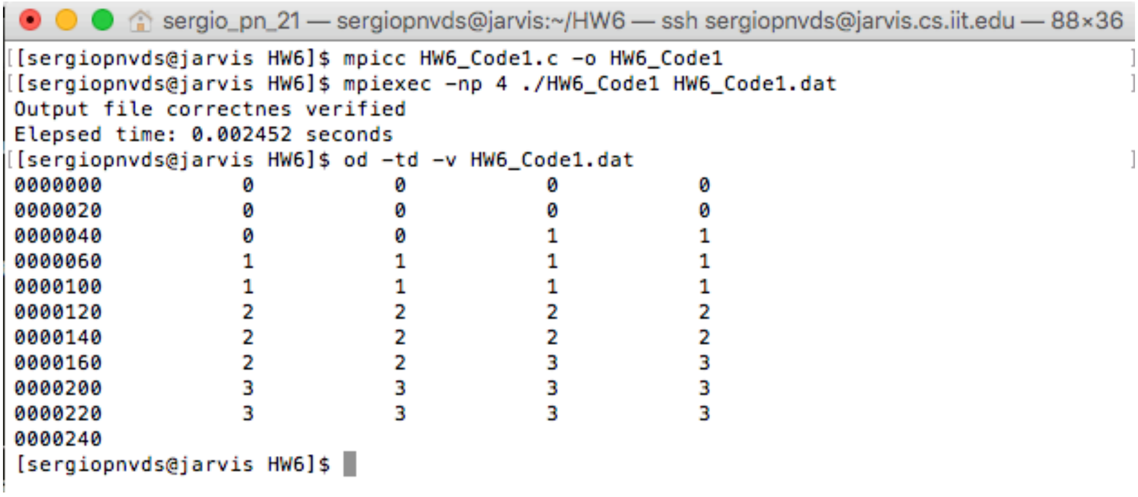
The figure bellow represents the result of the first part in my personal computer.



```
MBP-de-Sergio:HW6 sergiopnvds$ mpicc HW6_Code1.c -o HW6_Code1
MBP-de-Sergio:HW6 sergiopnvds$ mpiexec -np 4 ./HW6_Code1 HW6_Code1.dat
Output file correctnes verified
Elepsed time: 0.015752 seconds
MBP-de-Sergio:HW6 sergiopnvds$ od -td -v HW6_Code1.dat
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
MBP-de-Sergio:HW6 sergiopnvds$ 
```

3.2.2. *Jarvis computer*

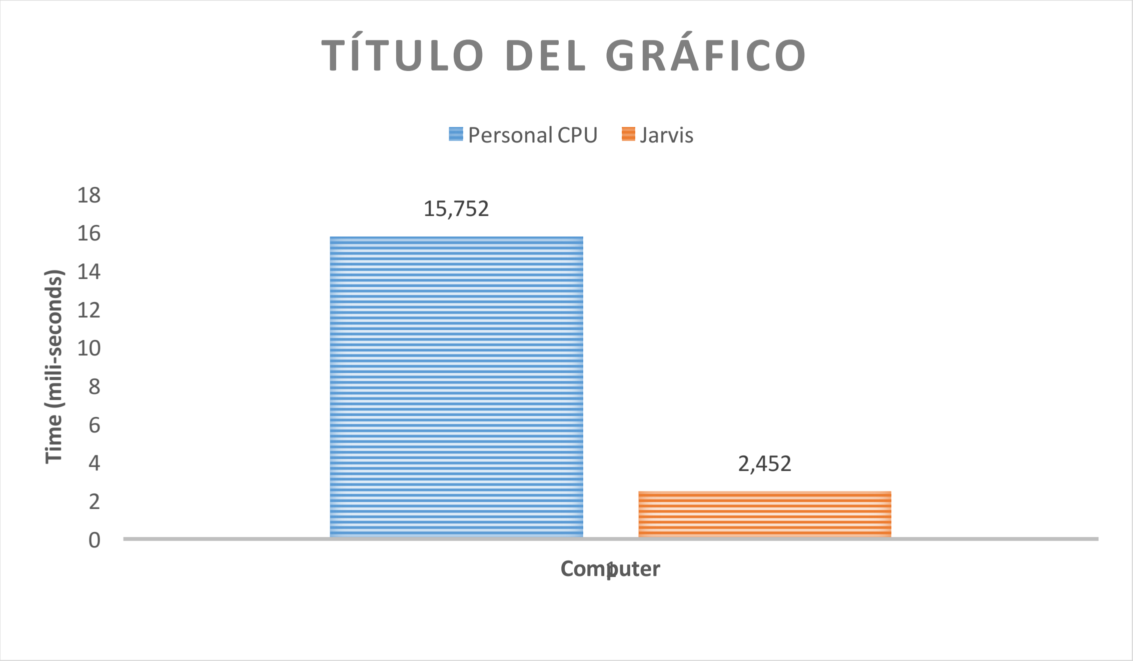
The figure bellow represents the result of the first part in jarvis.


sergio_pn_21 — sergiopnvds@jarvis:~/HW6 — ssh sergiopnvds@jarvis.cs.iit.edu — 88×36

```
[[sergiopnvds@jarvis HW6]$ mpicc HW6_Code1.c -o HW6_Code1
[[sergiopnvds@jarvis HW6]$ mpiexec -np 4 ./HW6_Code1 HW6_Code1.dat
Output file correctnes verified
Elepsed time: 0.002452 seconds
[[sergiopnvds@jarvis HW6]$ od -td -v HW6_Code1.dat
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
[sergiopnvds@jarvis HW6]$ █
```

3.3. Results

If we observe the results presented in the graph bellow, it is clear that jarvis executes this parallel code faster because of its parallel features.



4. Part 2 - Benchmak

4.1. Develop process

In this second part of the homework, the problem presented requires to do some modifications in the part one code. Equally to the first part, each process calculates its offset, open the file generates the random chars and write the file. After that, each process reads the file at the same time, measure total times, delete the temporal file and write the output file.

Similarly to the first part, introducing how I check different set of possible errors that can appear in the code, I used the MPI_CHECK() function provided to check handle errors in all the MPI Statements. Additionally, I check if the input provided has the file where write the results.

Describing step by step how the code:

- Check if the the number of Megabytes and output file is provided
- Share with all process if there is a file error input
- If there is not error
    - Share the bytes that each process has to write
    - Take initial writing time
    - Calculate the offset
    - Open the output file for writing
    - Each process writes its buffer ten times
    - Close the file
    - Take end writing time
    - Take initial reading time
    - Each process opens the file for reading
    - Each process reads its part of the file
    - Take end reading time
    - Delete the temporal file
    - MPI finalize
    - Close file
    - Write the total times in the output file
- If and error
    - Return "Input error"
    - MPI finalize
    - Return elapsed time
- Exit

4.2. Performance Analysis

Because of the impossibility to use Jarvis, it was down. I tested my code using AWS creating a EC2 instance using M1.xlarge with 4 virtual cores. Additionally, I used a parallel file system with Amazon Elastic File System.

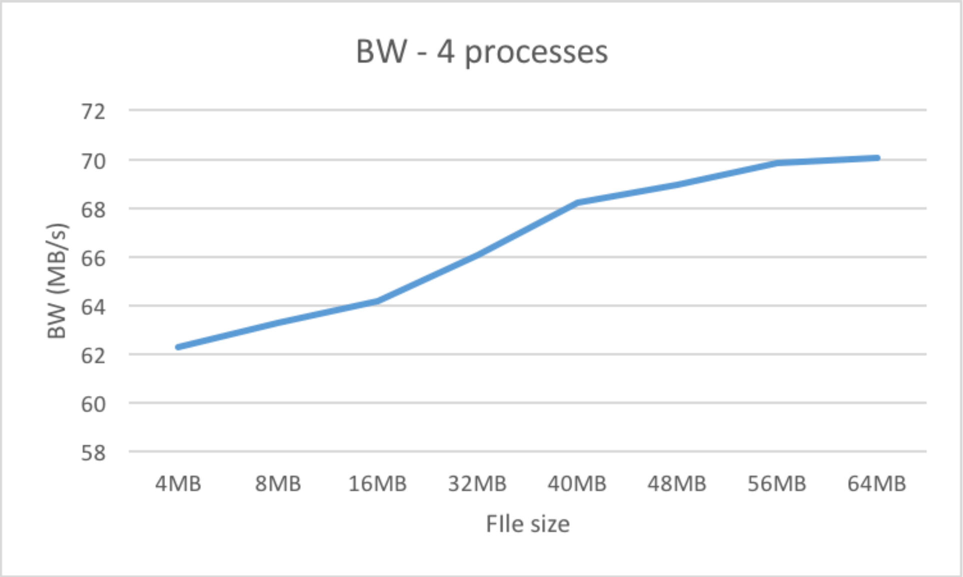
The scenario presented above gave me the necessary environment to test the bandwidth for each number of processes required.

4.2.1. Number of processes = 4

In this first case I test the scenario with 4 processes. In the table bellow we can observe the data obtained for different file sizes:

MB per Rank	1MB	2MB	4MB	8MB	10MB	12MB	14MB	16MB
MB in total	4MB	8MB	16MB	32MB	40MB	48MB	56MB	64MB
BW per Rank	15.5700	15.8224	16.0396	16.5225	17.0488	17.2398	17.4637	17.5081
BW in total	62.2823	63.2898	64.1585	66.0900	68.1954	68.9594	69.8549	70.0327

In the graph bellow we can see that as well as file size increases, the bandwidth increases too because the system has enough bandwidth to process the file.

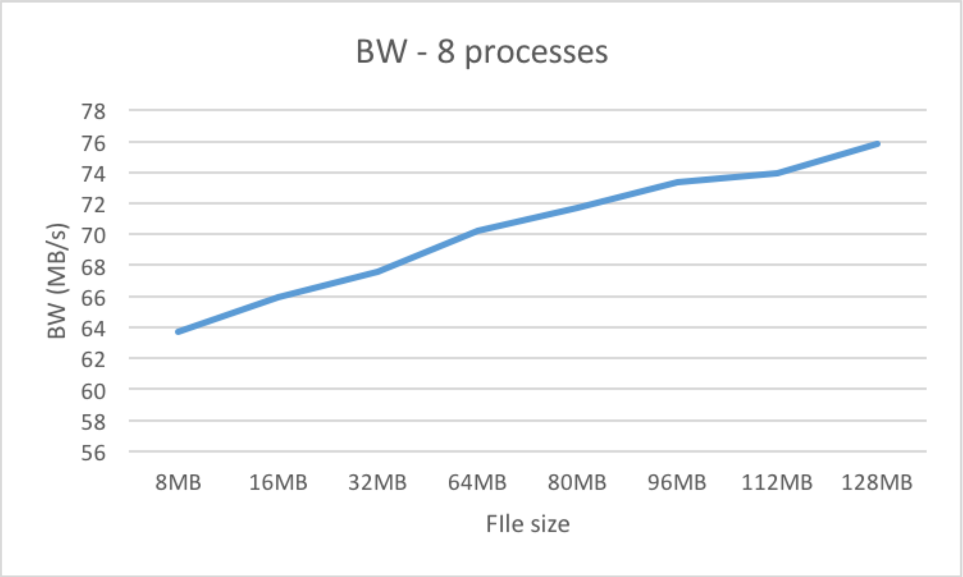


4.2.2. Number of processes = 8

In this second case I test the scenario with 8 processes. In the table bellow we can observe the data obtained for different file sizes:

MB per rank	1MB	2MB	4MB	8MB	10MB	12MB	14MB	16MB
MB in total	8MB	16MB	32MB	64MB	80MB	96MB	112MB	128MB
BW per rank	7.9596	8.2375	8.4433	8.7787	8.9653	9.1692	9.2376	9.4827
BW in total	63.6768	65.9132	67.5471	70.2297	71.7225	73.3536	73.9011	75.8617

Equally to the part before, in the graph bellow we can see that as well as file size increases, the bandwidth increases too because the system has enough bandwidth to process the file.

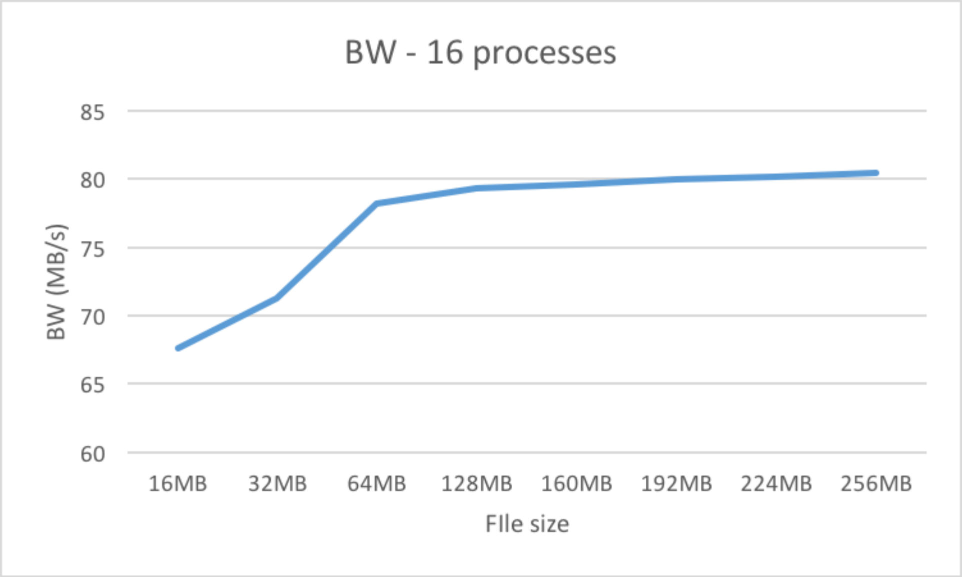


4.2.3. Number of processes = 16

In this third case I test the scenario with 16 processes. In the table bellow we can observe the data obtained for different file sizes:

MB per rank	1MB	2MB	4MB	8MB	10MB	12MB	14MB	16MB
MB in total	8MB	16MB	32MB	64MB	80MB	96MB	112MB	128MB
BW per rank	7.9596	8.2375	8.4433	8.7787	8.9653	9.1692	9.2376	9.4827
BW in total	63.6768	65.9	67.5471	70.2297	71.7225	73.5436	73.9011	75.8617

In this occasion in the graph bellow we can see that as well as file size increases, the bandwidth increases too because the system has enough bandwidth to process the file.

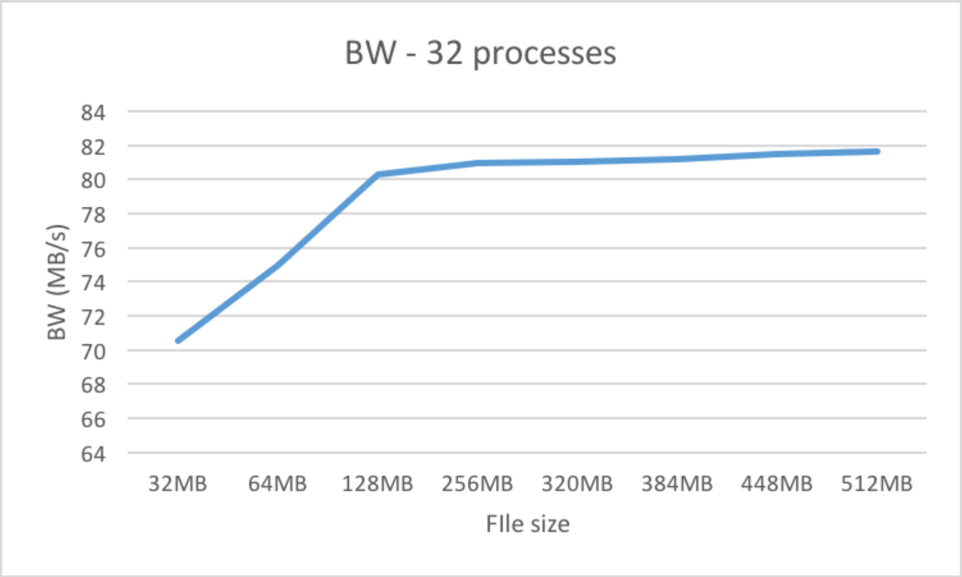


4.2.4. Number of processes = 32

In this forth case I test the scenario with 32 processes. In the table bellow we can observe the data obtained for different file sizes:

MB per rank	1MB	2MB	4MB	8MB	10MB	12MB	14MB	16MB
MB in total	32MB	64MB	128MB	256MB	320MB	384MB	448MB	512MB
BW per rank	2.2051	2.3423	2.5087	2.5294	2.5329	2.5378	2.5457	2.5505
BW in total	70.5635	74.9545	80.2787	80.9425	81.0547	81.2102	81.4636	81.6173

As in the first case, in the graph bellow we can see that as well as file size increases, the bandwidth increases too because the system has enough bandwidth to process the file.

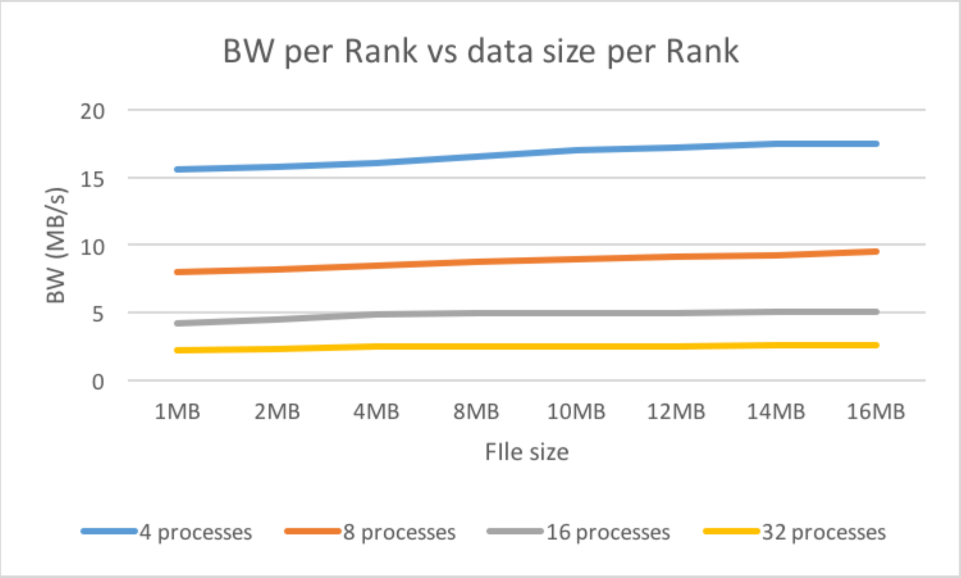


4.2.5. BW per Rank vs BW per Rank

In this fifth case I compare the bandwidth per Rank with the data size that each Rank executes with 4, 8, 16 and 332 processes. In the table bellow we can observe the data obtained for different file sizes:

MB per rank	1MB	2MB	4MB	8MB	10MB	12MB	14MB	16MB
BW per Rank in 4 processes	15.5700	15.8224	16.0396	16.5225	17.0488	17.2398	17.4637	17.5081
BW per Rank in 8 processes	7.9596	8.237	8.4433	8.7787	8.9653	9.1692	9.2376	9.4827
BW per Rank in 16 processes	4.2241	4.4537	4.8861	4.9571	4.9784	4.9984	5.0136	5.0307
BW per Rank in 32 processes	2.2051	2.3423	2.5087	2.5294	2.5329	2.5378	2.5457	2.5505

In the table bellow we can see how as more processes the system have as less bandwidth per Rank it has. I choose this graph because is the only form to show a graph with all the four possible configurations (4, 8, 16 and 32 processes) due here they share the same scale and range values.



5. Conclusion

Thanks to MPI-IO it is possible to improve the efficiency over codes that perform a big amount of write and read operations. In this kind of operations, it is necessary to guarantee a bandwidth when a cluster is stressed because as more processes the app executes as less bandwidth each process has.

Likewise, it is important to highlight the importance that how MPI-IO works. Each process has to execute a part of code at the same time and perform the data operation parallel using a particular offset. Thanks to MPI-IO each process works in a different part in the same file avoiding collisions. Therefore, it is necessary to make a deep study of the code because the key is to divide the target file perfectly to ensure the operations correctness. So the engineer has to be aware of the application features (loops, exchange of data, data synchronization, etc..) to implement an adequate parallelization.

6. References

[1] MPI IO
http://beige.ucs.indiana.edu/I590/node86.html

[2] Introduction to Parallel I/O
https://www.tacc.utexas.edu/documents/13601/900558/MPI-IO-Final.pdf/eea9d7d3-4b81-471c-b244-41498070e35d

[3] Introduction to Parallel I/O
http://wgropp.cs.illinois.edu/courses/cs598-s16/lectures/lecture32.pdf

[4] Parallel I/O
http://www.cac.cornell.edu/education/training/ParallelMay2012/ParallelIOMay2012.pdf

[5] Open MPI
https://www.open-mpi.org/