Assignment 4 - MPI Collectives and MPI-IO

Ioannis Kouroudis, Patrick O'Connor and William Parker

January 2018

1 MPI Collectives

MPI Collectives are a set of instructions that utilise every available process of a communicator. Even when these commands can be substituted with other non collectives without loss of speed (which is rare and due to a custom special application). The syntax and command names for the collective operations are intuitive and short. This also provides the added benefit of increased readability, less chance of error and better code maintainability which can prove significant for large programs. The patterns that allow the use of MPI collectives are those where all the processes execute the same command, or all but one process executes the same command with respect to the root. This can mean various send and receive operations or a data transfer coupled with a certain mathematical operation (i.e. reduction). Performance and scalability benefits should arise from the use of collectives. This is mainly due to the fact that MPI collectives are developed by teams of professional developers over a period of many years. Although these operations are generated with general applications in mind, sometimes a programmer might achieve an increase in performance or scalability by substituting them with non collective MPI commands but this is not the general rule.

1.1 Development Task

In the cannon code there were several patterns that were ripe for improving by using collective methods. Namely, when the root needs to send the matrix dimensions to each process the observed pattern is that of a root needing to broadcast the same values to all the processors of the communicator and is hence replaced with MPI_Bcast. Similarly, when the root partitions the matrices and sends each piece to the appropriate process MPI_Scatter is used. Lastly, when the root gathers the pieces of the matrix C from the processes, MPI_Gather can be used. The blocking variant of the commands was used. Although non blocking equivalents exist, it was not deemed necessary to utilise them, as that would result in a more complicated code with more programming effort, while the projected benefit wouldn't be significant. As can be seen in Figures 1, 2 & 3), while both computation and communication time seem to scale exponentially, the computation time dominates the problem. This is partially the reason that we opted to use the simpler blocking variant of the collectives.

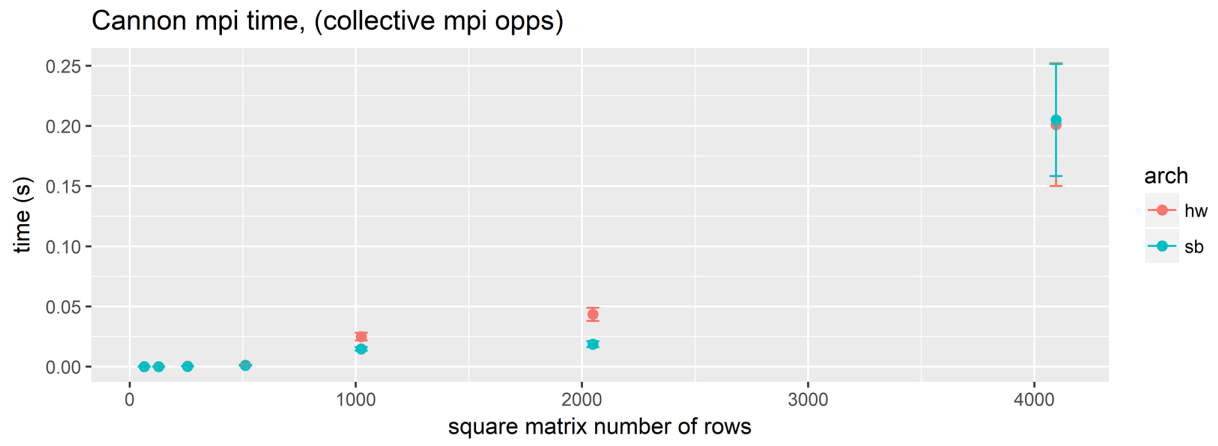

Figure 1: A graph showing the time taken to perform communication using collective MPI pattern.

Figure 1 shows the amount of time spent communicating whilst using the MPI collective operations. It should be noted that there was a marginal measured improvement, that is a reduction in communication time on the Sandy Bridge architecture as compared to Haswell. For matrices of small dimensions there doesn't appear to be any increase in communication time but as the size of the matrices increases the communication time starts to increase at a more than linear rate.

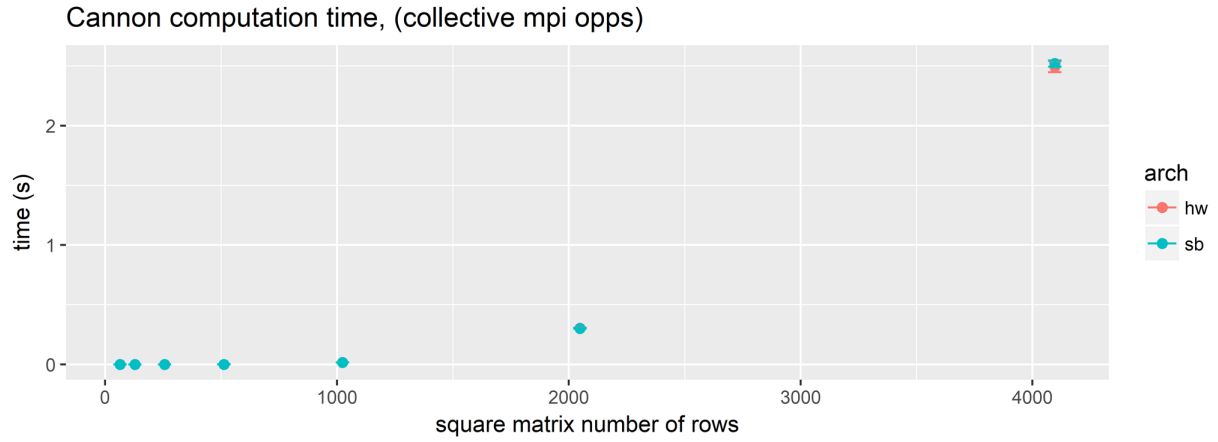

Figure 2: A graph showing the time taken to perform the computations using collective MPI pattern.

Figure 2 shows no discernible difference in computational time between the two architectures and shows that as the matrix dimension increases the computational effort appears to increase exponentially.

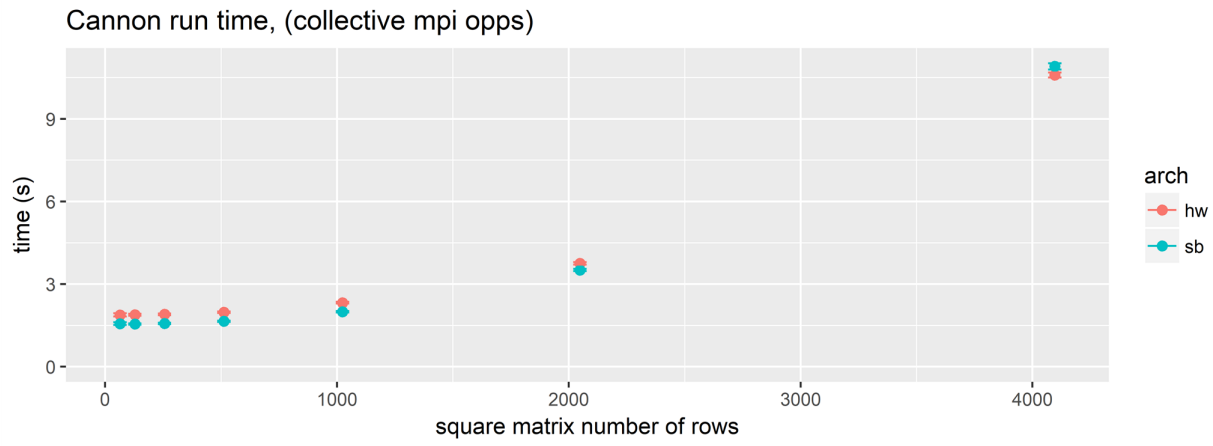

Figure 3: A graph showing the run time of the program.

1.2 Questions

1.2.1 Would you expect performance or scalability benefits from the changes in this application? Explain.

Our algorithm has been proven to be computationally bound after a certain matrix size, while in the communication bound domain the times are small enough to hide any results. Furthermore, even in the larger matrices the run time is close to 1 second thus obstructing the result comparison. It is expected that a speedup will be more pronounced in large matrices. This is evident in the following graphs. Although they don't provide any immediately usable results, combined with the actual MPI times, computation times and the cannon algorithm they can support a reasonably solid case for the use of collective operations.

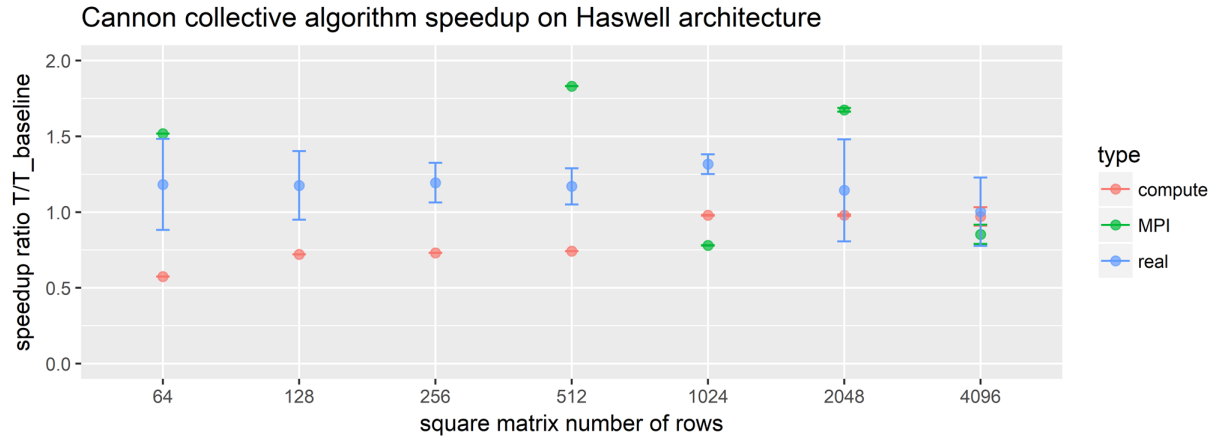

Figure 4: A graph showing the speedup provided by collective operations on Haswel with respect to the code of assignment 3.

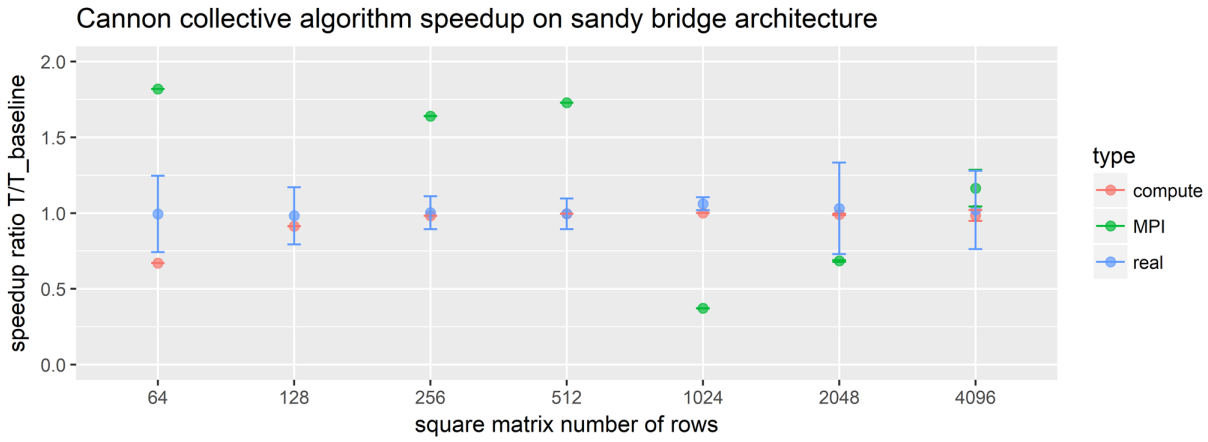

Figure 5: A graph showing the speedup provided by collective operations on Sandy Bridge with respect to the code of assignment 3.

1.2.2 Is the resulting code easier to understand and maintain after the changes? Why?

Lastly, even excluding the optimisation benefits, the code that utilises MPI collectives has increased readability and is less error prone. This is achieved by firstly decreasing the number of lines required and allowing the complicated matching communication operations to be implicitly dealt with and secondly by the use of intuitive names for the collective commands, which allow the user to understand the purpose of those lines at a glance. Overall, whenever possible it is best to use MPI collectives save for the most advanced and specific applications.

2 MPI Parallel IO

2.1 Development Task

In this task the provided MPI cannon code was transformed to use MPI-IO operations. To read and write the files the files were converted to a binary format as is required by the MPI-IO operations. It was thought that better performance would be gained by first converting the files and not including this transformation in the actual cannon code. The advantages of this were that we avoided having to keep the existing implementation of one rank communicating all the parts of the matrix and saving the file, as was previously the case. Meaning our implementation was a 'true' MPI IO collective implementation. We made use of the following MPI IO collective operations

- MPI_File_read()
- MPI_Type_create_subarray()
- MPI_File_set_view()
- MPI_File_read_all()
- MPI_File_close().

The main strategy was to move from having one rank being designated to perform all the IO, to allowing all the ranks to read and write from / to a predetermined part of the same file. Once the files were converted we were able to read the files using MPI_File_read, extracting the dimensions of the matrices to perform the computations with. These dimensions were then used to compute the local block sizes that each rank would be responsible for and operate with. Once these were calculated we were able to utilise MPI_Type_create_subarray to allow each rank to just extract the data it needs from our arrays to perform the calculations. It does this by creating a custom type that describes the memory layout of a given subarray which we determined using the

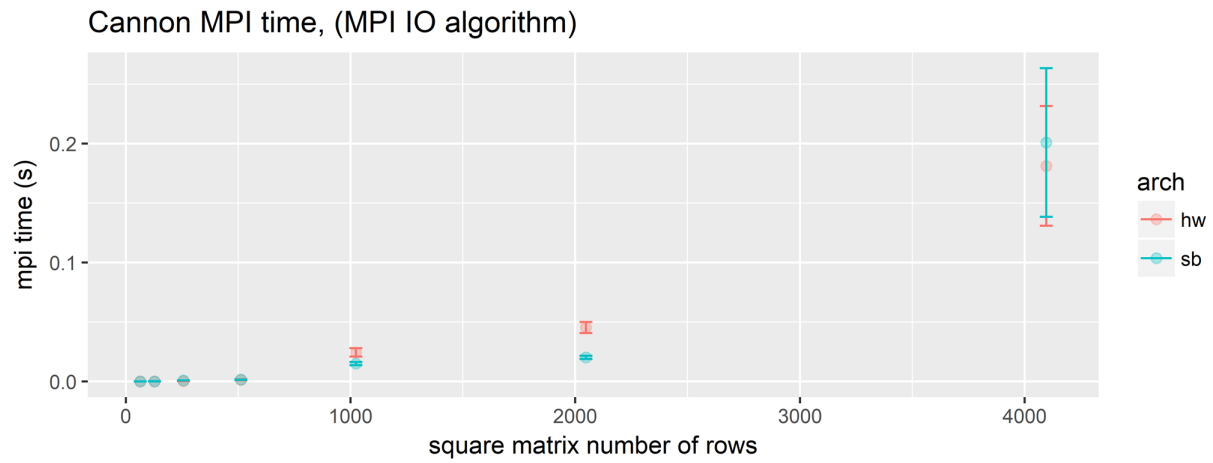

Figure 6: A graph showing the time taken communicating the matrix blocks as implemented using Cannons algorithm with MPI IO operations.

extracted dimension sizes. One thing worth noting is that in the despite as listed in the documentation here `https://www.open-mpi.org/doc/current/man3/MPI_File_set_view.3.php` when setting the datarep argument of the MPI_File_set_view() function you have to pass "native" if you want to obtain the default data value and not "NULL" as listed (at least this was our painstaking experience).

2.2 MPI time of Cannon's algorithm

This is the time that is spent communicating between ranks i.e the cycling of blocks within Cannon's algorithm. From Figure 6 we can see that for small matrix sizes of less than 1000 there is no significant distinction between the two architectures. For matrix sizes between 1-4000 there the Haswell architecture communicates significantly more efficiently than Sandy Bridge. Beyond 4000 there is again no distinction. Overall it can be observed that as the size of the matrix increases so does the time of the communication. It doesn't appear to increase linearly after matrices of a dimension size greater than 2000.

2.3 Compute time of Cannon's algorithm

This is the time spent performing the actual computations within Cannons algorithm excluding the time spent communicating between ranks. Figure 7 shows that there is no significant difference between the two architectures when it comes to the actual computation performed of the matrix blocks. We see that the compute time increases rapidly as the dimension size of the matrices becomes greater than that of 1000 due to matrix matrix product being an $\mathcal{O}(n^3)$ computation. Comparing Figure 6 & 7 we can see that the granularity here is very good, that is much more time is spent computing than communicating and that by using the MPI IO implementation we can expect our program to scale well.

2.4 Initialisation time using MPI IO

This is the time spent performing all the collective IO operations and preparing each rank so that it can perform the computation, essentially the opening and reading of the files and the division of the matrices into smaller blocks. Upon analysis of Figure 8 it can be seen that for small matrices up to a dimension of 2000, the initialisation time is almost constant, only spiking significantly after a dimension size of 4000. Even then the uncertainty on these last two points is so big that we can't draw a conclusion about how the initialisation time will continue to behave beyond this point. This again indicates that in terms of IO the MPI IO methods are very scalable and will proffer a huge benefit when used to IO operations within a program.

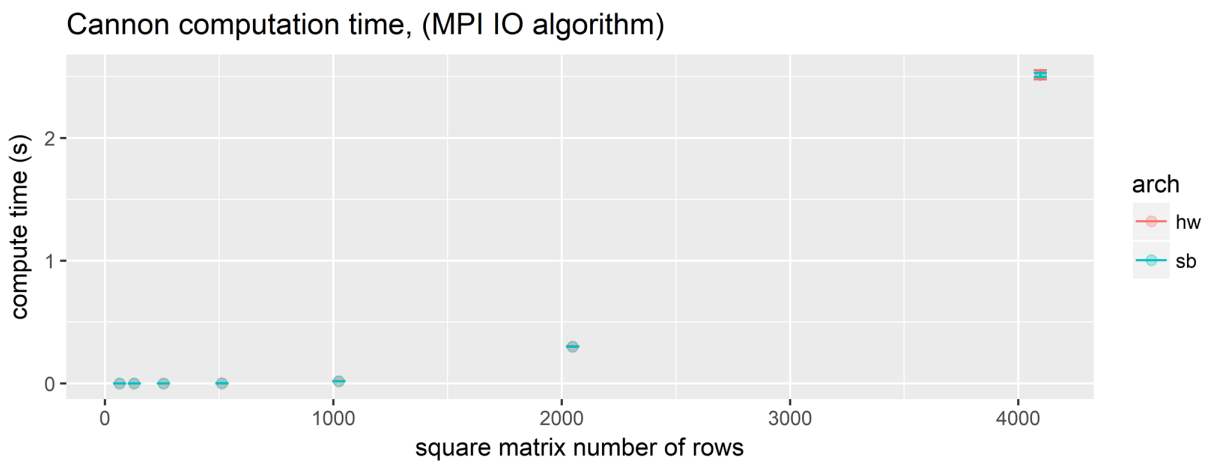

Figure 7: A graph showing the time taken to compute the matrix product as implemented using Cannons algorithm with MPI IO operations.

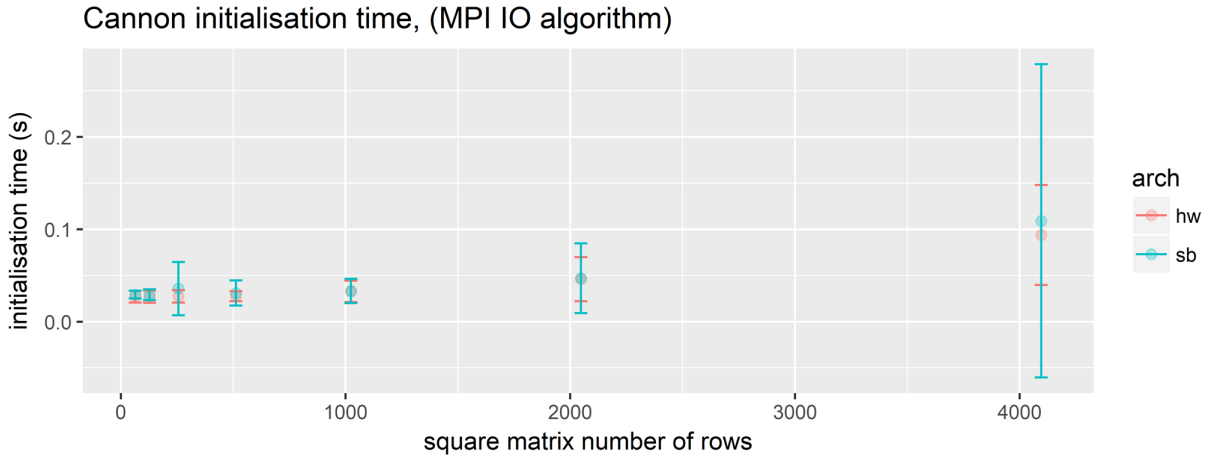

Figure 8: A graph showing the time taken to open and read files and compute block sizes using MPI IO.

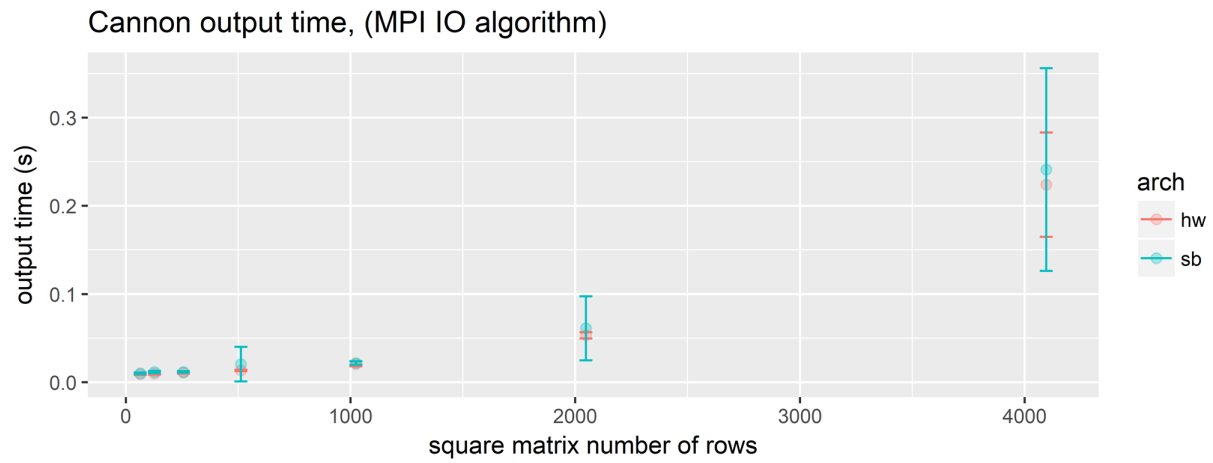

Figure 9: A graph showing the time taken to write to a file using MPI IO.

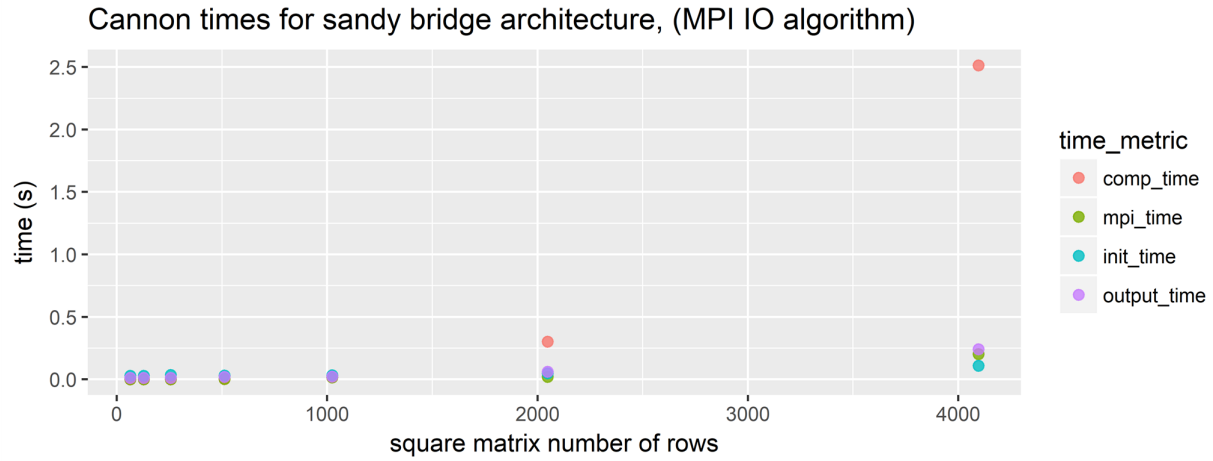

Figure 10: A graph comparing overall times of the different phases of our MPI IO implementation.

2.5 Output time

Once the computation had been performed the results were output to a file. Figure 9 shows that again for small matrices the time to write a file is almost constant only really varying after sufficiently big matrix sizes such as 2000. At this point the uncertainty increases significantly and it is hard to draw an accurate conclusion about how well the write time would scale, but we can see that the time is probably more than linear. No significant difference between the two architectures was observed for the output time and again Figure 9 indicates that adoption of MPI IO functions into a program would improve the scalability of a program and reduce IO overhead.

2.6 Scalability of our program

Overall use of the MPI IO functions within a program shows to produce very promising results in terms of scalability of that program and increases the granularity of our program significantly. Figure 10 shows that for our matrix sizes the time spent in computation is much more significant than that spent in the IO. This is desirable as we want processes to spend time computing instead of communicating.

2.7 Questions

2.7.1 What are Data Sieving and 2-Phase IO?

Data Sieving and 2-Phase IO are techniques designed to fix the overhead caused by IO. Data sieving commands a process which requires many noncontinuous parts of a file to simply make one read request instead of many consecutive ones and simply take data from the first requested byte to the last in a temporary buffer in memory. Then it extracts the required data from this temporary memory buffer thus reducing overall IO overhead [1]. 2-Phase IO is a process whereby the many processes require noncontinuous access to certain parts of the same file, but together all processes would read the whole file causing many reads. If the IO access pattern is known however then the data can be accessed in two phases. Initially a large continuous access and then all the processes can organise themselves and take which parts they need. Thereby reducing the amount of IO requests [1].

2.7.2 Was the original implementation scalable in terms of IO performance?

No, as the matrix dimensions increased so would the amount of time spent performing IO operations, this can be greatly improved by implementing the program using MPI IO operations. As shown in Figure 10 the implementation of MPI IO functions proved to be very scalable and produced a good granularity. We would advocate the use of collective MPI IO when possible to reduce the overhead of IO and increase the simplicity of MPI code. We conclude it turned out to be scalable in terms of IO overhead, ease of use and maintainability.

2.7.3 Was the original implementation scalable in terms of RAM storage?

The original implementation is in our opinion not scalable in terms of RAM storage. By relying on one rank to read and write everything, the program is inherently limited by the available storage space. Utilising the MPI IO operations, each rank only reads and writes the part of the file they need access to, which should improve the scalability here, but obviously there will always be an upper bound still.

2.7.4 How much of the communication in the application was replaced with MPI-IO operations?

A significant amount. The reading and writing of files is now completely done using MPI IO commands and so is the distribution of the data to the ranks, instead of everything being designated to just one rank. The MPI communication within the Cannon algorithm implementation itself remains unchanged, that is the cyclic rotation of the blocks.

References

[1] Rajeev Thakur, William Gropp & Ewing Lusk `http://www.mcs.anl.gov/~thakur/papers/romio-coll.pdf`, retrieved 1/2/2018.