Assignment 4
MPI Collectives and MPI-IO

Smith Agarwal
Shyam Arumugaswamy
Siddhesh Kandarkar

January 23, 2019

3 MPI Collectives

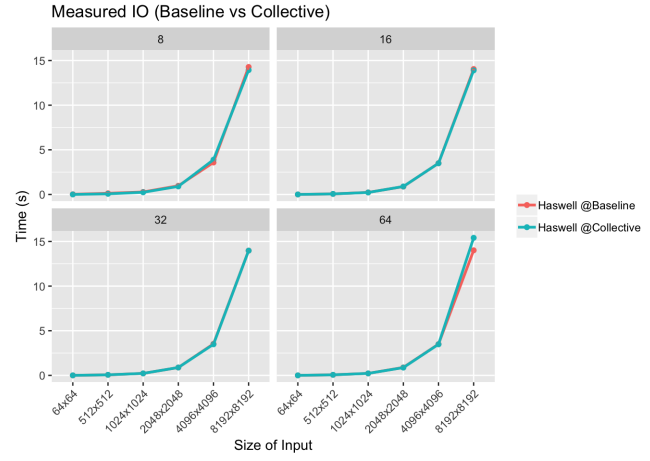
3.1 Required submission files

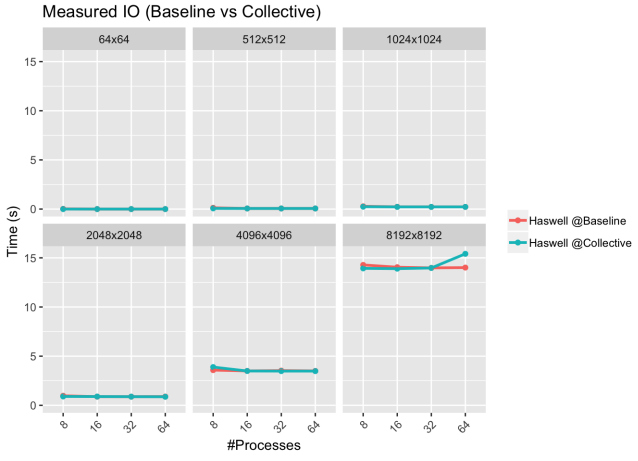
1. **The updated gauss.c**

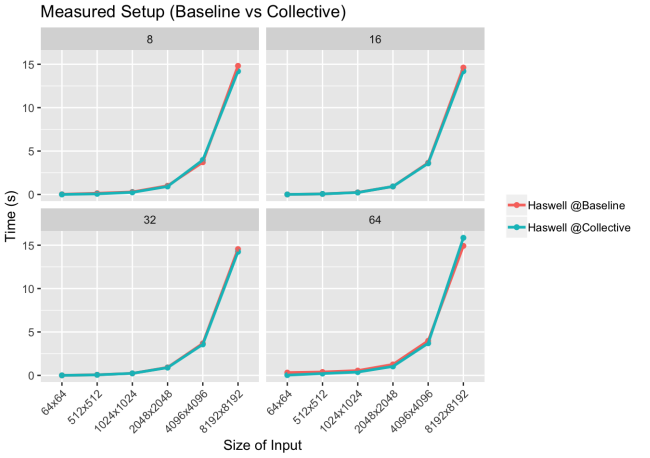
   Refer gauss_collective.c file from code scripts of Task 3.1

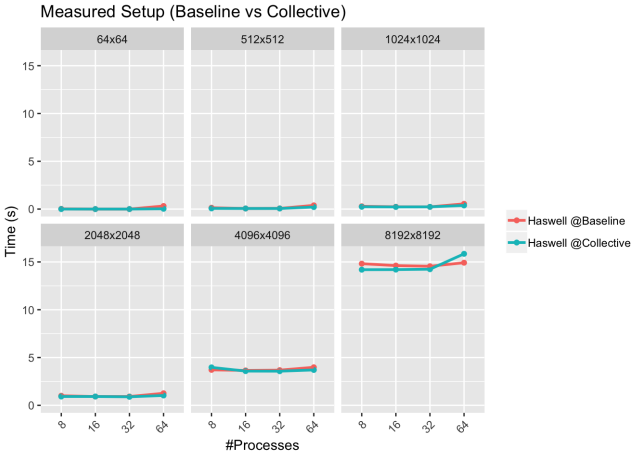
2. **The performance plots and description in the report.**

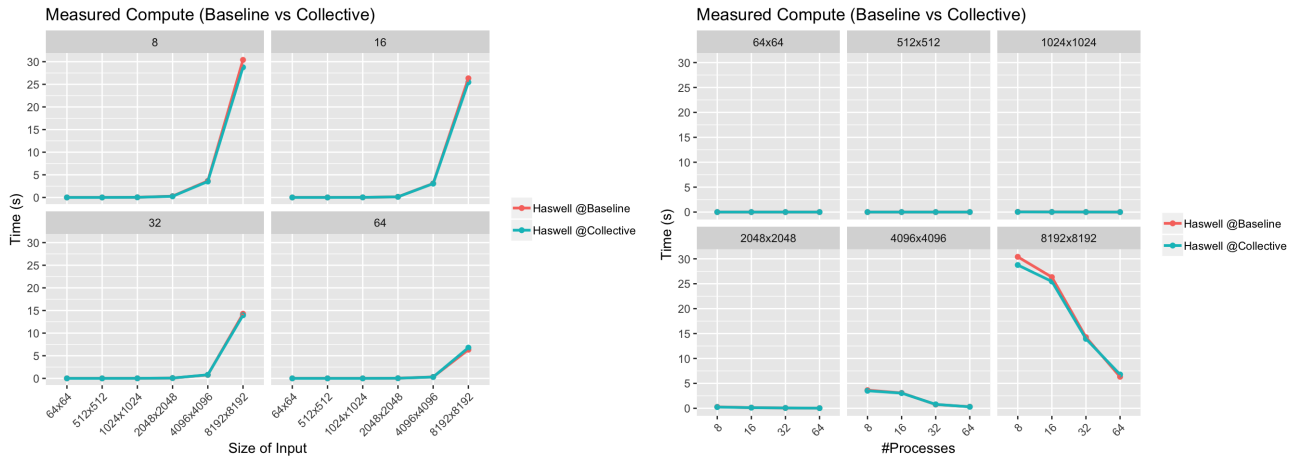
   The various performance plots for MPI Collective communication compared with baseline are:

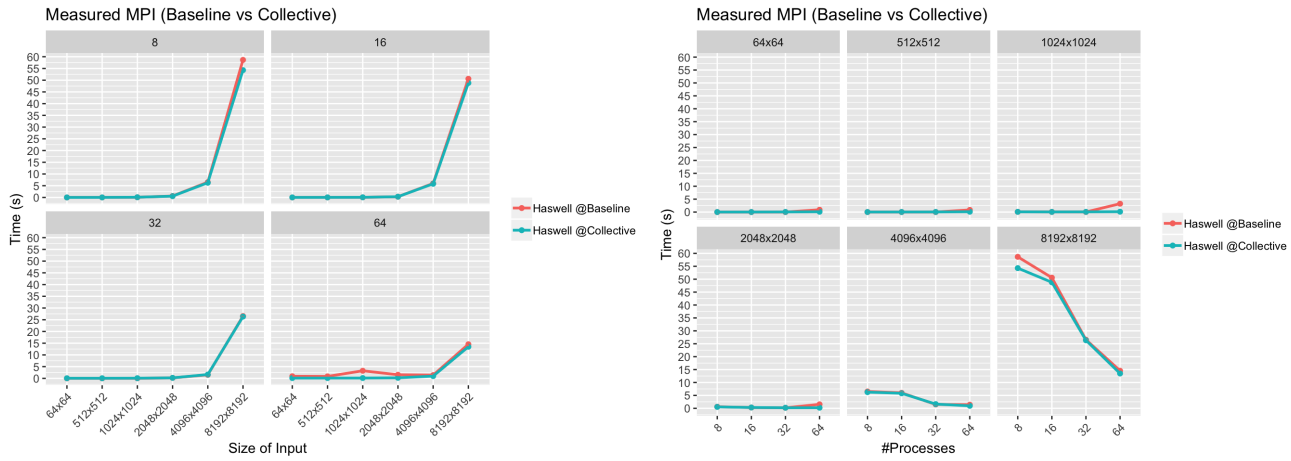


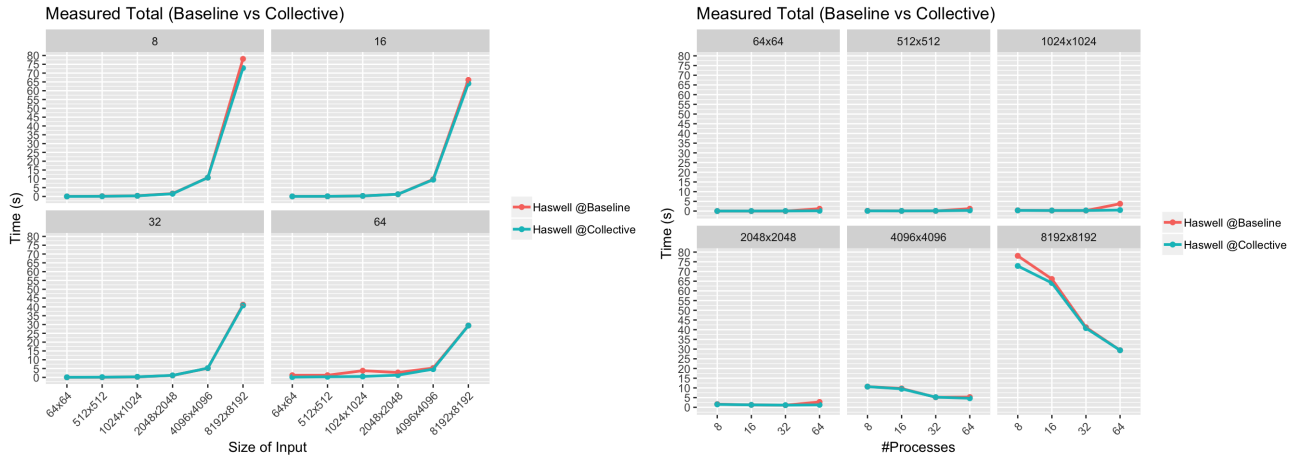












3.2 Questions

1. **Which patterns were identified and replaced with collective communication in the code? Explain.**

   The various places which could be replaced with collective communication are :

   (a) For Baseline setup, `MPI_Send` was used for rank 0 and `MPI_Recv` for others
   These were replaced by `MPI_Bcast` which is used for sending from same memory and `MPI_Scatter` which is from shifted memory

```
if(rank == 0) {
        for(i = 1; i < size; i++){
            MPI_Send(&rows, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
            MPI_Send(&columns, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
        }
    } else {
        MPI_Recv(&rows, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
        MPI_Recv(&columns, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
    }
```

```
    MPI_Bcast(&rows,1,MPI_INT,0,MPI_COMM_WORLD);
    MPI_Bcast(&columns,1,MPI_INT,0,MPI_COMM_WORLD);

    MPI_Scatter(matrix_1D_mapped,local_block_size *
  rows,MPI_DOUBLE,matrix_local_block,local_block_size * rows,MPI_DOUBLE,0,MPI_COMM_WORLD);
    MPI_Scatter(rhs,local_block_size,MPI_DOUBLE,rhs_local_block,local_block_size ,MPI_DOUBLE,0
  ,MPI_COMM_WORLD);
```

(b) For Baseline, we used separate computation logic for processes from 0 to rank-1 and then from rank+1 to size.
This can be replaced by `MPI_Bcast`

```
    for (process = 0; process < rank; process++){
        mpi_start = MPI_Wtime();
        MPI_Send( accumulation_buffer, (2 *
  local_block_size), MPI_DOUBLE, process, rank, MPI_COMM_WORLD);
        mpi_time += MPI_Wtime() - mpi_start;
    }


    for (process = (rank + 1); process < size; process++) {
        pivots[0] = (double) rank;
        mpi_start = MPI_Wtime();
        MPI_Send( pivots, (local_block_size * rows + local_block_size + 1
  ), MPI_DOUBLE, process, rank, MPI_COMM_WORLD);
        mpi_time += MPI_Wtime() - mpi_start;
    }
```

```
.
.
.
.....
MPI_Bcast(pivots, (local_block_size * rows + local_block_size + 1), MPI_DOUBLE, 0, new_comm);
.....
.
.
.
.....
MPI_Bcast(accumulation_buffer, (2 * local_block_size), MPI_DOUBLE,0,new_comm);
.....
.
.
.
```

(c) For Baseline, after computation we used `MPI_Recv` for rank 0 and `MPI_Send` for others.
This can be replaced by `MPI_Gather`

```
if(rank == 0) {
        for(i = 0; i < local_block_size; i++){
            solution[i] = solution_local_block[i];
        }
        mpi_start = MPI_Wtime();
        for(i = 1; i < size; i++){
            MPI_Recv(solution + (i * local_block_size), local_block_size, MPI_DOUBLE, i, 0
  , MPI_COMM_WORLD, &status);
        }
        mpi_time += MPI_Wtime() - mpi_start;
    } else {
        mpi_start = MPI_Wtime();
        MPI_Send(solution_local_block, local_block_size, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
        mpi_time += MPI_Wtime() - mpi_start;
    }
```

```
    MPI_Gather
  (solution_local_block,local_block_size ,MPI_DOUBLE,solution,local_block_size ,MPI_DOUBLE,0
  ,MPI_COMM_WORLD);
```

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

   As observed from the vampir plots below, there was no potential for overlap of communication and computation and so we didnt use any non-blocking collectives.

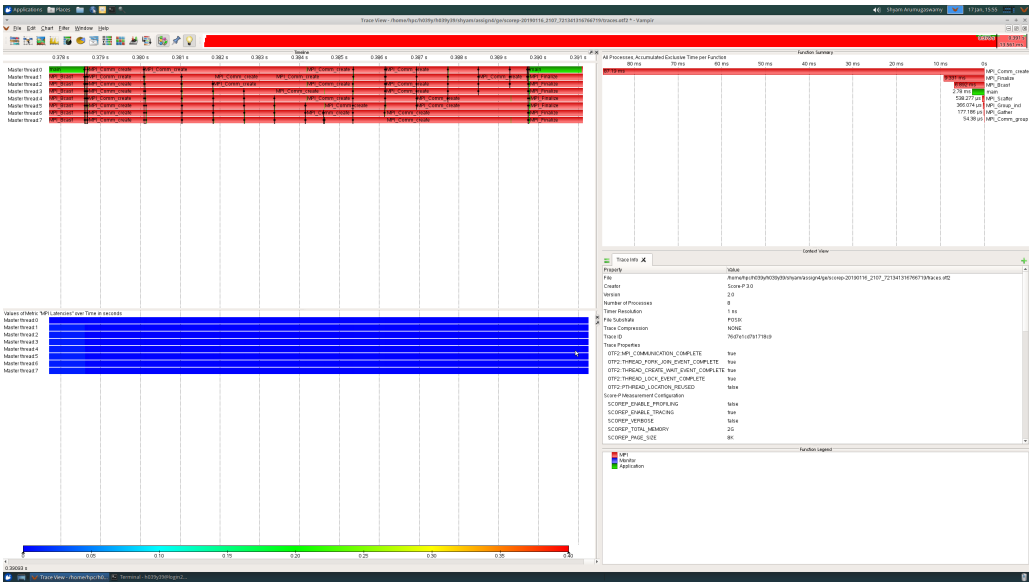


Figure : Vampir output for MPI Collective Haswell - 8 processes

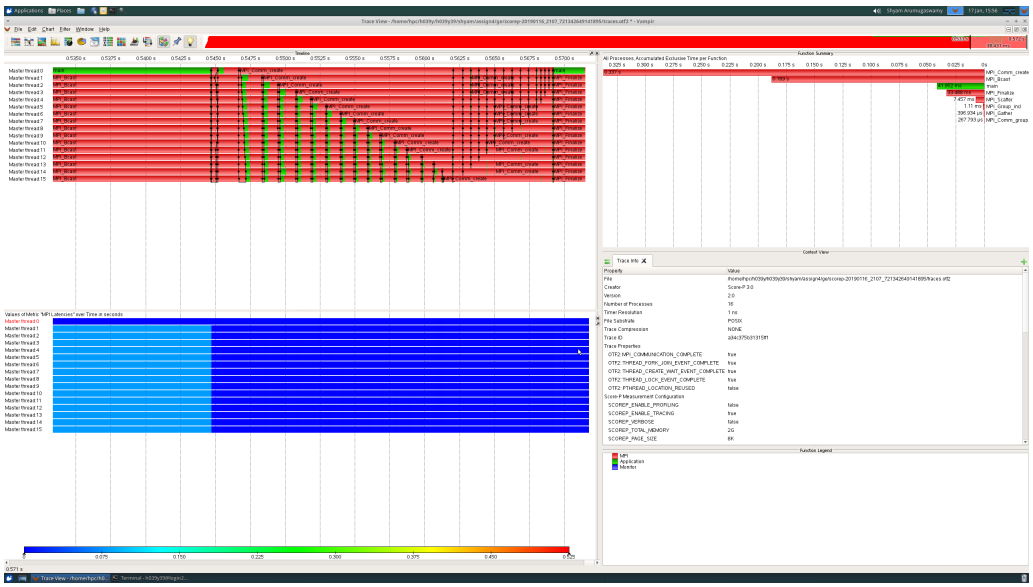


Figure : Vampir output for MPI Collective Haswell - 16 processes

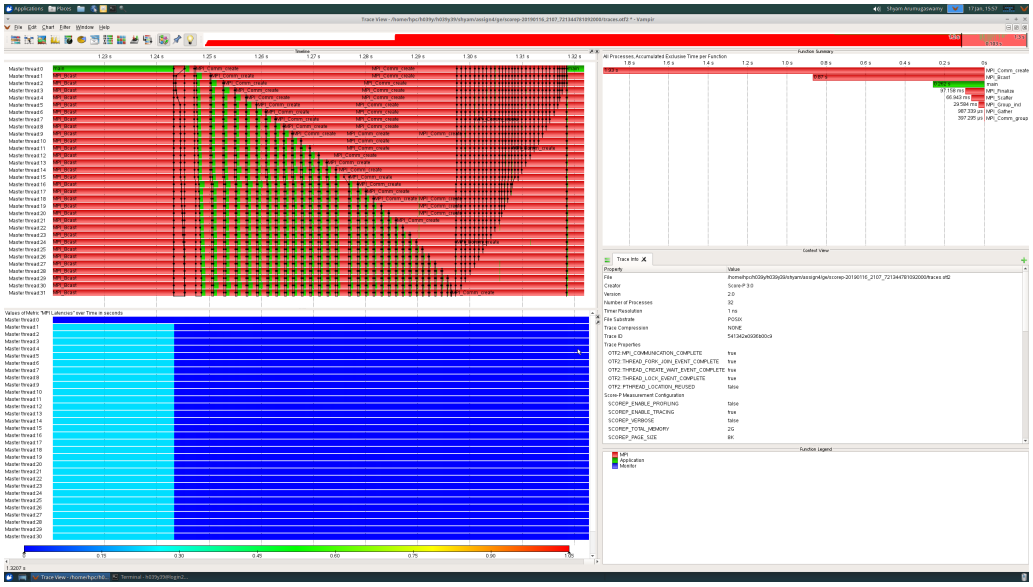


Figure : Vampir output for MPI Collective Haswell - 32 processes

3. **Was there any measurable performance or scalability improvement as a result of these changes?**

   From the performance plots shown in Section 3.1.2 , we observe that communication time doesn't dominate for large problem size and so there was no significant improvement in its performance. On the contrary, we observed improvement in performance for small problem size with large number of processes.

4. **Is the resulting code easier to understand and maintain after the changes? Why?**

   The resulting code is easier to understand because we have replaced the long loops of MPI_Send and MPI_Recv with single collective command of MPI_Bcast, thus making the code less dense. Code is read easily due to small individual chunks and communication is taken care of. The only disadvantage is to maintain different groups and communicators due to use of MPI_Group and MPI_COMM for various collective operations.

4 MPI Parallel IO

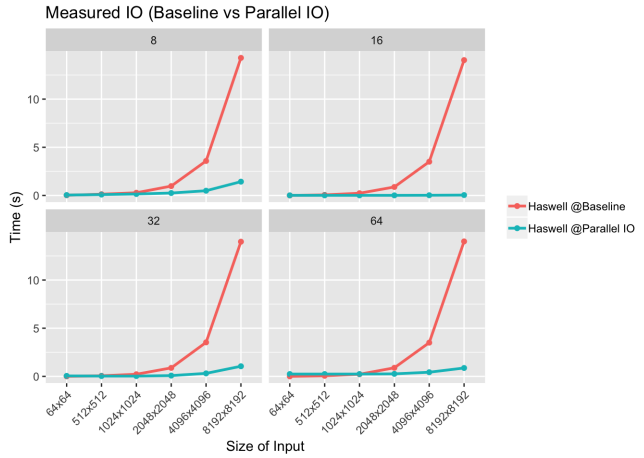
4.1 Required submission files

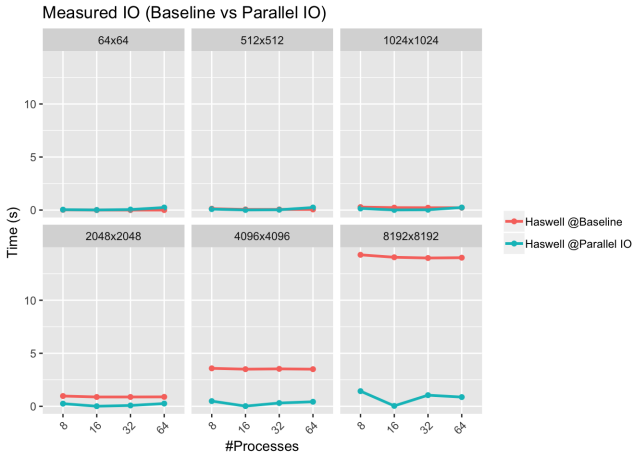
1. **The updated gauss.c file.**

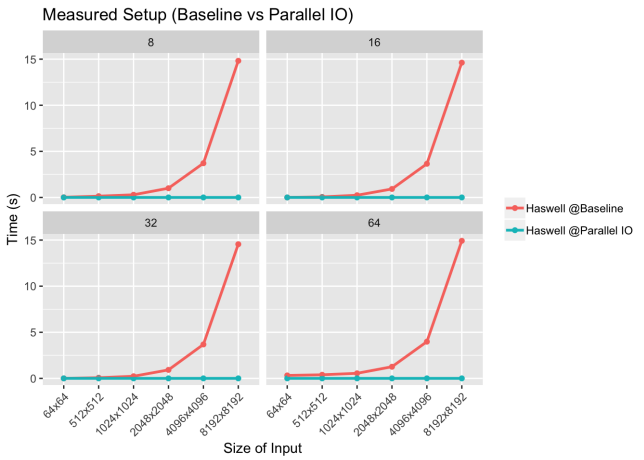
   Refer gauss_io.c file from code scripts of Task 4.1

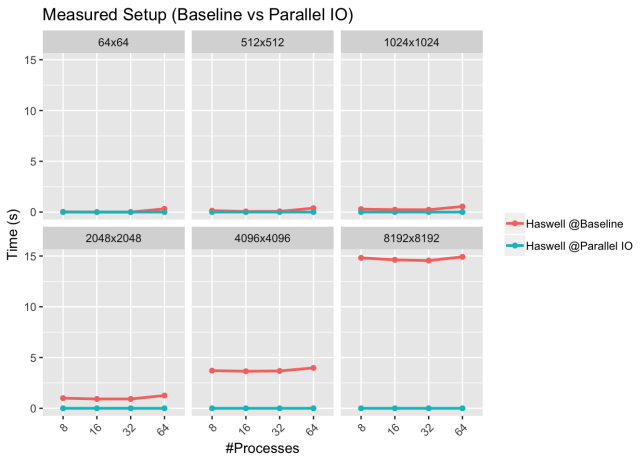
2. **The new performance plots and the description in the report.**

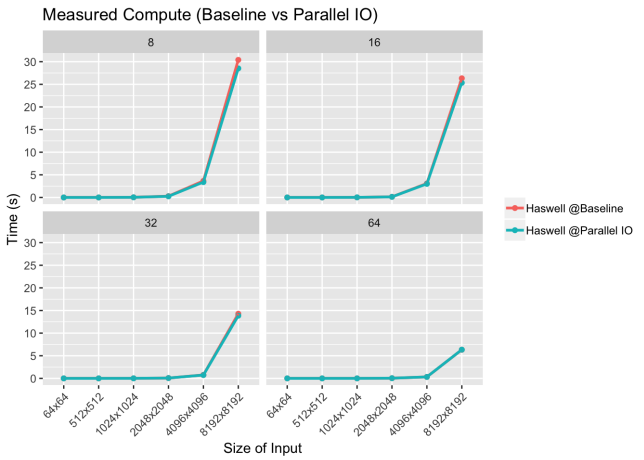
   The various performance plots for MPI Parallel IO communication compared with baseline are:

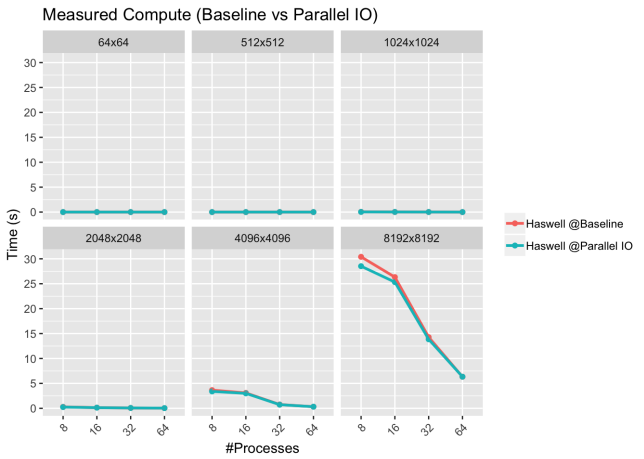


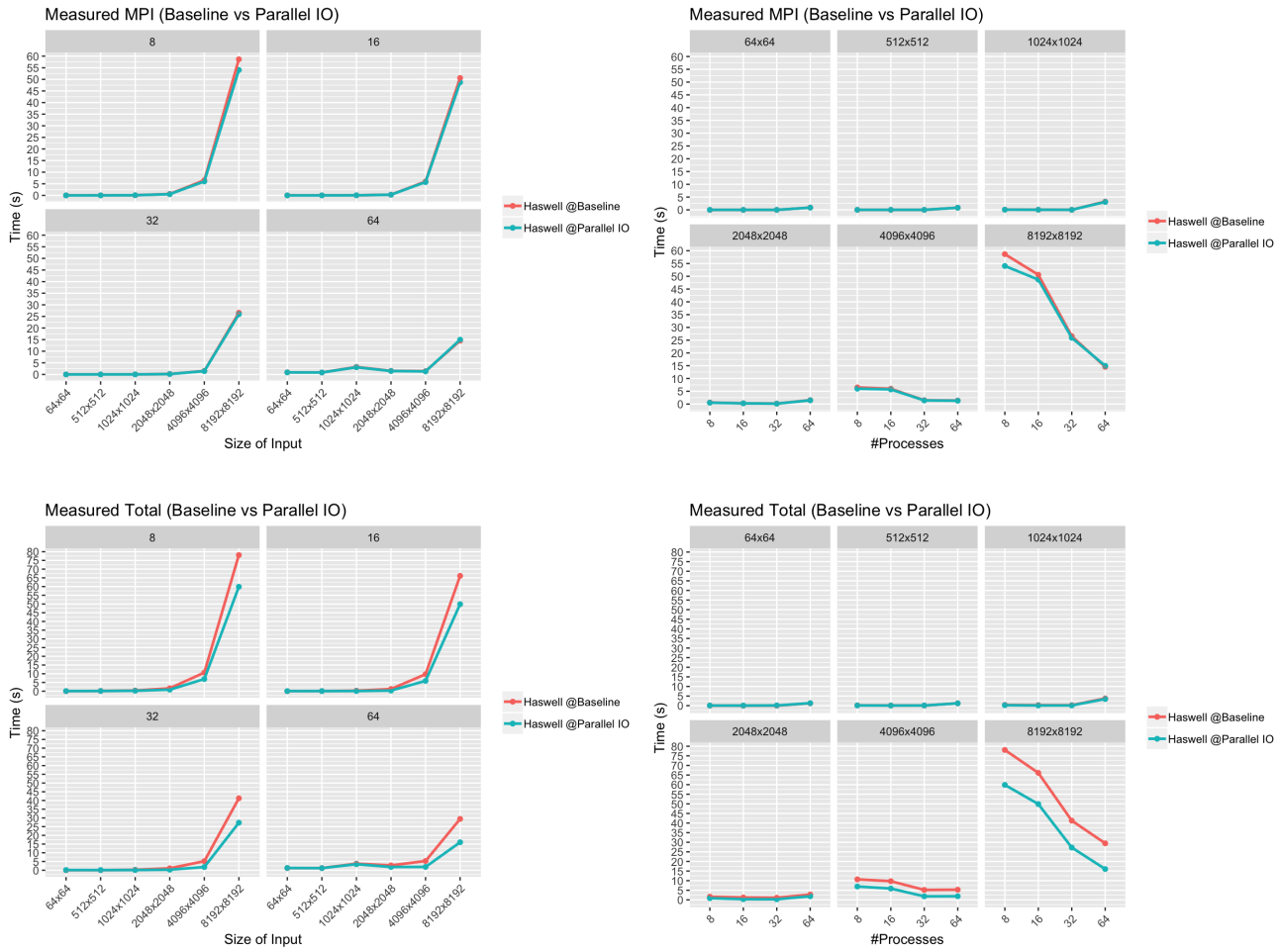












4.2 Questions

1. **Which MPI-IO operations were applied to transform the code? Explain your choices.**

   Following MPI-IO operations were applied to transform the code :

   - `MPI_File_open`: Opens a file.
   - `MPI_File_read`: Read a file using individual file pointer
   - `MPI_File_read_at`: Read a file using explicit offset
   - `MPI_File_close`: Closes a file.

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

   - **Data Sieving :**
     `[http://home.agh.edu.pl/~kzajac/ooc_tut/node6.html]`
     `[https://www.igi-global.com/dictionary/improved-checkpoint-using-the-effective-management-of-`
     `59596]`

     Data sieving is an optimization technique that enables an implementation to make a few large, contiguous I/O requests instead of large number of small, non-contiguous I/O requests to the file system.

     This improves the I/O performance by reducing the effect of high I/O latency, this is done by reducing the number of requests to the file system as much as possible.

     A potential drawback can be memory requirement, as large contiguous chunk of data is placed in temporary buffer to handle user requests, instead of accessing each portion of data separately for

each request. But, the advantage of using this technique outweighs the cost of reading extra data.

- **2-Phase IO :**
  [https://www.mcs.anl.gov/~thakur/papers/romio-coll.pdf]
  [http://www.nersc.gov/users/storage-and-file-systems/i-o-resources-for-scientific-applications
  optimizing-io-performance-for-lustre/]

  It breaks the IO operation into two stages. It is also known as "Collective buffering".

  For a collective read, a large chunk of contiguous data is read by the aggregators into a temporary buffer in the first stage. In the second stage, this data is moved from the buffer to its destination. The process is reverse for a collective write, firstly large chunks of data are aggregated in specific processes, which is then written to file.

  This method significantly reduces I/O time by making all file accesses large and contiguous. This advantage over-weighs the cost of inter-process communication for redistribution.

3. **Was the original implementation scalable in terms of IO performance?**

   The original implementation was not scalable in terms of IO performance.

   Since, only one process can perform IO operations , the time required for IO operations will remain same irrespective of the number of MPI processes. This also leads to increase in required time with respect increase in problem size.

4. **Was the original implementation scalable in terms of RAM storage?**

   The original implementation was not scalable in terms of RAM. This is mainly due to limit on RAM for single node. This problem would not change even if more number of processing units are added.

5. **How much of the communication in the application was replaced or eliminated with MPI-IO? (Use Vampir)**
   Operations such as fopen, fclose, fscanf, `MPI_Send`, `MPI_Receive` were replaced with `MPI_File_open`, `MPI_File_read`, `MPI_File_read_at`, `MPI_File_close`

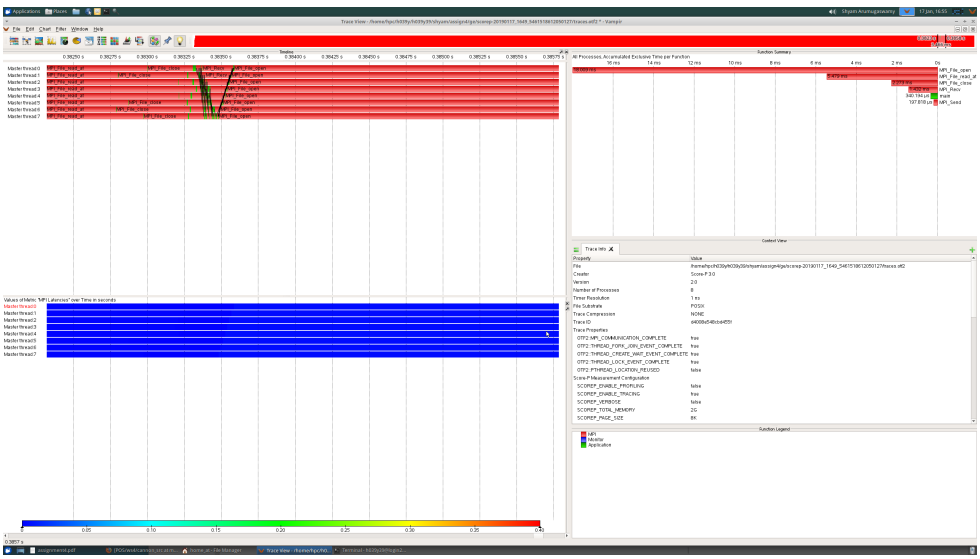


Figure : Vampir output for MPI Parallel IO Haswell - 8 processes

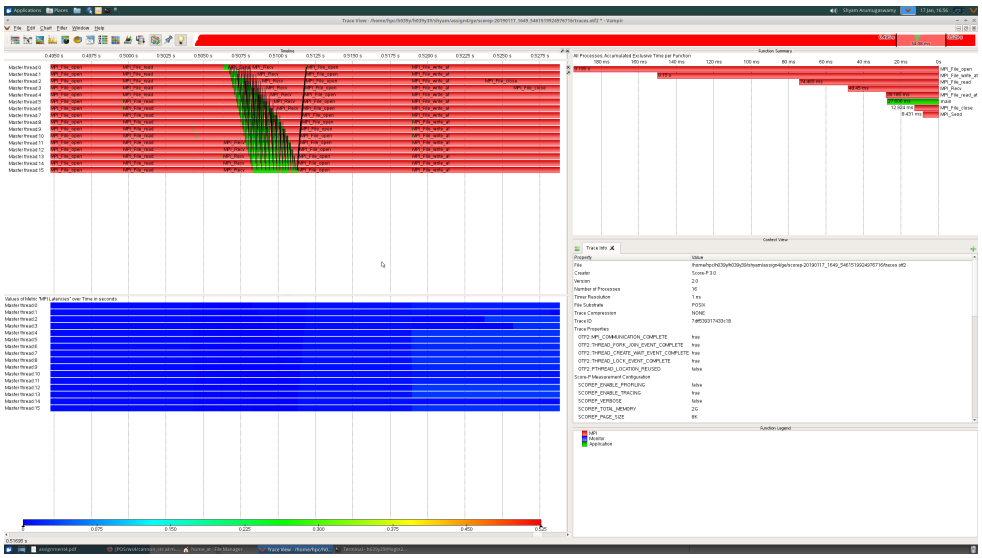


Figure : Vampir output for MPI Parallel IO Haswell - 16 processes

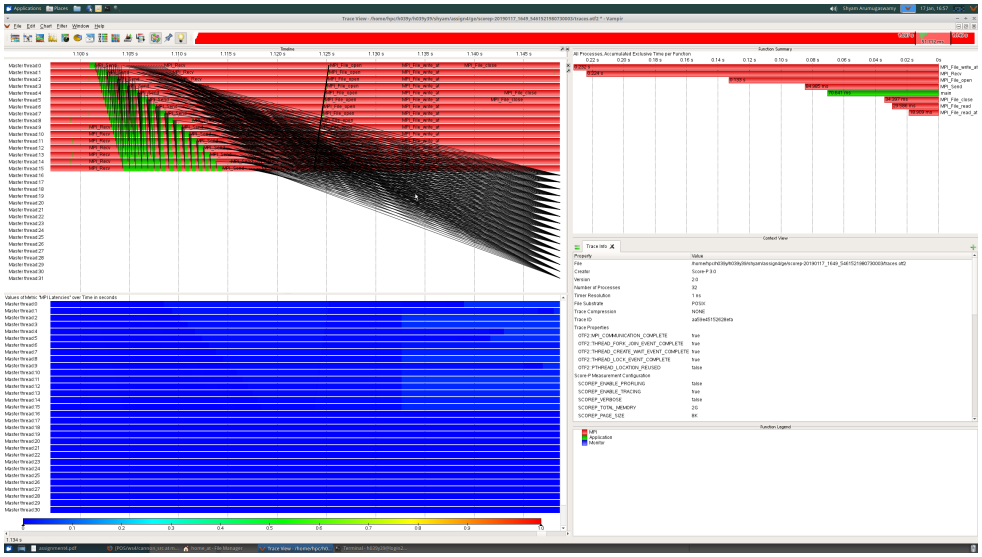
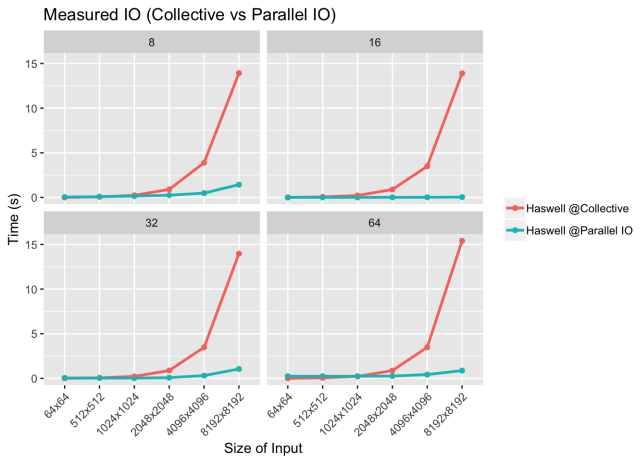
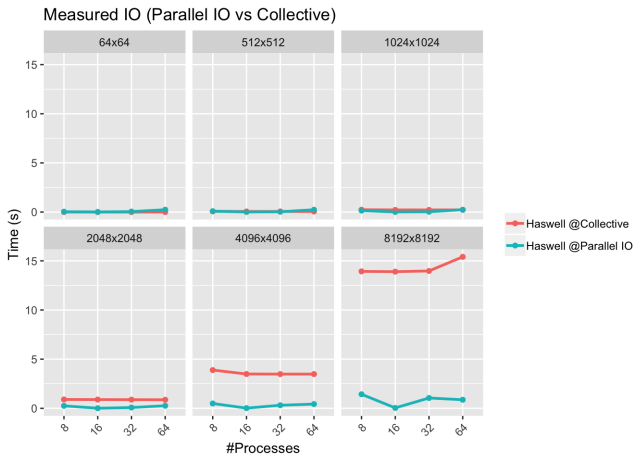


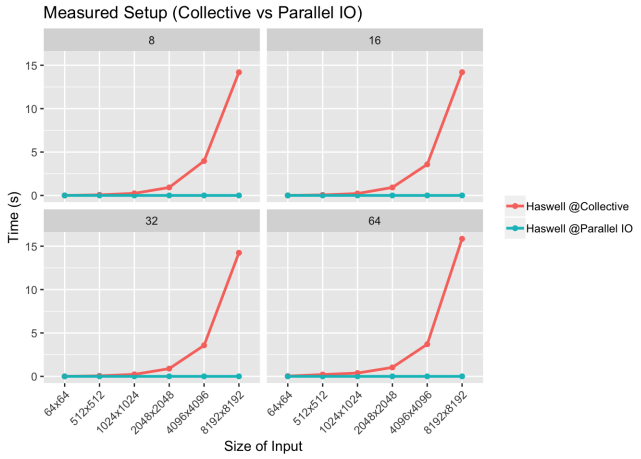
Figure : Vampir output for MPI Parallel IO Haswell - 32 processes

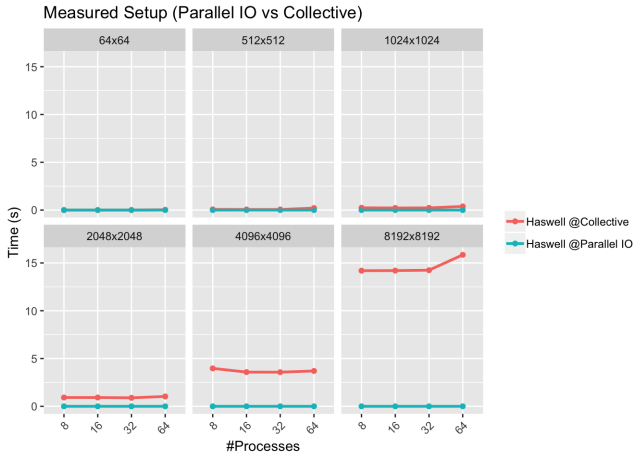
6. **Were there any performance improvements due to the change to MPI-IO?**

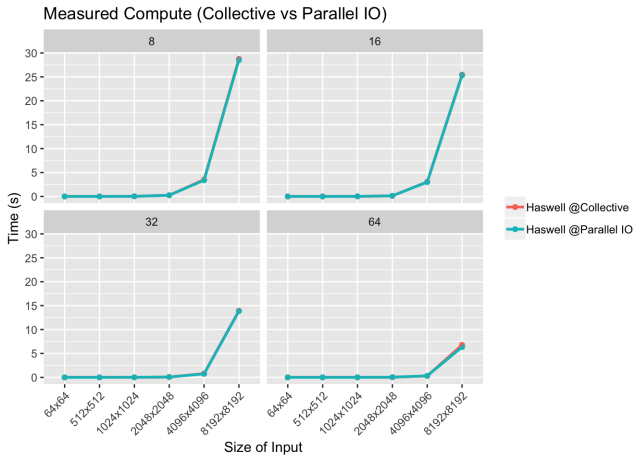
   Yes, some performance improvement was observed due to the changes to MPI-IO.

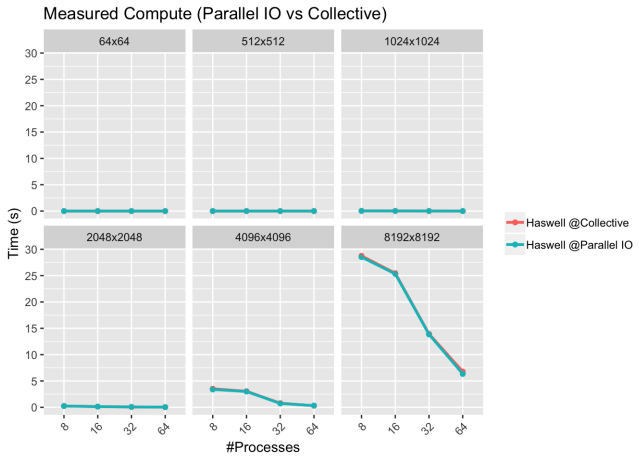

Measured IO (Collective vs Parallel IO)
8
16
32
64
Time (s)
Size of Input
64x64
512x512
1024x1024
2048x2048
4096x4096
8192x8192
Haswell @Collective
Haswell @Parallel IO

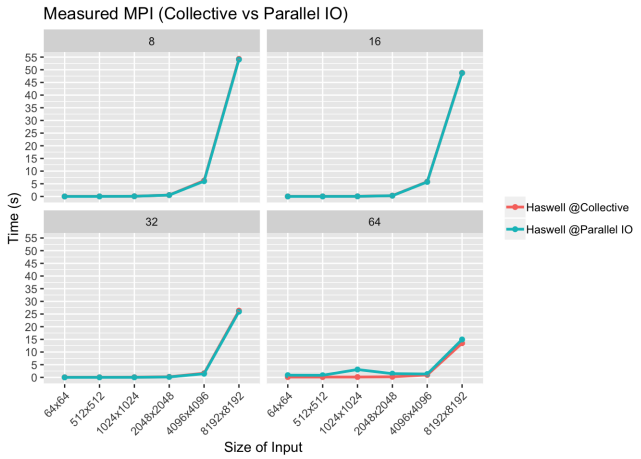

Measured IO (Parallel IO vs Collective)
64x64
512x512
1024x1024
2048x2048
4096x4096
8192x8192
Time (s)
#Processes
Haswell @Collective
Haswell @Parallel IO

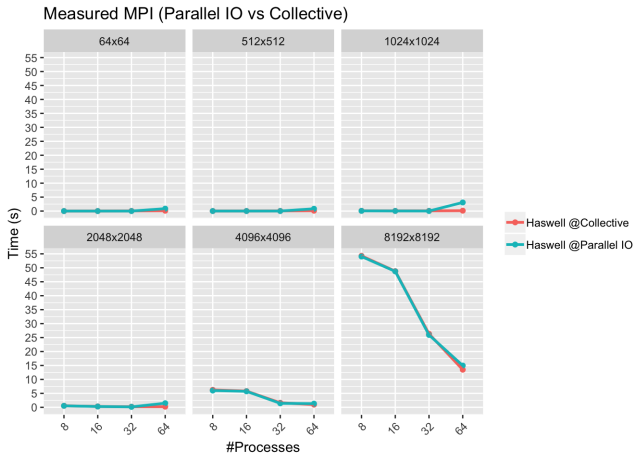

Measured Setup (Collective vs Parallel IO)
8
16
32
64
Time (s)
Size of Input
64x64
512x512
1024x1024
2048x2048
4096x4096
8192x8192
Haswell @Collective
Haswell @Parallel IO

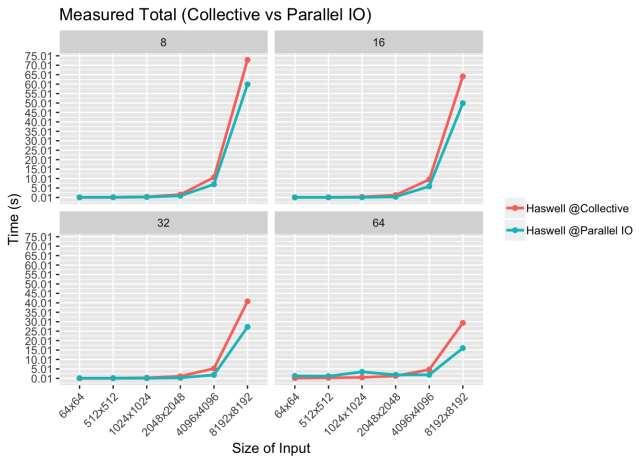

Measured Setup (Parallel IO vs Collective)
64x64
512x512
1024x1024
2048x2048
4096x4096
8192x8192
Time (s)
#Processes
Haswell @Collective
Haswell @Parallel IO

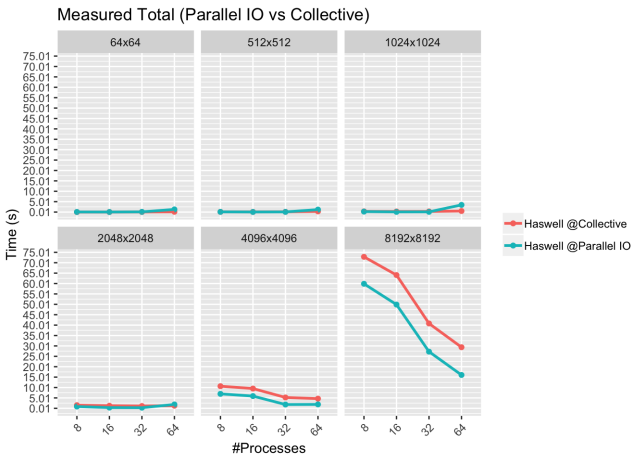

Measured Compute (Collective vs Parallel IO)
8
16
32
64
Time (s)
Size of Input
64x64
512x512
1024x1024
2048x2048
4096x4096
8192x8192
Haswell @Collective
Haswell @Parallel IO


Measured Compute (Parallel IO vs Collective)
64x64
512x512
1024x1024
2048x2048
4096x4096
8192x8192
Time (s)
#Processes
Haswell @Collective
Haswell @Parallel IO


Measured MPI (Collective vs Parallel IO)
8
16
32
64
Time (s)
Size of Input
64x64
512x512
1024x1024
2048x2048
4096x4096
8192x8192
Haswell @Collective
Haswell @Parallel IO


Measured MPI (Parallel IO vs Collective)
64x64
512x512
1024x1024
2048x2048
4096x4096
8192x8192
Time (s)
#Processes
Haswell @Collective
Haswell @Parallel IO





5 Contribution

1. **Smith**
   Build and run batch scripts for MPI Collectives and MPI Parallel IO

2. **Shyam**
   Understanding of the assignment and final report creation.
   Worked on MPI Collectives

3. **Siddhesh**
   Understanding of the assignment and report creation.
   Worked on MPI Parallel IO