Assignment 4: MPI Collectives and MPI-IO

Mihai-Gabriel Robescu (03709201)
mihai.robescu@tum.de
Yi JU (03691953)
yi.ju@tum.de

Hsieh Yi-Han
kimihsieh.tw@tum.de
Koushik Karmakar (03704659)
koushik.karmakar@tum.de

3. MPI Collectives

The performance plots

Fig 3.1 shows that with fixed process counts, the execution time increases along with increasing sizes except for process count 64 where the average execution time decreases initially but then increases alongside increasing input size.

From Fig 3.2, we can conclude that the execution time keeps increasing as we increase the process count except for input size 4096x4096 and 8192x8192. For 4096x4096, the execution time can be said to have a very weak scalability against increasing process counts when we increase the process count approximately above 30. For input size 8192x8192, the time performance follows a strong scalability against increasing process count.

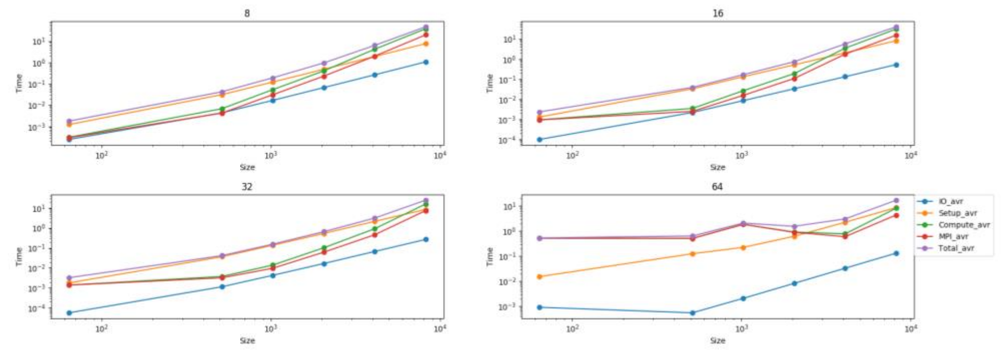


Fig 3.1 Times with fixed process counts and varying size of input for Haswell node

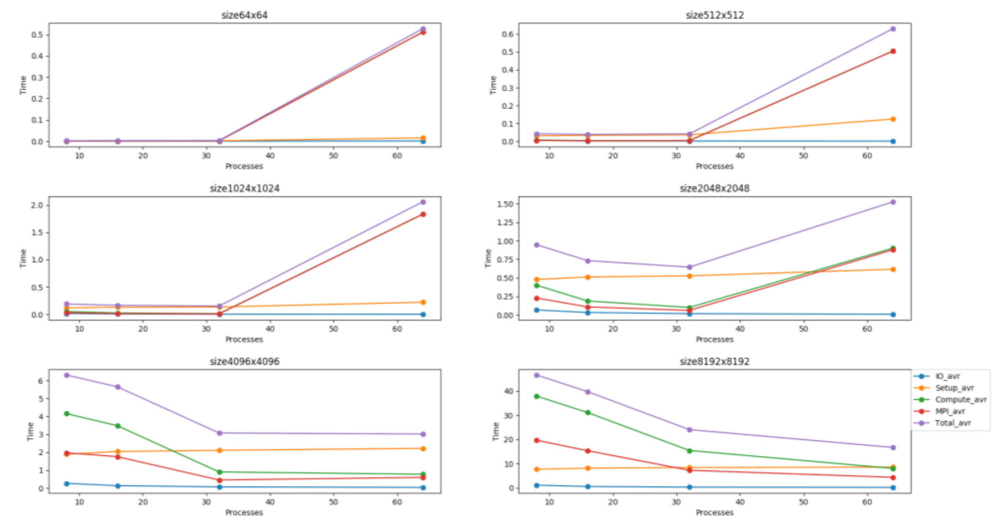


Fig. 3.2 Times with fixed input sets and varying process counts for Haswell node

3.2 Questions

1. Which patterns were identified and replaced with collective communication in the code? Explain. (8 points)

The first communication from the root process to all other processes was modified to collective communication, as it was one process broadcasting *&rows* and *&columns* to all other processes. The second communication pattern was modified using *MPI_Scatter* as each process had to take their part of the matrix and the vector from the root process and then process it later. Moreover, two more *for* instructions have been deleted as *MPI_Scatter* creates also a copy on the root process. The point-to-point communications during the Gaussian Elimination were not modified as they sent the same data to just a few processes and maybe not even from the root process. The last pattern has also been modified by implementing the *MPI_Gather* as it had to collect the solution from all the processes. As in the case of *MPI_Scatter* two more *for* instructions have been removed as the *MPI_Gather* also created a copy on the root process.

2. Were you able to identify any potential for overlap and used any non-blocking collectives? (Use Vampir)(6 points)

As during the Gaussian elimination no modifications were made, there was no need of non-blocking collectives. The computation time saved by using non-blocking collectives for the already modified patterns would not have been much bigger.

3. Was there any measurable performance or scalability improvement as a result of these changes? (6 points)

From the graphics it resulted that there was a big improvement by using *MPI_Collectives*.

4. Is the resulting code easier to understand and maintain after the changes? Why? (4 points)

The resulting code is much easier to understand. As it was stated before, some instructions were removed like the *for* instructions to copy the data for the root process. Moreover, by using *MPI_Scatter, MPI_Bcast, MPI_Gather*, there were removed 1 pair of

MPI_Send and *MPI_Recv* per instruction, thus being much easier to follow were the data was going.

4. MPI Parallel IO

The performance plots

Fig 4.1 shows that even with different processes counts, running time will increase along with increasing size. Fig 4.2 shows that the time performance has strong scalability especially in large size from 2048 to 8192.

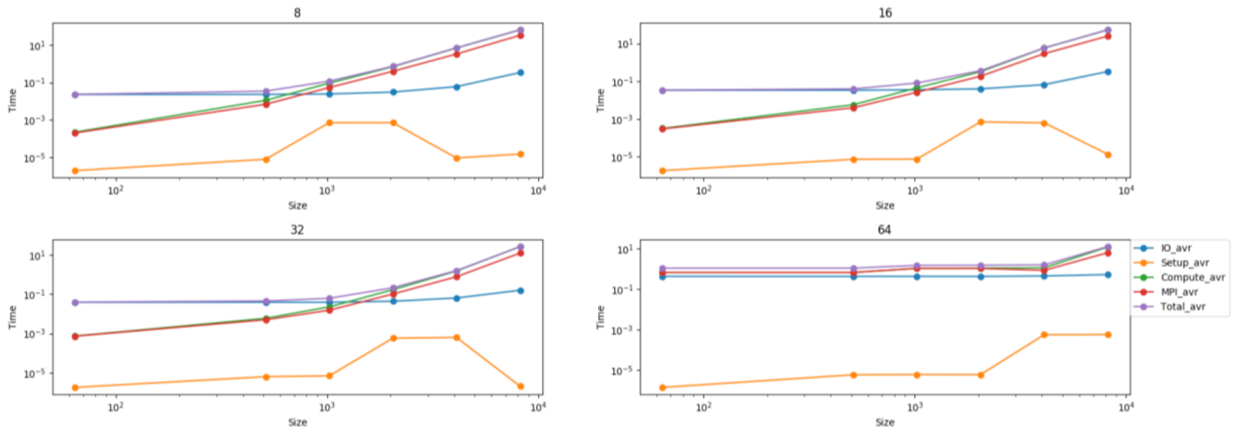


Fig 4.1 Times with fixed process counts and varying size of input for Haswell node

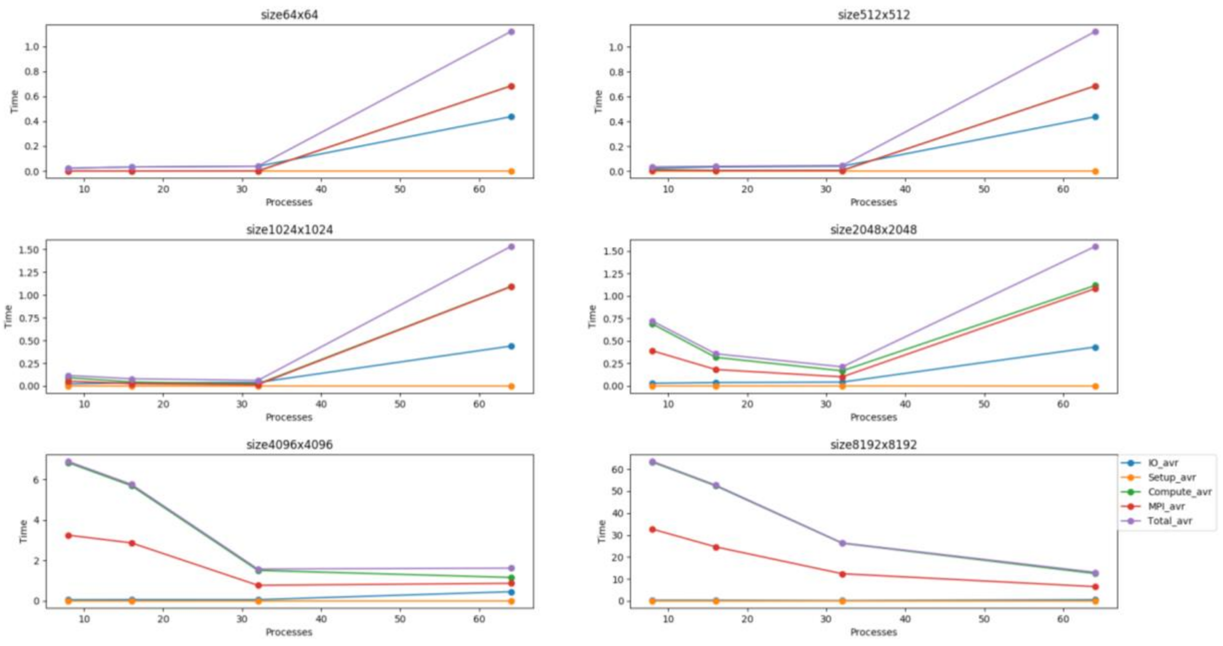


Fig. 4.2 Times with fixed input sets and varying process counts for Haswell node

4.2 Questions

1. Which MPI-IO operations were applied to transform the code? Explain your choices. (8 points)

I use *MPI_File_read(write)_(at)_all.* Firstly, because all processes only get the data from the same file together, it is best choice to use collective operations. Secondly, each process doesn't capture the same data but the contiguous blocks which store data in the matrix and the two vectors, so I use "at" to set the offset to let each process capture the desired block orderly. Thirdly, for writing, to let all processes write the solution blocks into same file orderly, I use *MPI_File_write_at_all*. Some parameters are constant in all processes, like rows and columns, so I use *MPI_File_read_all.*

2. What is "Data Sieving" (2 points) and "2-Phase IO" (2 points)? How do they help improve IO performance? (2 points)

To begin with, making many requests to get access to the file is time consuming and has some risks. Data Sieving and 2-Phase IO help us to reduce those problems.

Data Sieving creates a large, temporary buffer to store the whole input data sets which may be contiguous or noncontiguous, instead of reading these data sets separately. Data Sieving captures the useful data sets and transfers these noncontiguous data sets into contiguous, that is without gaps, and then puts the contiguous data sets into user's buffer. Although creating the temporary buffer may store many holes due to the noncontiguous data sets and be storage consuming, reading the data sets separately are much more cost than these side effects.

2-Phase IO, also called collective buffering, makes all processes read the entire file together instead of calling many requests individually. In the first phase, also called shuffle phase, processes access data and merged into larger ones. In the second phase, processes redistribute data contiguously to final user's desired location. Although creating the temporary buffer may store many holes due to the noncontiguous data sets and be storage consuming, reading the data sets separately are much more cost than these side effects. The cost of merging small data sets into large one and the communication for redistribution are very small but I/O time reduces significantly.

3. Was the original implementation scalable in terms of IO performance? (2 points)

No. Based on Fig 4.3, for the original implementation, because rank0 is the only one which is responsible for IO, increasing number of process doesn't help at all.

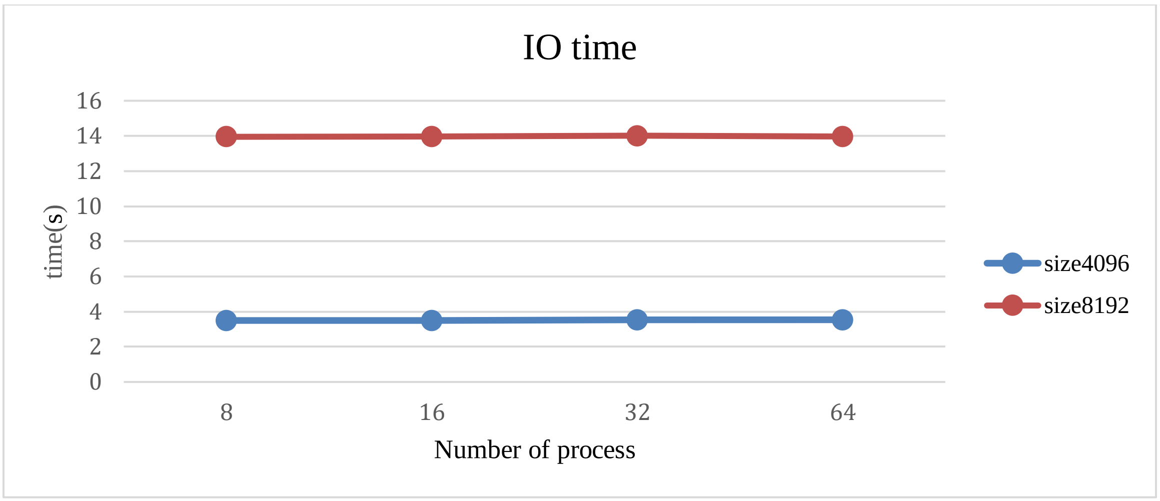
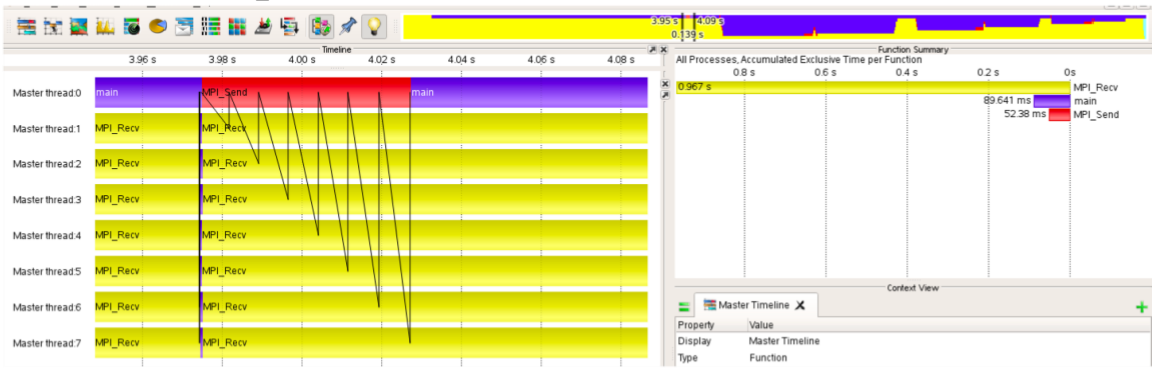


Fig. 4.3 IO times versus varying process counts in the original implementation

4. Was the original implementation scalable in terms of RAM storage? (2 points)
   Yes, the larger size we input, the more RAM storage we need.

5. How much of the communication in the application was replaced or eliminated with MPI-IO? (Use Vampir) (6 points)
   We save three communication.
   1. The communication of sending input matrix form rank_0 to other ranks.

Source Code without MPI_IO Read



MPI_IO Read matrix

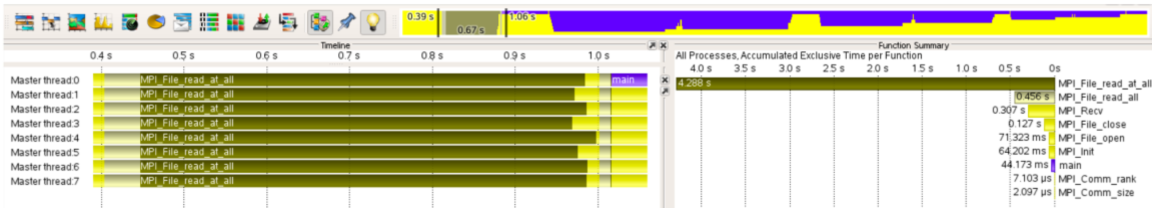
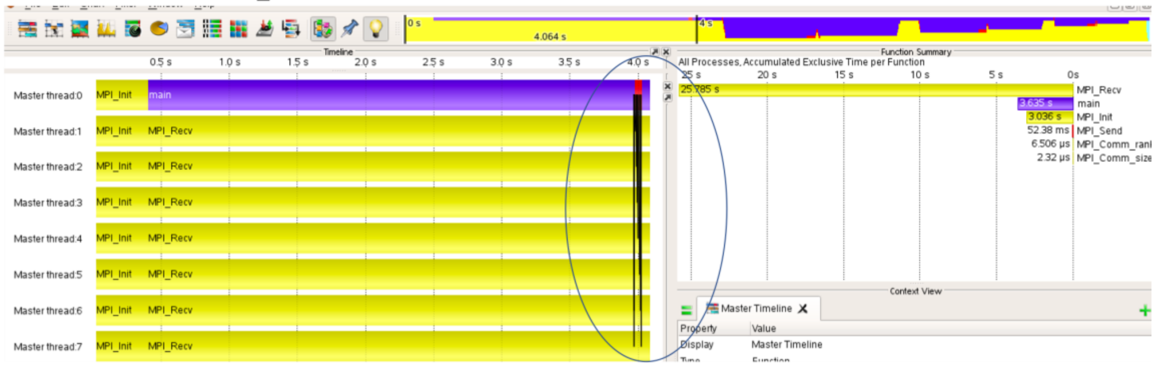


Fig. 4.4 The comparison of reading input matrix between Without MPI_IO and With MPI_IO.

   2. The communication of sending input vector form rank_0 to other ranks.

Source Code without MPI_IO read



MPI_IO read vector

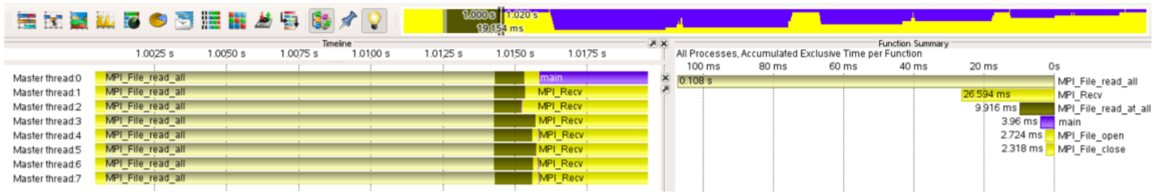
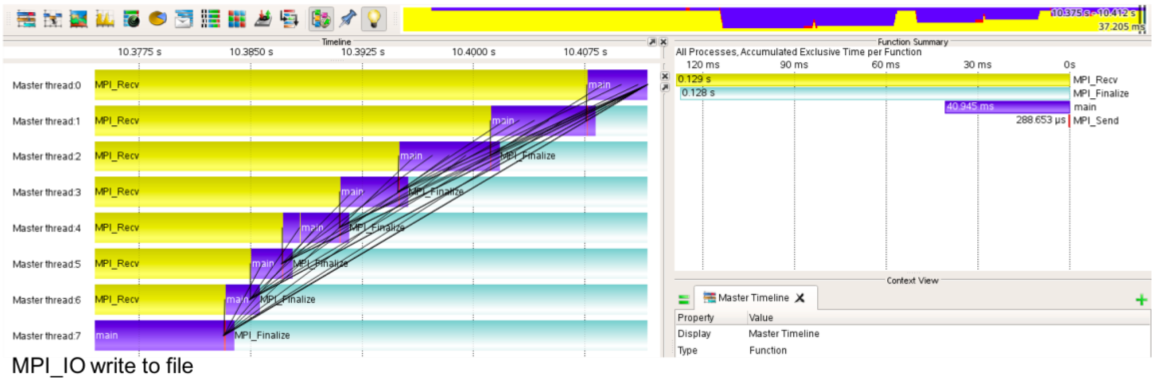


Fig. 4.5 The comparison of reading input vector between Without MPI_IO and With MPI_IO.

3. The communication of sending input local solution from other ranks to rank_0.

Source Code without MPI_IO write to file



MPI_IO write to file

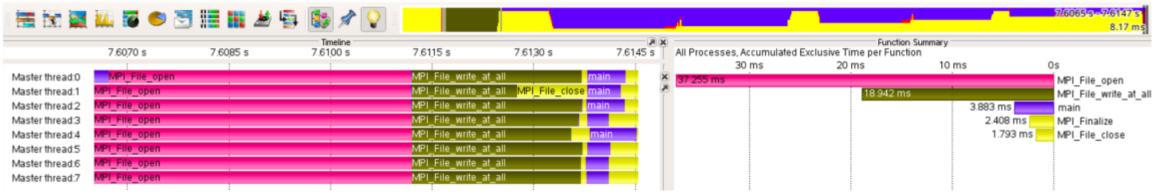


Fig. 4.6 The comparison of writing solution to file between Without MPI_IO and With MPI_IO.

6. Were there any performance improvements due to the change to MPI-IO? (6 points)

Yes, for large input-size, the improvement of IO is significant. Because we only modify IO, the computing time should be the same as original implementation. Similarly, the communication in computing dominate the MPI time in large size, so the MPI speedup is limited. To sum up, because of the significant speedup of IO, the total time decreases a lot for large size.

Table 4.1 Speedup of MPI_IO versus the baseline for Haswell nodes.

Size	Num_process	IO	Setup	Compute	MPI	Total
size64x64	8	0.151	1187.625	2.529	2.675	0.147
size64x64	16	0.085	917.241	4.321	4.519	0.096
size64x64	32	0.087	1289.655	3.212	3.254	0.123
size64x64	64	0.007	229826.472	1.217	1.217	1.028
size512x512	8	2.522	6431.938	1.073	1.131	2.068
size512x512	16	1.732	7709.391	1.213	1.305	1.699
size512x512	32	1.550	9578.542	1.147	1.179	1.540
size512x512	64	0.138	65301.819	1.213	1.213	1.083
size1024x1024	8	9.251	278.858	1.034	1.056	2.788
size1024x1024	16	6.421	29447.975	1.140	1.205	3.554
size1024x1024	32	5.770	32190.929	1.114	1.189	4.172
size1024x1024	64	0.506	89951.301	2.940	2.947	2.454
size2048x2048	8	**28.764**	1106.481	0.975	0.974	**2.183**
size2048x2048	16	**22.724**	1239.389	1.040	1.068	**3.478**
size2048x2048	32	**20.161**	1530.054	1.097	1.159	**5.146**
size2048x2048	64	**2.032**	203024.885	1.423	1.437	**1.816**
size4096x4096	8	**58.639**	337374.053	1.019	1.020	**1.533**
size4096x4096	16	**53.467**	5487.957	1.005	1.001	**1.620**
size4096x4096	32	**54.903**	5586.756	1.063	1.060	**3.319**
size4096x4096	64	**7.721**	6954.017	1.157	1.208	**3.275**
size8192x8192	8	**41.655**	826929.205	1.004	1.007	**1.225**
size8192x8192	16	**43.372**	1012018.444	0.941	0.973	**1.210**
size8192x8192	32	**87.866**	6786477.848	0.982	0.975	**1.523**
size8192x8192	64	**25.930**	25037.098	1.046	1.056	**2.133**

Table. Assignment distribution

Task / **Name**	3	4	script	Report
Mihai-Gabriel Robescu	R	P	A	R
Hsieh Yi-Han	P	R	R	R

Yi JU	P	R	R	R
Koushik Karmakar	R	P	A	R

P = **P**articipate **R** = be **R**esponsible to **A** = **A**bsent