Assignment 4:
MPI Collectives and MPI-IO

Emily Bourne (emily.bourne@tum.de)
Abraham Duplaa (abraham.duplaa@tum.de)
Jiho Yang (jiho.yang@tum.de)

February 1, 2018

1 Baseline

Unlike Assignment 3 where we used IBM MPI (mpi.ibm) for execution and Intel MPI only for trace analysis to observe how non-blocking communication behaves, Intel MPI (mpi.intel/2018) was used also for execution in this assignment. Different time measurements were made for this assignment. For this assignment, input time refers to the time taken to import the input matrices and setting up of arrays (until right before executing cannon's algorithm block). Output time refers to the time taken to collect the output matrix C to rank 0 until the end right before freeing memory (for MPI IO section this will be for writing local output matrix blocks to output file). Total time refers to time taken from importing matrices (where input time starts) until before freeing memory (where output time ends). Compute and mpi time from the main computation block remains the same as the last assignment. MPI time, however, is not considered for discussion since there are no collective operations or IO operations during the main cycle of the algorithm. Figure. 1 depicts the execution time for baseline implementation on Sandybridge and Haswell nodes. Note how the baseline implementation is IO bound, even for high dimensional problem where compute time is significantly increased. Figure. 1 is in log-log scale for better visualization, but for quantitative observation Tab. 1 provides the average execution time for baseline with $N = 4096$. Note how the input time takes most of the execution time, even up to 72% of the total time. Figure. 5 depicts the trace analysis of baseline implementation on Sandybridge node for $N = 4096$. Note the long waiting time for other processes except P0, caused by P0 sending out messages at the initial stage which leads to non-synchronized start of the main loop. It can also be visually observed that the initial matrix import process via message passing takes a significant amount of time in the total execution.

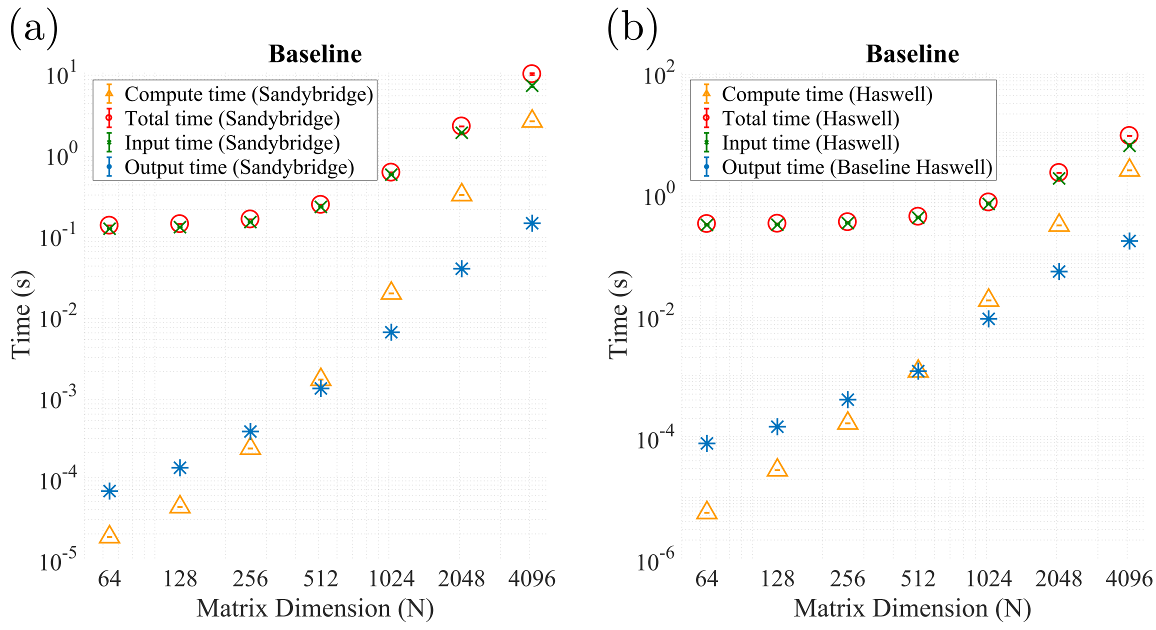


Figure 1: Execution time for baseline implementation in log-log scale on (a) Sandybridge and (b) Haswell nodes. Intel MPI (mpi.intel/2018) was used. Note how the baseline implementation is IO bound, even for high dimensional problem where compute time is significantly increased.

Node	Computation (s)	Input (s)	Output (s)	Total (s)	Input/Total (%)
Sandybridge	2.43	6.67	0.14	9.28	71.9
Haswell	2.38	6	0.16	8.74	68.6

Table 1: Average execution time for baseline with N = 4096. Input time takes significant amount of portion from the total time, making it an IO bound problem and less scalable.

2 MPI Collectives

In Cannon's algorithm 3 collective communication patterns were identified and replaced with collective operations.

2.1 Broadcast

The following code was identified as being equivalent to a broadcast operation:

```
// send dimensions to all peers
if(rank == 0) {
        int i;
        for(i = 1; i < size; i++){
                MPI_Send(matrices_a_b_dimensions, 4, MPI_INT, i, 0,
                        cartesian_grid_communicator);
        }
} else {
        MPI_Recv(matrices_a_b_dimensions, 4, MPI_INT, 0, 0,
                cartesian_grid_communicator, &status);
}
```

It was therefore replaced with the following line:

```
MPI_Bcast(matrices_a_b_dimensions,4,MPI_INT,0,cartesian_grid_communicator);
```

The dimensions are then used for all subsequent calculations so the algorithm cannot progress until this step is completed. As a result a blocking operation is used.

2.2 Scatter

The following code was identified as being equivalent to a scatter operation:

```
// send a block to each process
if(rank == 0) {
                int i;
                for(i = 1; i < size; i++){
                        MPI_Send((A_array + (i * A_local_block_size)),
                        A_local_block_size, MPI_DOUBLE, i, 0,
                    cartesian_grid_communicator);
                        MPI_Send((B_array + (i * B_local_block_size)),
                        B_local_block_size, MPI_DOUBLE, i, 0,
                    cartesian_grid_communicator);
                }
                for(i = 0; i < A_local_block_size; i++){
                        A_local_block[i] = A_array[i];
                }
                for(i = 0; i < B_local_block_size; i++){
                        B_local_block[i] = B_array[i];
                }
        } else {
                MPI_Recv(A_local_block, A_local_block_size, MPI_DOUBLE, 0, 0,
                        cartesian_grid_communicator, &status);
                MPI_Recv(B_local_block, B_local_block_size, MPI_DOUBLE, 0, 0,
                        cartesian_grid_communicator, &status);
        }
```

It was therefore replaced with the following lines:

```
MPI_Scatter(A_array,A_local_block_size,MPI_DOUBLE,A_local_block,
          A_local_block_size,MPI_DOUBLE,0,cartesian_grid_communicator);
MPI_Scatter(B_array,B_local_block_size,MPI_DOUBLE,B_local_block,
          B_local_block_size,MPI_DOUBLE,0,cartesian_grid_communicator);
```

The local arrays are required in order for the algorithm to continue. As a result a blocking operation is used.

2.3 Gather

The following code was identified as being equivalent to a gather operation:

```
// get C parts from other processes at rank 0
if(rank == 0) {
    for(i = 0; i < A_local_block_rows * B_local_block_columns; i++){
        C_array[i] = C_local_block[i];
    }
    int i;
    for(i = 1; i < size; i++){
        MPI_Recv(C_array + (i * A_local_block_rows * B_local_block_columns),
            A_local_block_rows * B_local_block_columns, MPI_DOUBLE, i, 0,
            cartesian_grid_communicator, &status);
    }
} else {
    MPI_Send(C_local_block, A_local_block_rows * B_local_block_columns,
          MPI_DOUBLE,0, 0,cartesian_grid_communicator);
}
```

It was therefore replaced with the following lines:

```
int C_local_block_size = A_local_block_rows * B_local_block_columns;
MPI_Request request;
MPI_Igather(C_local_block,C_local_block_size,MPI_DOUBLE,
          C_array,C_local_block_size,MPI_DOUBLE,0,
          cartesian_grid_communicator,&request);
```

The results of the gather operation are not used immediately. As a result a non-blocking operation can be used. At the point at which it is needed *MPI_Wait()* must be called on all processors to synchronise them and ensure that the data is available.

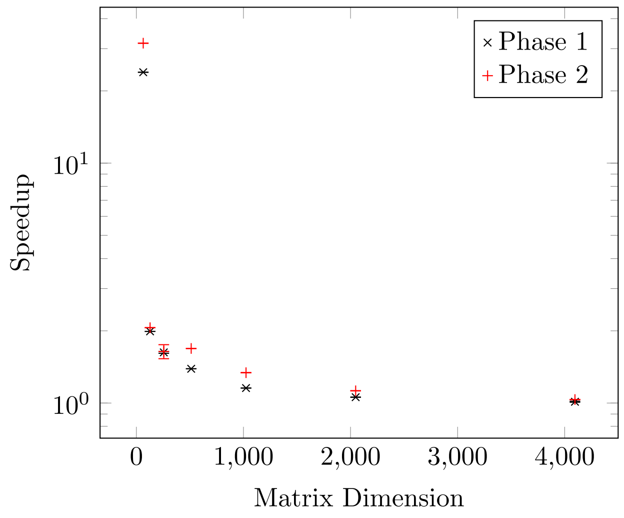
Rank 0 is a special case, but for all other ranks *MPI_Wait()* can be called before freeing the local block matrices. For rank 0 the position of the synchronisation depends on how many arguments are passed to the program. If less than 4 arguments are passed then the solution is not tested against the serial code. In this case *MPI_Wait()* can be called before freeing all memory.

The remaining case (rank 0 with comparison against serial code) is the most interesting. In this case multiple operations can be carried out before synchronising, namely the printing of the sizes and compute times (as for the case without comparison against serial code), as well as the printing of the matrices A and B.

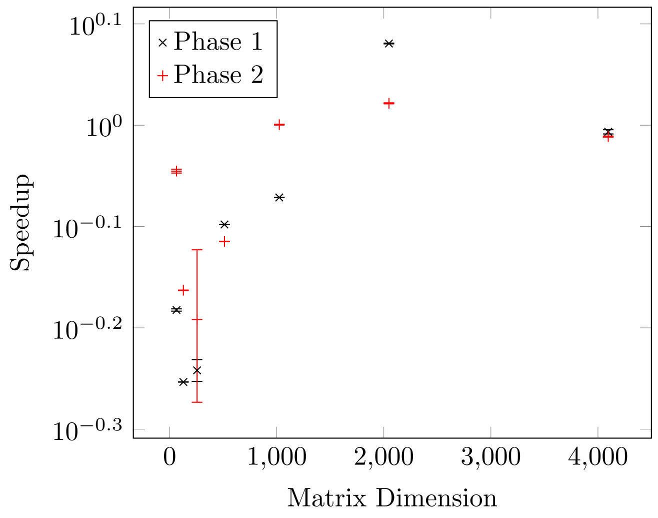
2.4 Results

The results of changing the code to use collective operations can be seen in figure 12. It can be seen that collective operations have a large effect on the time taken for the input for small matrices but this improvement rapidly reduces for larger matrices (speedup of 2 for 128x128 matrices with speedup tending towards 1). This is probably because the problem is IO bound and therefore any effect due to the collective operations is hidden by the increasing time taken for the matrices to be imported.

During the output the improvement is much smaller and sometimes the results are in fact worse. This may be due to the fact that a non-blocking operation has extra overhead. The results in Assignment 3 show that using non-blocking operations can have unexpected results. The main advantage of non-blocking operations is that they allow overlap between computation and communication. This is discussed earlier in the report. However when the tests were run they were not run in ”test” mode. This means that almost all possible overlap was removed. Thus the only effect remaining is additional overhead. This may explain why the results seem to be so poor. This analysis is borne out by the trace analyzer (figure 12) as we see that there is very little difference between the output with and without collective operations and that there is very little possible overlap.



(a) Comparison of the speedup of the input on phase 1 and phase 2 showing the improvement gained from collective operations until IO dominates the input time



(b) Comparison of the speedup of the output on phase 1 and phase 2 showing little or no improvement perhaps due to non-blocking operations.

Figure 2: Comparison of the speedup on different phases during different sections of the code when collective operations are used.

1. **Would you expect performance or scalability benefits from the changes in this application? Explain :** We would expect performance benefits as collective operations are highly optimised. This means that they are at least as good, but usually better, than the non-collective alternatives. We would not however expect scalability. Collective MPI doesn't affect scalability because communication time takes only a small portion of the whole input time. For small matrices performance can be gained by using collective MPI, because IO time (on rank 0) is fast, but for larger matrices this takes longer and dominates the time recorded. In addition even if the computation and input/output were improved enough to make the problem communication bound, the collective operations only occur at the start and end of the program and do not make up the majority of the communications.

2. **Is the resulting code easier to understand and maintain after the changes? Why?** The resulting code is easier to understand as the collective operations are common and therefore easily recognised. In contrast the original code which takes multiple lines must be read in its entirety and the logic followed in order to come to the same conclusion. The code is much easier to maintain as the MPI code is optimised. If improvements are made to this algorithm then the computer will automatically use the newer version without the user having to recompile. In addition if the matrix multiplication algorithm should be changed then it is much easier to locate the appropriate section.

3 MPI Parallel IO

3.1 Brief overview of Parallel IO

The standard methodology for performing IO is based on POSIX IO. POSIX IO arises from Portable Operating System Interface for Computing Environments (POSIX), which defines a standard interface for applications to obtain standard services from the operating system [1]. The POSIX IO which stems from POSIX was designed primarily with the intention to work with a single machine with a single memory space. The semantics of these interfaces make it hard to achieve good performance when a large number of machines access shared and/or distributed storage, just like modern supercomputers. Ken Batcher, an Emeritus professor of Computer Science, stated that *A supercomputer is a device for turning compute-bound problems into IO-bound problems.* As described in his statement, IO is becoming a significant bottleneck for modern supercomputing. By using the standard POSIX IO, which is inherently sequential, parallel applications suffers from lack of scalability from IO, both time and memory-wise. Not only is parallel IO important for general performance improvements, but also for fault tolerance (e.g. checkpointing and restart policies), where the application will be stopped periodically, then will copy all the required data from memory to reliable storage (e.g. Parallel file systems). This process requires copying very large amounts of data, hence it must also be minimized. This is why parallel IO is so important when developing truly scalable and memory efficient parallel applications. There are three main methods to perform IO operations.

1. Sequential IO - Use of dedicated process
   - A dedicated process (e.g. P0) performs IO and distributes/collects data via message passing
   - Advantage: Simple to implement (sequential IO)
   - Disadvantage: Memory management becomes an issue, and additional message passing slows down the performance.
   - See Figure 9 for a visual representation of Sequential IO.
2. Independent Parallel IO - Each process has own files
   - Advantage: High performance (local memory access)
   - Disadvantage: Distribution and integration of data becomes an issue, and large number of files are required.
   - See Figure 10 for a visual representation of independent parallel IO.
3. Cooperative Parallel IO - Processes read/write same file
   - Advantage: Easy management of data redistribution
   - Disadvantage: Difficult to get same performance as each process having own files.
   - See Figure 11 for a visual representation of cooperative parallel IO.

3.2 Parallel IO using MPI

MPI has specific IO operations in order to achieve parallel **and** cooperative IO. This involves two major steps: Opening/Closing and Reading/Writing. Common MPI IO operations are described below:

3.2.1 Opening and Closing Files

- **MPI_File_open**: Opens the specified file on all processes in the communicator group. This is a collective operation, hence all processes must provide the same value for access mode(amode) and file names (filename) that reference the same file. Optimization hints can be specified using MPI_Info for further details of the operation. The returned file handle (fh) can be used for later access to the file until the file is closed using MPI_File_close.
- **MPI_File_close**: Closes the file identified by the file handle and deallocates associated internal data structures. It may also delete the file if the appropriate mode was set when the file was opened (MPI_MODE_DELETE_ON_CLOSE). This is a collective operation.

3.2.2 Reading and Writing from Files

- **MPI_File_read**: Reads the file identified by the file handle. This command uses an individual file pointer and is a blocking, non-collective operation. MPI also offers non-blocking and collective sub-operations of MPI_File_read (e.g. MPI_File_iread, MPI_File_iread_at_all_begin)
- **MPI_File_write**: Writes in the file identified by the file handle. This command uses an individual file pointer and is a blocking, non-collective operation. MPI also offers non-blocking and collective sub-operations of MPI_File_write (e.g. MPI_File_iwrite, MPI_File_write_at_all_begin)

3.2.3 Setting File Views

- **MPI_File_set_view**: This operation is used to set each process's view of the data in the file. The view begins after the displacement (disp). The displacement is a non-negative integer that specifies the absolute offset in bytes from the beginning of the file to where the view of the process's begins. The disp can be used to skip headers or to offset the view to be accessed in different patterns when the file includes a sequence of data segments. Each process has a specified view which the process then reads or writes from. MPI_File_set_view has other inputs, notably elementary datatype (etype) and data representation (datarep). The etype specifies the data layout of the file and the datarep specifies how the data is represented (eg. native, internal, external32). In addition, this operation resets the individual file pointer and shared file pointer to zero. This operation was used in our code implementation.

3.2.4 Collective IO Operations

- **MPI_File_read_all**: This operation is a collective version of the MPI_File_read operation. This operation reads from the file associated with the file handle and reads a total number of count data items. This operation is collective and blocking. This operation has been explicitly described in the report because it was used in our implementation.
- **MPI_File_write_all**: This operation is a collective version of the MPI_File_write operation. This operations writes to the file associated with the file handle and writes a total number of count data items. This operation is collective and blocking. This operation has been explicitly described in the report because it was used in our implementation.

3.2.5 Positioning

The access position of each process within the file is dictated by explicit offsets, individual file pointers, or shared file pointers depending on which MPI operation is executed.

- **Explicit Offsets**: Data access with explicit offsets provides the explicit location, measured in elementary data type units relative to the current view. For each call to a data-access routine, a process attempts to access a specified number of file types of a specified data type (starting at the specified offset) into a specified user buffer.
- **Individual File Pointers** : Each MPI process has one individual file pointer per file handle and the current value of the individual file pointer implicitly specifies the offset when the suitable MPI IO operations are used. When using operations that take the individual file pointer as an argument, only the individual file pointer is updated while the shared file pointer is not used nor updated. The individual file pointer is updated relative to the current view of the file.
- **Shared File Pointers**: MPI will also maintain one shared file pointer per collective MPI_FILE_OPEN operation. The value of the shared file pointer implicitly specifies the offset in the MPI operations that receive shared file pointers. Similarly to individual file pointers, when MPI operations which receive shared file pointers are used, only the shared file pointer is updated and maintained by MPI. The individual file pointers are not used and are not updated. A shared file pointer MPI IO operation should only be used if all processes use the same file view, otherwise the operation is erroneous.

3.2.6 Collective and Split-Collective Explanation

Traditional collective operations (e.g. MPI_Gather) are blocking operations and are used for operations which involve many processes. However for IO operations, MPI has "non-blocking collective" operations for all data accesses which are known as **split-collective** operations. The reason that they are "split"-collective is due to the fact that the collective function is actually broken up in to two parts, a begin routine and an end routine. By being broken up in to two parts, the split-collective operations are able to begin the IO routine in a non-blocking fashion, and end the IO routine similar to how an MPI_WAIT would synchronize a non-blocking operation. The begin and end routines are needed for split-collective operations. An example of a begin and an end routine are MPI_File_read_at_all_begin and MPI_File_read_at_all_end, respectively

3.3 Cannon's algorithm with MPI IO

3.3.1 Summary of MPI IO implementation

Our implementation of Cannon's algorithm with MPI IO consists of five steps:

1. Pre-processing step that converts the provided matrix files into binary files. Although listed here, this must be considered as an additional/external processing step
2. Parallel reading of sub-domains of matrices at each processes from a single matrix file(s).
3. Parallel computation (main computation block). For sake of isolating the performance improvements from MPI IO, baseline computation approach with blocking communication is used.
4. Parallel writing of local sub-domains of resulting matrix from each processes to a single output file.
5. Post-processing step that converts the resulting output matrix file from binary format to ASCII format. Again, this must be considered as external processing step.

3.3.2 Reading The Input Files

MPI IO natively works with binary files, and hence the input files must be converted into binary format, and the output file into ASCII format to make it human-readable. If the original human-readable input file is used wrong values are imported since in text format, the size of each number in the file varies. Since this conversion only needs to be done once before executing the program, we considered it as a pre-processing step. The same approach as baseline implementation was used for pre-processing step: read the input matrices from rank 0, distribute to other processes, but then write these sub-domain values from each process to a common output file using MPI_File_write_all for each matrix A and B (matrix dimension headers are written only with rank 0). In a real life application, however, a more sophisticated conversion technique would be needed. Pre-processing code (*generateBinaryInput.c*) is included in the submission files.

Once the binary files are obtained, *cannon.c* is executed. For simplicity only importing of matrix A is described (matrix B follows the same approach). Matrix A file is opened using MPI_File_open. Matrix dimensions are read first using MPI_File_read_all. 2D sub-array MPI data type is then created using MPI_Type_create_subarray and committed using MPI_Type_commit. This sub-array data structure provides an elegant and intuitive mean to access sub-domains of the matrix by specifying a few parameters, in particular its dimension, size, and starting indices. Using a displacement with size of two integers along with sub-array, a file view is set by using MPI_File_set_view. File view indicates where each process should be *viewing/referring* to the input file. The displacement skips the header (matrix dimensions) and sub-array guides the process to point to the right location of sub-domain to be read. Then the sub-domains of the matrix are read using MPI_File_read_all.

3.3.3 Writing The Output File

Shortly after cannon's algorithm is executed, the original implementation of cannon.c gathers the local output results of C to rank 0 and then prints the complete C matrix to an output file. Although this IO implementation is intuitive, sequential IO operations through a single process can lead to a significant bottleneck (as shown in figure 9). The output section has been modified in order to prevent this bottleneck and make use of useful MPI cooperative (Figure 11) IO operations. This section first begins with creating a file handle for the output file (fh_c) and then the file name is created by using the following convention: "A_rows x B_columns".
The main part begins when the C output file is opened by using MPI_File_open (refer to 3.2). MPI_MODE_CREATE is passed as an argument to MPI_File_open so that the output file is created if it does not exist. To prohibit the program from reading the file at any point, the argument MPI_MODE_WRONLY is also passed to ensure the file is only accessible for writing.
Once the file has been opened and created, the file is ready to be written. For comprehensiveness, a header is written to the first row of the output file. The header is only written by process rank 0.
Writing the C matrix involves a couple steps. Since a header has already been written to the output file, an MPI_Offset variable, disp_header, is created to ensure the accessible view of the output file for the processes does not include the header. The displacement is defined as 2*size(int). If a displacement was not used, the header would be overwritten by the matrix. In order to optimize MPI IO in writing the matrix, a subarray (c_subarray) is created using MPI_Type_create_subarray similarly to how a subarray is created for the previously explained read operation. Once c_subarray for writing has been defined, the subarray can then be used to set the file view for every process. MPI_File_set_view receives c_subarray as an argument and effectively indicates where each process should be *writing* to. Refer to Section 3.2.3 for information on MPI_File_set_view.
Now that the output file is created and opened, the subarray has been created, and the view has been set for each process, the matrix can finally be written. In order to take advantage of MPI IO collective operations, MPI_File_write_all is used to collectively write the output matrix. Once the file has been completely written by all the processes, the file is closed using MPI_File_close.
The output file is written in binary format. In order to make the file human-readable, the binary files are converted to ASCII format by using a shell script. The shell script is included in the submission files.

3.3.4 Justification of MPI IO Operation Choices

In order to gain the most performance increase by using IO operations, each IO operation used in the code was given considerable thought. Initially, non-blocking and split-collective operations were considered for implementation. However, our implementation of Cannon's algorithm requires every process to contain the correct data before any computation is done. Using a non-blocking and split-collective operation for this case would not provide any performance increase since there is no local computation before cannon's algorithm section. Non-blocking, split-collective operations could decrease performance when compared to our IO implementation due to MPI overhead.

By using MPI_File_read_all and MPI_File_write_all combined with MPI_File_set_view, we gain a significant performance increase since all processes can access data concurrently, we limit the amount of additional potential

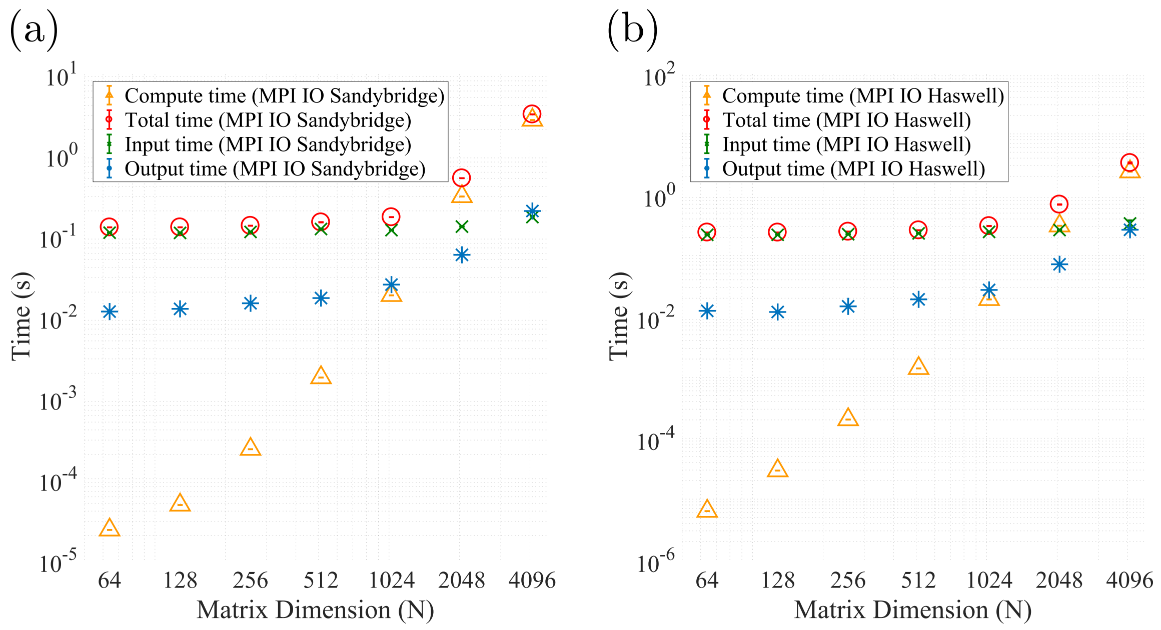


Figure 3: Execution time for MPI IO implementation in log-log scale on (a) Sandybridge and (b) Haswell nodes. Compared to baseline the problem is now compute-bound for big matrices. Particularly for $N \geq 2048$ compute time becomes dominant. The input time is near constant for different matrix dimensions, hence it is decoupled from the problem scale and more scalable.

overhead caused by using non-blocking and split-collective operations. Lastly, the MPI IO operations chosen provide an elegant and intuitive code that is easy to understand. Using MPI_File_set_view with a subarray is intuitive and makes MPI IO operations much easier to follow.

3.4 Results and Discussion on MPI IO

3.4.1 Reading and Writing Analysis

Figure. 3 depicts the execution time for MPI IO implementation in log-log scale. Comparing it to the baseline performance (Fig. 1) it must be noted that now the problem is compute bound for big matrices. Particularly for $N \geq 2048$ compute time becomes dominant. By using MPI IO, the input time becomes near constant for different matrix dimensions, hence becoming decoupled from problem scale and more scalable. Output time, on the other hand, slowly increases up to $N = 1024$ then increases more significantly afterwards. For a more quantitative observation, Tab. 2 provides the execution time for MPI IO implementation for $N = 4096$. Now the input/total time is reduced down to 0.05%, making computation time to be the most dominant. Also note the reduction in total compared to the baseline. Speedup of 3 was achieved for total time, greatly improving the program performance. This is a very important aspect of parallel IO: efficient IO helps to make a parallel application truly scalable.

Node	Computation (s)	Input (s)	Output (s)	Total (s)	Input/Total (%)
Sandybridge	2.63	0.17	0.2	3.1	0.05
Haswell	2.39	0.34	0.27	3.33	0.1

Table 2: Average execution time for MPI IO implementation with N = 4096. Note the reduction in total and input time compared to the baseline implementation. By using MPI IO the problem becomes compute-bound.

Figure. 4 depicts the speedup gained from MPI IO implementation in log-log scale. Note significant speedup gained for input time. Unlike sequential IO, which suffers from increasing input time for increasing matrix dimensions, MPI IO achieves near constant input time, hence obtaining a significant speedup for big matrices. For small matrices, on the other hand, the speedup is not so significant. Total time also gained very good speedup. Similar to input time, its speedup is marginal for small matrices, but its performance improves for bigger matrices until it reaches critical point at $N = 2048$. However, this critical point happens due to large amount of computation needed, and can be solved by having more compute resources. Unlike MPI input time, where we get significant speedup, MPI output time faced speed-down. This speed-down does get better as matrix dimension increases, but never reached speed-up. Furthermore, for Haswell node, we observe slight performance deterioration for $N = 4096$. We believe this could potentially be caused by the overhead from starting MPI IO, and hence if the problem size is not big enough, no speedup is guaranteed (i.e. MPI IO optimization is only possible at some range of problem size). This argument, however, is not completely sound. This would mean that for big matrices (say $N = 2048$) we should be gaining speedup since the matrix size is bigger than two 1024×1024 matrices, but we get speedup from MPI input for $N = 1024$, yet speed-down from MPI output for $N = 2048$. Comparing read and write times is a very complicated process that hugely depends on the hardware and file systems. Generally speaking for distributed systems, output time tends to be slower than read times. In order to correctly study this matter, we would need to run the application for matrices $N > 4096$. And despite critical-point like behaviour for $N \geq 2048$, we would expect the output time to eventually gain speedup compared to the baseline implementation for bigger matrices.

3.4.2 Speedup Comparison Between Haswell and Sandybridge

Table. 3 gives quantitative values of speedup for $N = 4096$. For Sandybridge, significant speedups of 40 and 3 were achieved for input and total times, respectively. Although less, Haswell node also gained notably good speedup. The difference between Sandybridge's and Haswell's IO speedups is due to architecture and specification differences. A potential reason for their speedup discrepancy could be due to their differences in bandwidth specifications. Although Haswell has higher bandwidth to node memory of $137GB/s$ compared to Sandybridge's $102.4GB/s$, **Sandybridge has higher bandwidth per core as well as lower latency per core**. The bandwidth per core and latency per core should have a larger impact, as shown in the example below.

To transfer a file from memory to core would require the files first being sent to the node and then being sent to the core. For example, the largest input file is approximately 100MB. To send 100MB on Haswell and Sandybridge are described below:

- **Haswell**: Time to node: $\frac{100MB}{137GB/s} = 0.0007s$, Time to core: $\frac{100MB}{6.7GB/s} = 0.0149s$, **Total Time = 0.0156s**
- **Sandybridge**: Time to node: $\frac{100MB}{102.4GB/s} = 0.0009s$, Time to core: $\frac{100MB}{8.8GB/s} = 0.01136s$, **Total Time = 0.0123s**

**Note: Latency is not taken in to consideration above.*
The overall speedup (shown in Table 3) is similar for both nodes even with the notably large difference in IO speedups because the application becomes a compute-bound program when MPI IO operations are used. Any additional speedup due to IO operations after the program has become compute-bound will have less of an impact on the overall total speedup.

3.4.3 Trace Analysis Discussion

For sake of reducing redundancy and saving space in this report, the results of the trace analysis from Sandybridge nodes are provided in this report. The Haswell architecture produced similar trace files and therefore are not required. All trace analysis plots were created for $N = 4096$, the largest matrix size.

Reading and Distribution of Input File Data As shown in the trace analysis of the baseline in Figure 5, the baseline cannon implementation spends a significant amount of time performing MPI operations at the start of running the code (shown in red). This MPI section is caused by rank 0 reading the input files and then communication between rank 0 and the other processes to distribute blocks of input data. From the trace analysis, it's clear that the initial MPI section accounts for a considerable amount of the total run time and that a speedup in IO would significantly affect run time. Figure 6 visually displays the impact of using MPI IO operations. The initial MPI section when using MPI IO is significantly smaller than the baseline. Furthermore, rank 0 is also equally involved in the IO process. Each process has access to the data and the processes can access the file concurrently, which leads to the notable difference in the Trace Analyzer. The initial MPI section ends in unison between the processes because MPI_File_read_all is a blocking operation. This is shown in Figure 7. For a detailed explanation of the significant performance gains in reading by using MPI IO operations, please refer to Section 3.4.1.

Gathering and Writing of Output File Data Using MPI IO operations for writing did not have the considerable performance impact that IO operations had while reading. In fact, it is difficult to see a difference between writing with MPI IO and the baseline in Figure 6. This agrees with Figure 4 (speedup is roughly 1 for the $N = 4096$ case). However, communication is greatly reduced after cannon algorithm computation. As shown in Figure 8, the communication between processes has been decreased significantly. For a detailed explanation to why output time using MPI IO operations did not provide a speedup, please refer to Section 3.4.1.

Overall, just like the previous assignment, Sandybridge node performed better than Haswell node.

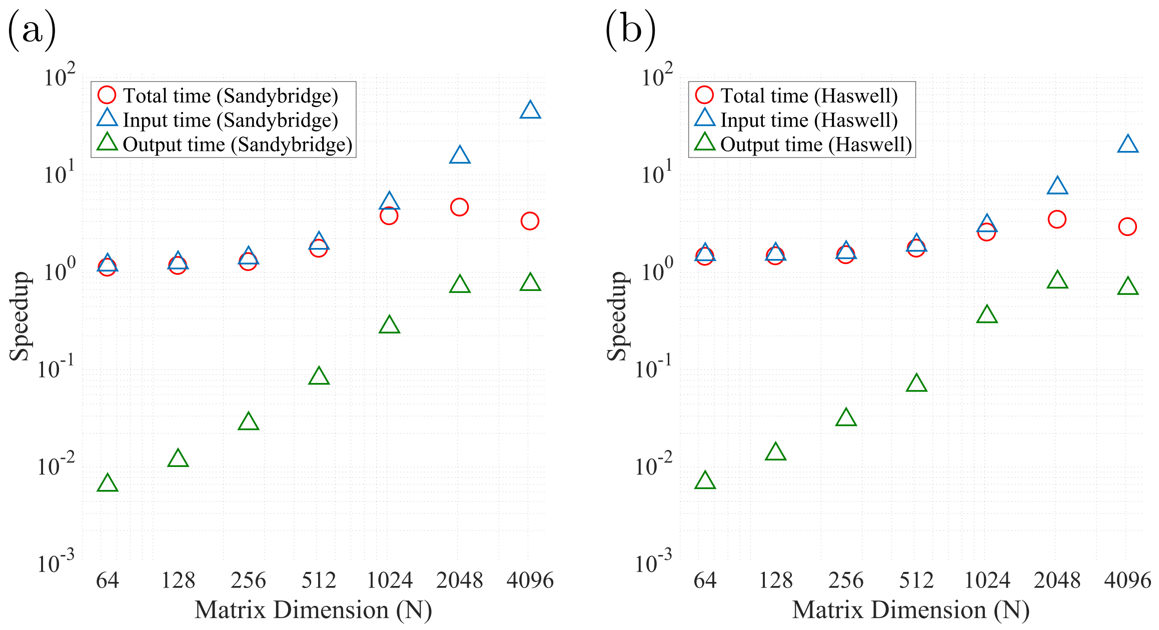


Figure 4: Speedup gained from MPI IO implementation in log-log scale on (a) Sandybridge and (b) Haswell nodes. Near constant input time was achieved, hence obtaining a significant speedup for big matrices.

Node	Input	Output	Total
Sandybridge	39.73	0.68	3
Haswell	17.78	0.62	2.62

Table 3: Speedup gained from MPI IO implementation for N = 4096.

Other then performance improvements gained, MPI IO also makes the code far more elegant, intuitive, and readable. The names of the MPI functions for parallel IO are very intuitive and compact. In particular, creation of 2D sub-arrays greatly simplifies access to location of intended sub-domain within the full matrix. Therefore, MPI IO not only significantly improves parallel application's performance and scalability, but also improves the quality of the code in software engineering perspective.

N	Input time (s)	Communication time (s)	Communication / Input (%)
64	0.11	0.11	97.82
128	0.12	0.11	93.80
256	0.14	0.11	80.66
512	0.22	0.11	53.08
1024	0.52	0.12	23.80
2048	1.74	0.17	9.79
4096	6.52	0.24	2.58

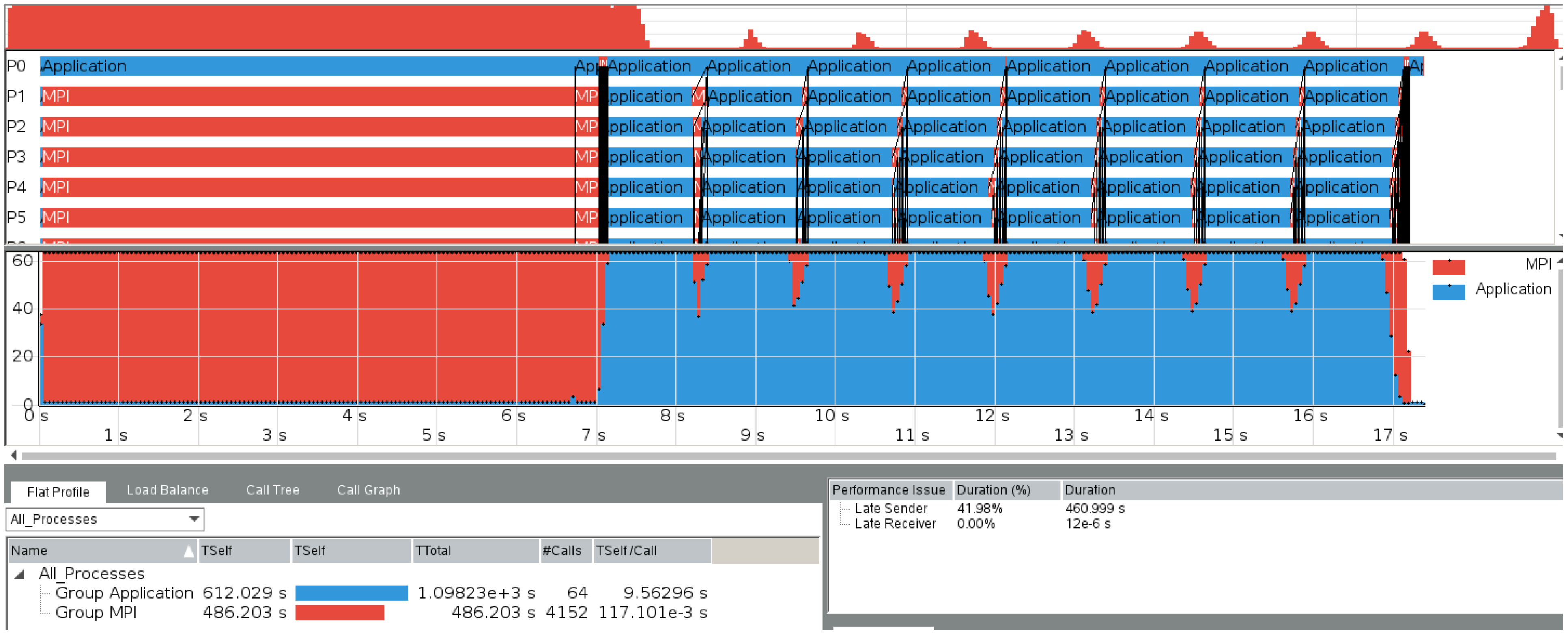
Table 4: Input and communication time from baseline implementation on Sandybridge node rounded to two decimal places. N is matrix dimension, input time is total time taken to import the matrices, and communication time is the MPI communication time taken for distributing the sub-domain of the imported matrices to other processes.

The following block provides a summary/short answers to the questions. For details, please refer to previous subsections on MPI Parallel IO.

Q1) What are "Data Sieving" and "2-Phase IO"? How do they help improve IO performance?

Data Sieving is a simple solution to the challenge of IO when data reading/writing needs to be performed on a large number of small, non-contiguous (not in a continuous sequence) pieces of data from a file. Instead of performing IO for each non-contiguous portion of data, data sieving accesses a single contiguous chunk of data starting from the first requested byte up to the last requested byte. This data is read in to a temporary buffer. Once all the data is in the temporary buffer, the data to be used by the program is then

extracted and placed in the user's buffer. Unfortunately, the non-contiguous data issue is usually the case for many scientific problems. If the program were to make many small IO requests to access this data, the IO performance would be very poor due to IO overhead. Generally, reading more data than is needed in to a temporary buffer usually outweighs the cost of performing several IO operations. Data sieving is memory bound by the temporary buffer, and so data sieving can be performed in several parts in order to fit the data in to the temporary buffer. As stated above, data sieving improves IO performance by decreasing the amount of IO operations needed to be performed and the cost of accessing more data than necessary usually performs faster than multiple IO operations. However, when the distance between one sequence of data and another sequence of desired data is very large within the file, data sieving may not be the best option for optimal performance.
2-Phase IO can be used to improve IO performance when the access patterns of all processes is known. The data access is split in to two phases (hence the name). For the first phase, each process accesses the data assuming that a large and contiguous distribution of data needs to be accessed leading to a single, long, contiguous access for each process. In phase 2, the processes redistribute the data amongst themselves to finally obtain the correct data. 2-Phase IO improves IO performace because it reduces the IO time by reducing the amount of IO operations (recall from data sieving that one large, contiguous IO operation takes less time than multiple IO operations). The cost of interprocess communication for phase 2 is generally small compared to the cost of performing several common IO operations.
Q2) Was the original implementation scalable in terms of IO performance?
No. The matrices are imported/read and exported/written only at the root process, and no parallel file system is used for baseline implementation. Therefore, it performs a strictly sequential IO.
Q3) Was the original implementation scalable in terms of RAM storage?
No. the root process imports the full matrices then distributes the decomposed blocks to other processes. This raises a problem for working with very big matrices which require more memory space than RAM equipped on the processor/node (depending on the architecture) where the root process is generated.
Q4) How much of the communication in the application was replaced with MPI-IO operations? All MPI communication operations utilized to distribute and gather data for IO purposes in the baseline code were replaced with MPI-IO operations. For a quantitative description of the time saved by replacing communication with MPI-IO operations, please refer to Table 4. When using MPI-IO, the file view for each process is set so that the process has access to the data it will need, hence not requiring communication after IO is executed. Communication between processes while computation of cannon's algorithm remains the same, as this communication was not part of the scope of this assignment.



Name	TSelf	TSelf	TTotal	#Calls	TSelf/Call
All_Processes					
Group Application	612.029 s	🟦	1.09823e+3 s	64	9.56296 s
Group MPI	486.203 s	🟥	486.203 s	4152	117.101e-3 s

Performance Issue	Duration (%)	Duration
Late Sender	41.98%	460.999 s
Late Receiver	0.00%	12e-6 s

Figure 5: Trace analysis of original Cannon's algorithm implementation on Sandybridge node for $N = 4096$. Note the long waiting time for other processes except P0, caused by P0 sending out messages at initial stage which leads to non-synchronized start of the main loop.

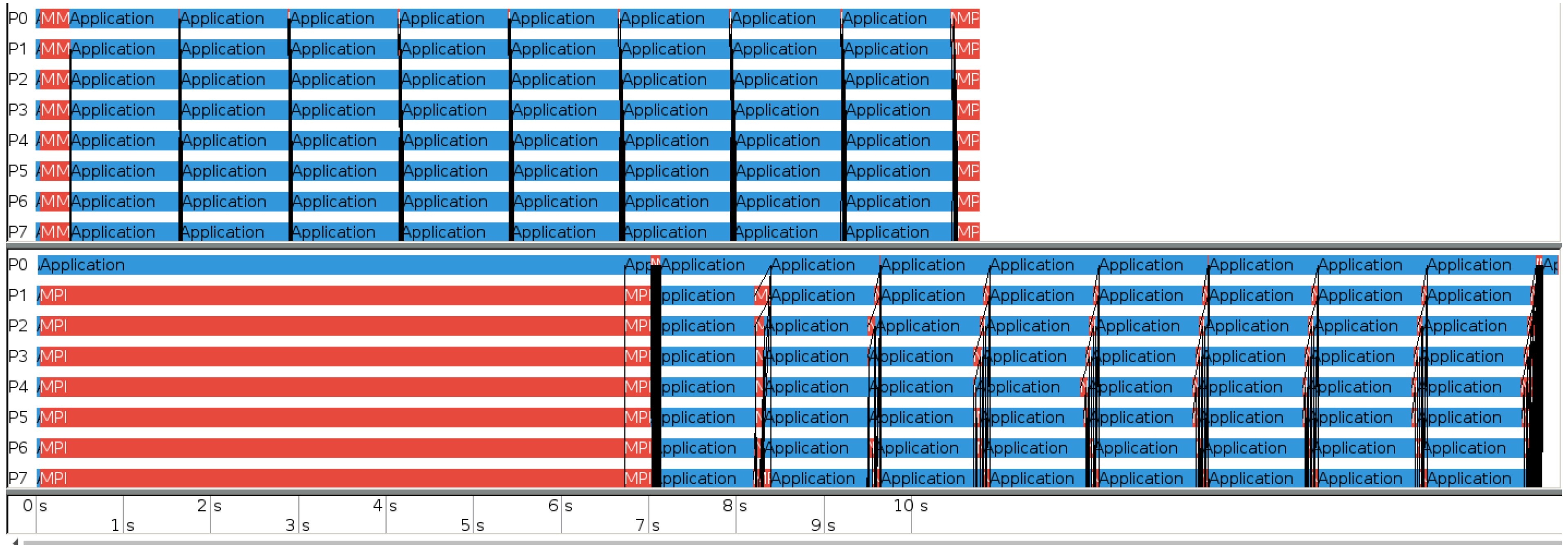


Figure 6: Comparison of trace analysis of baseline (below) and MPI IO implementations (above) on Sandybridge node.

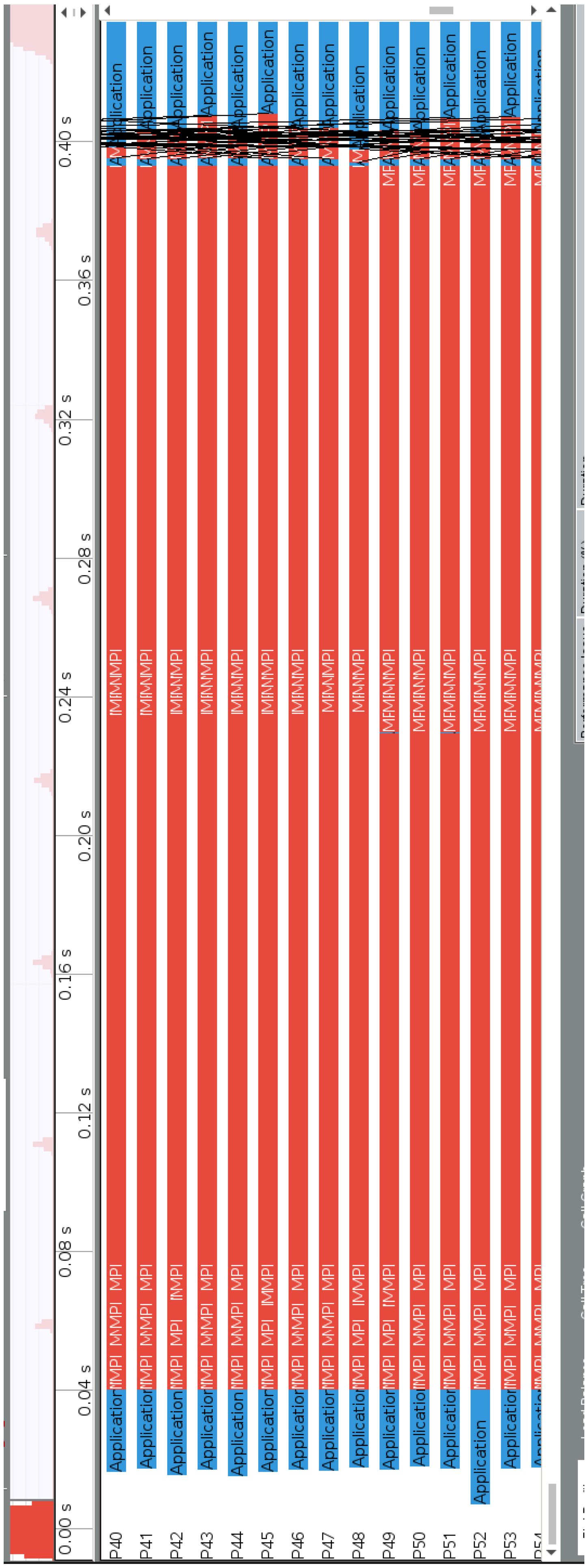


Figure 7: Detailed view of trace analysis of matrix import from MPI IO implementation on Sandybridge node.

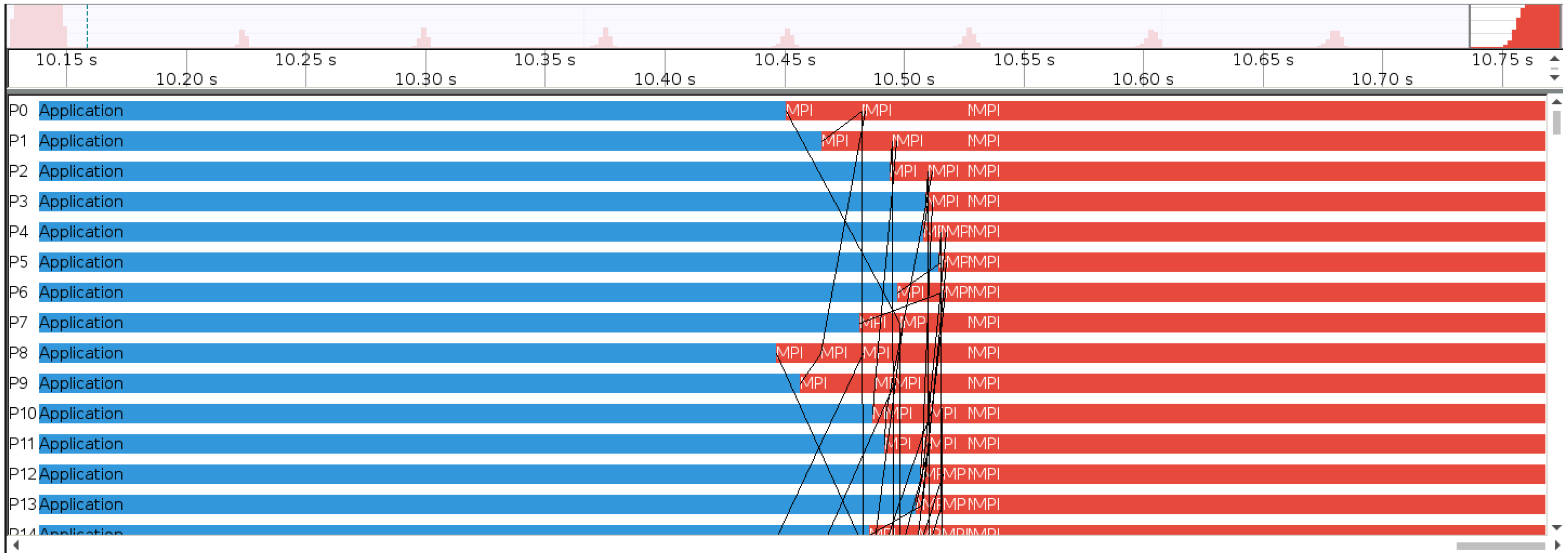


Figure 8: Detailed view of trace analysis of matrix export from MPI IO implementation on Sandybridge node.

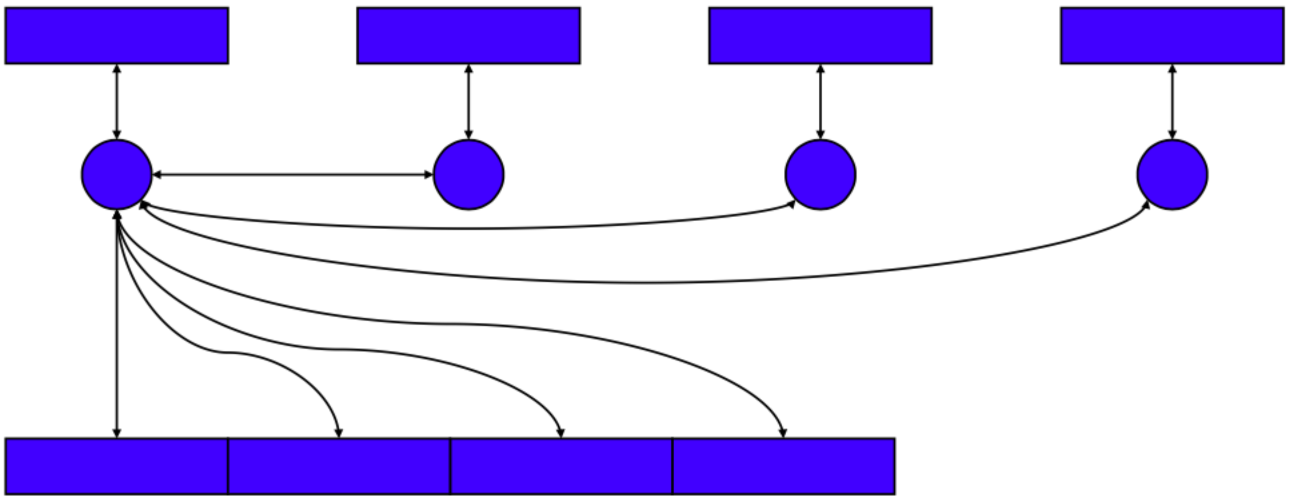


Figure 9: IO performed sequentially by rank 0 in the original implementation. Note the bottle-neck at the left-most blue circle.

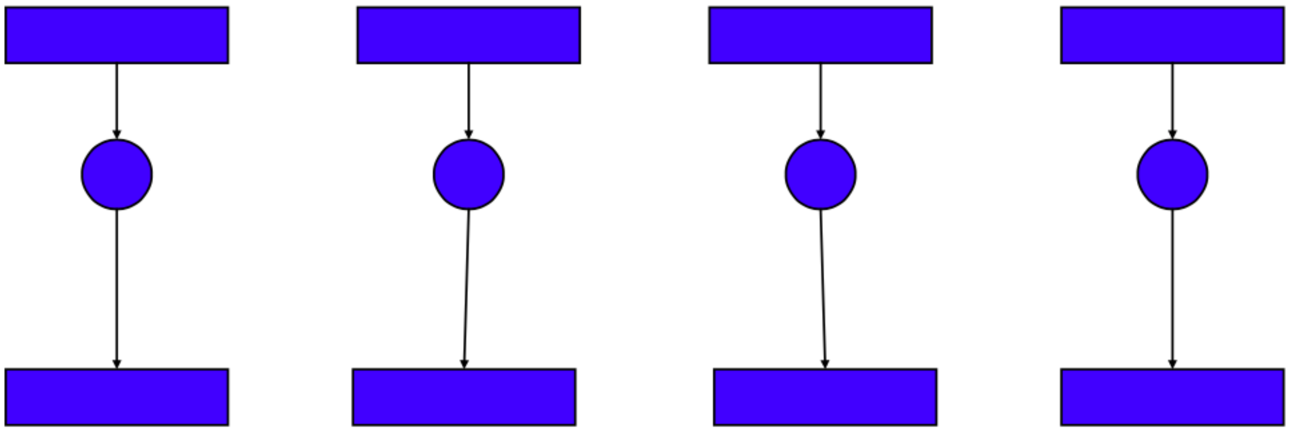


Figure 10: IO performed in parallel but independently. Note that each rank has access to their respective file.

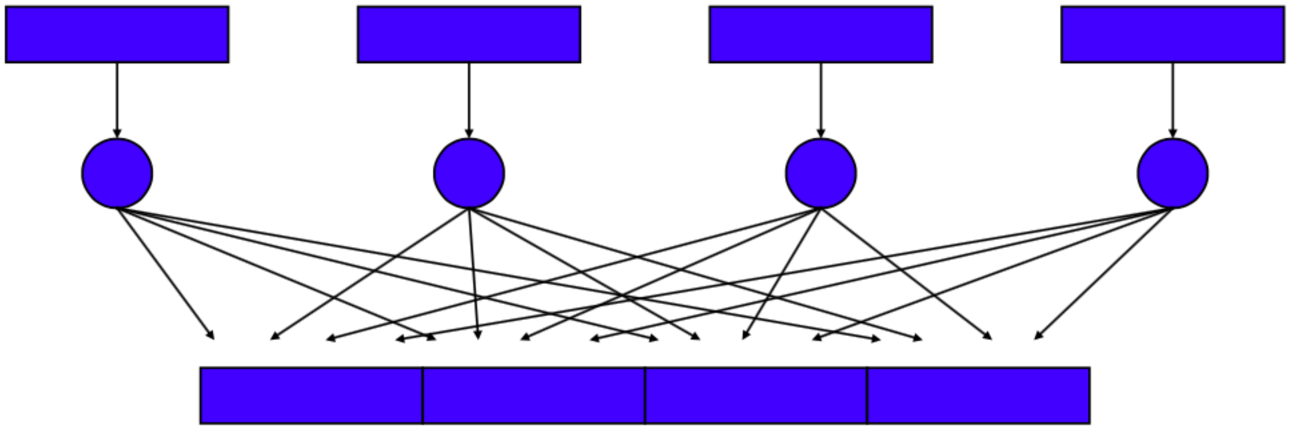


Figure 11: IO performed cooperatively by all ranks. This is what MPI IO operations can achieve. Note that all files are shared amongst all ranks.

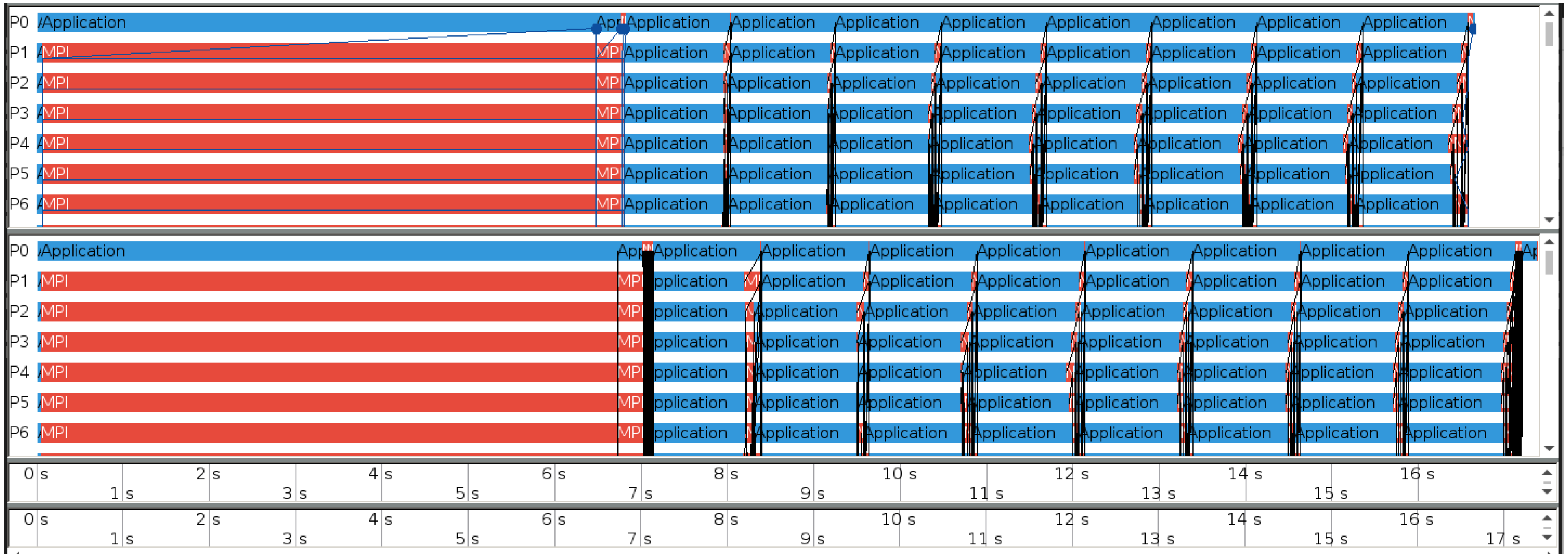


Figure 12: Comparison of cannon's algorithm using collective (above) or non-collective (below) MPI commands

References

[1] Extending the POSIX IO Interface: A Parallel File System Perspective
http://www.mcs.anl.gov/papers/TM-302-FINAL.pdf