Assignment 3 - MPI Point-to-Point and One-Sided Communication

Ioannis Kouroudis, Patrick O'Connor and William Parker

January 2018

1 Understanding Parallel Programming Challenges

This assignment describes some aspects of MPI implementation pertaining to a version of the generalised Cannon algorithm (GCA). In this restricted version of the GCA the input matrices of size A=Pxn and B=nxQ are mapped to an N^2 processor array, where the dimensions of the input matrices must all be divisible by N. The initial matrices are skewed, A by rows and B by columns. The algorithm requires n iterations with a cyclic row and column shift within A and B occurring between each iteration. To further improve upon the blocking version, two other variances were explored. Firstly the code was converted using non blocking command and secondly using one sided communication. This proved to be an improvement albeit minor, though it provided interesting insight on the working principles of MPI.

1.1 Bug Fixing and Baseline

The program was modified and compiled without optimisation. All executions were repeated 24 times to obtain statistically relevant mean execution times, and all plot error bars represent plus or minus one standard deviation. The plots below show how the MPI, computation and real time vary as a function of problem size. The log-log plot clearly shows that an initial linear relationship gives way to a power relationship for matrices greater than 1000x1000. As the computations are of the order of n^3 and communication is of the order of n^2 it can be surmised that for sizes greater than 1000x1000 the problem is computationally bound and bellow that communication bound.

1.2 Answers to section 3.2

1. :The data was incorrectly aligned in the MPI buffers. This is easier illustrated by figure 5. As can be seen, the block have the initial alignment but before the fix fail to transform to the c and d versions respectively.

2. :An MPI_Cart_shift command was used to modify the row and column source and destinations between iterations.

3. : The bug was not detectable with the given input because the homogeneity of the numbers in the rows and columns concealed the bug. This homogeneity facilitated the display of correct output despite the Cannon multiplications occurring in the incorrect sequence with within multiple MPI ranks.

4. :The compute time appears be a non-linear as a function of matrix size. This is consistent with matrix multiplication being O(N^3) on complexity.

5. :There appears to be a pseudo linear relationship between MPI time and matrix dimensions below matrix size 2000. A log-log graph of MPI time (not shown) has a very weakly steepening gradient above matrix size 2000 but the gradients are still close to zero and it is difficult to tell if a power relationship is developing.

2 MPI Point-to-Point Communication

Computation and communication are two major yet independent operations of a parallel program. Therefore, there is a much to be gained by overlapping them. To that end, a series of previously blocking commands were converted

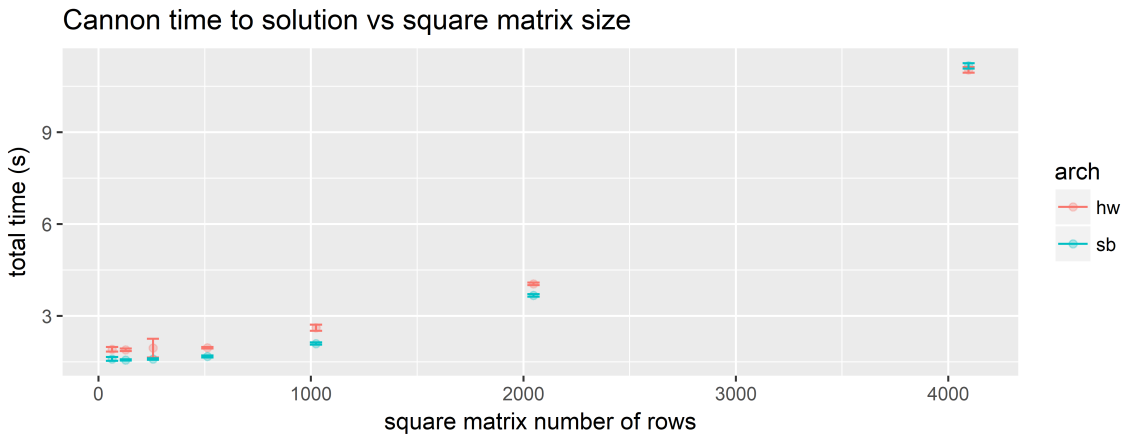

Figure 1: real time to solution vs problem size

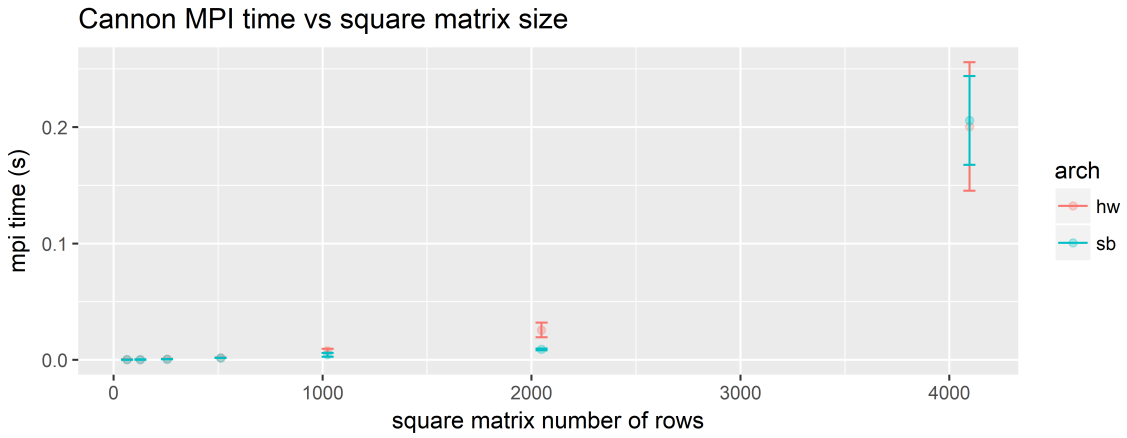

Figure 2: MPI time vs problem size

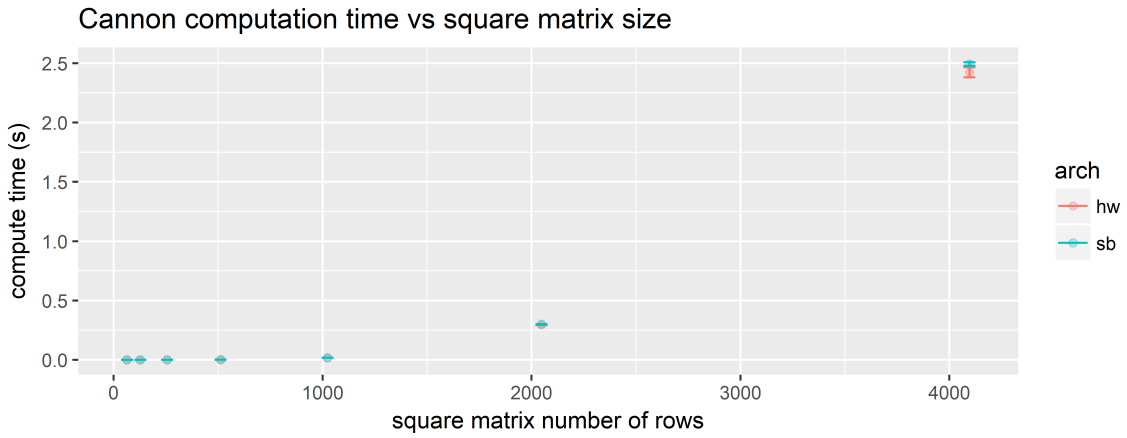

Figure 3: computation time vs problem size

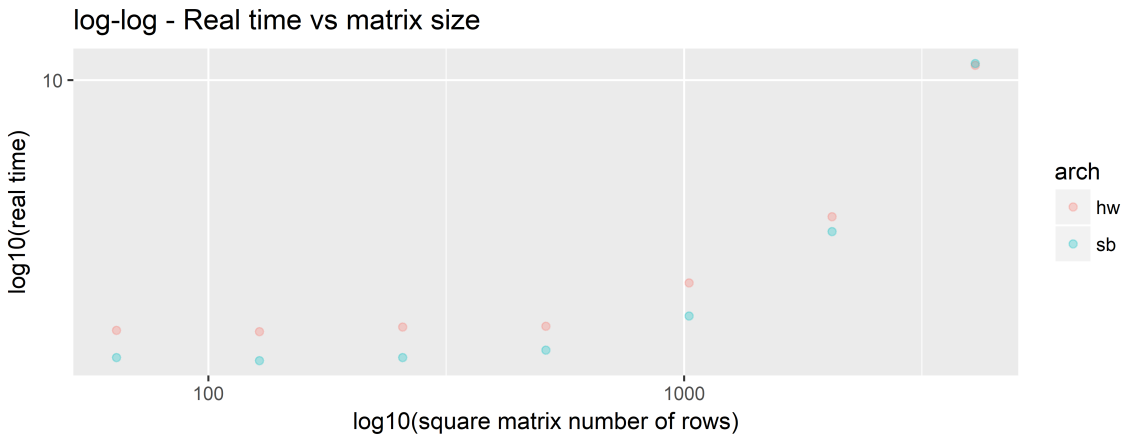

Figure 4: log-log graph showing transition from linear to exponential growth in time

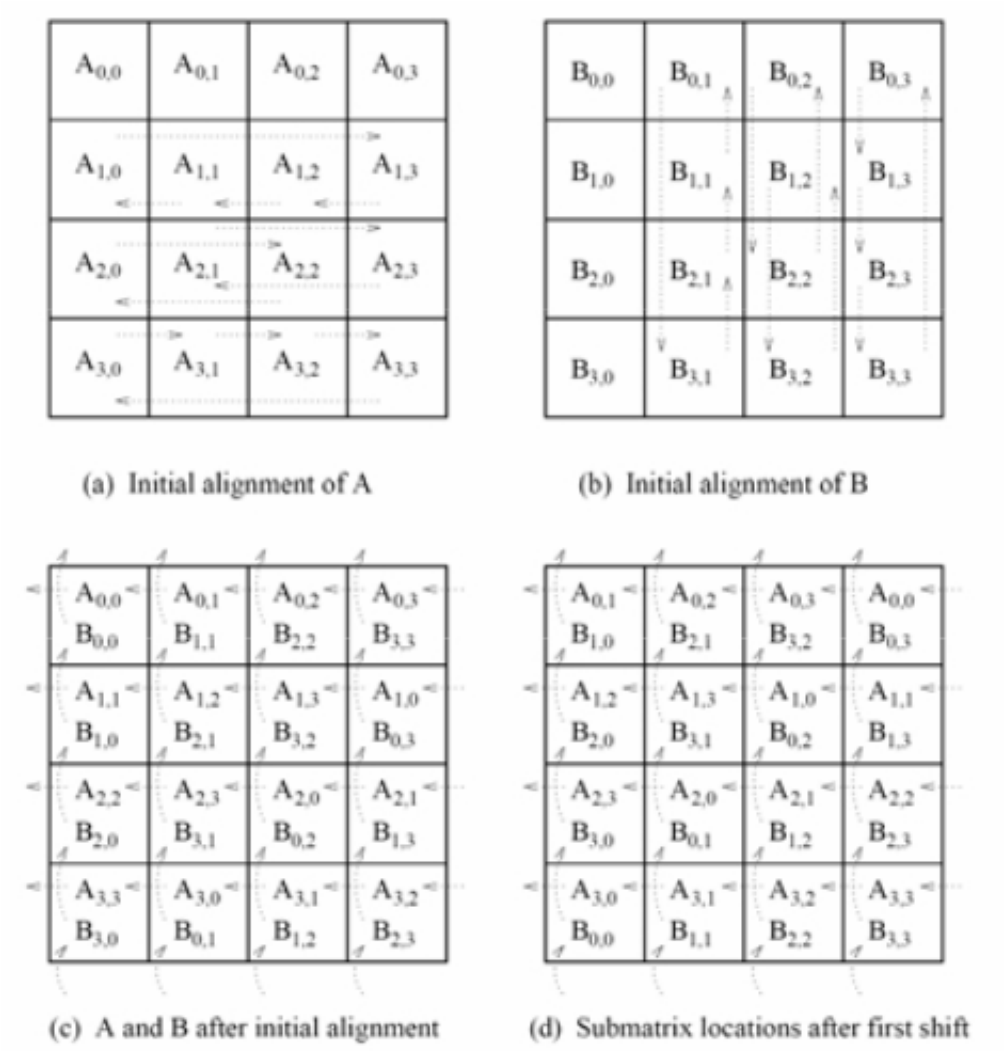

Figure 5: Initial and correct block alignment

to non blocking, to allow the otherwise idle time to be spent on computations. The largest bottleneck of this kind was identified to be in the cannon algorithm iterations. Secondarily, the distribution and gathering of the data by the root can also be converted to earn additional performance. Due to the increasing size of the sample, the second point only minorly affects performance.The approach towards the first optimisation was the following

During the algorithm we find three distinct domains. The first iteration, were the data are only sent but have already been received, the last were the data are to be received but not send and the body of the iteration were data are both sent and received. A buffer was created, in which the block that will be send will be stored when it is received. We post a send and a receive request. The next iterations are not allowed to start (blocked with a wait command) unless a new local data has been received. Once it has been received, it overwrites the current local block and the algorithm is ready to start. The buffer however remains with the previous value of the local variable, ready to sent it to the corresponding process. Once the current iteration ends, we block the process (again with a wait) until the buffer has been sent. (if it has already been sent, it will instantaneously return). Lastly, we update the values of the buffer with the current local block variables and process. It has to be noted, that before and after the bulk of the computations, the send buffer is checked. This allows us to send at best twice in a single iteration between the large computation part to minimise idle processors. In this manner, every process will be one sent further and will naturally come to a halt when the last iteration is reached.

Of the multitude of non blocking send and receive commands we chose the Isend and Irecv variant. I stands for immediate, as there is no requirement for a corresponding answer from the other process before it returns, which produces the best performance. Furthermore, due to the matrix sizes, that could potentially be very high, a user allocated buffer is used which negates the use of the B or Ib families.

The achieved overlap was, due to the circumstances of each run and the possible optimisation, lower than the theoretically maximum.

That being said, the results show a sum of communication and computation time that is smaller than the overall time. This leads to the conclusion that overlap has been achieved, albeit no substantial performance increase was noted. As a result, only a very slight speedup was observed, as can be seen in the given graphs below.

2.1 Answers to section 4.2

1. :

2. :The theoretical maximum speedup that can be achieved is a factor of 2. This assumes an initial worst case scenario in which communication and computation take the same time but happen completely sequentially. If these processes are now perfectly overlapped to run in parallel then the new time to solution will be decreased by a factor of 2 with respect to blocking case. The achieved overlap was, due to the circumstances of each run and the possible optimizations, lower than the theoretically maximum.

3. :In all cases we observed no significant speedup. The result speedup=1 was within a standard deviation error margin of the measured results. The real running time was measured by a system date call, delimited in nanoseconds, and issued before and after the execution of the program was used in the calculations.

4. :Communication and computation overlap was achieved through the use of non-blocking calls, and one sided MPI calls.

5. :We did not observe significant speedup. As the algorithm is computationally intensive, this is not unexpected.

6. :The algorithm performed slightly better on Sandy Bridge architecture for smaller matrix sizes, however it appears the advantage disappears for large matrices ¿ row size 2000. This is likely due to cache latency differences between the architectures and communication latency differences, specifically infiniband FDR14 (Haswell nodes) vs Infiniband FDR10 (Sandy Bridge nodes)

2.2 MPI One-Sided Communication

The source code was modified to incorporate non-blocking and one-sided communication. Plots of the speedup are shown below.

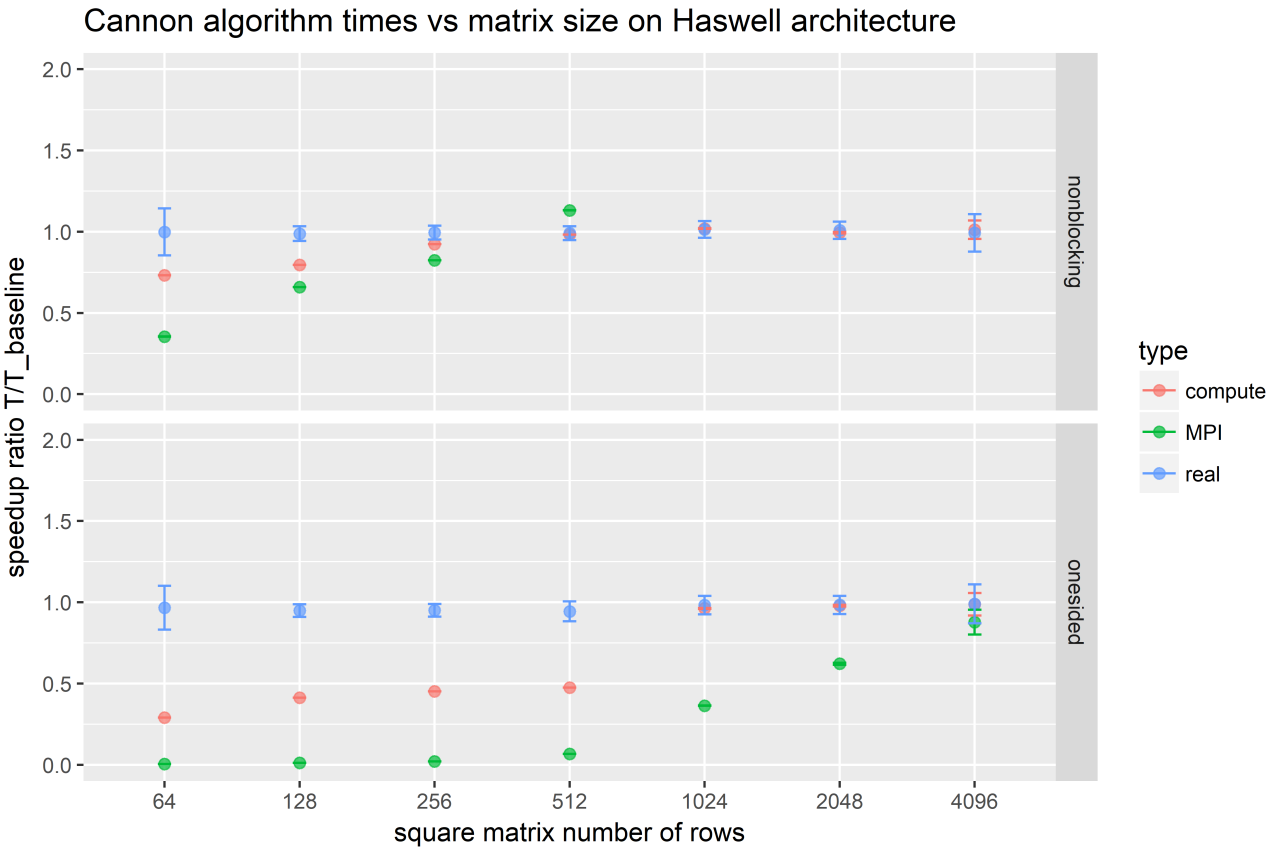

Figure 6: Speedup vs kernel version on Haswell architecture

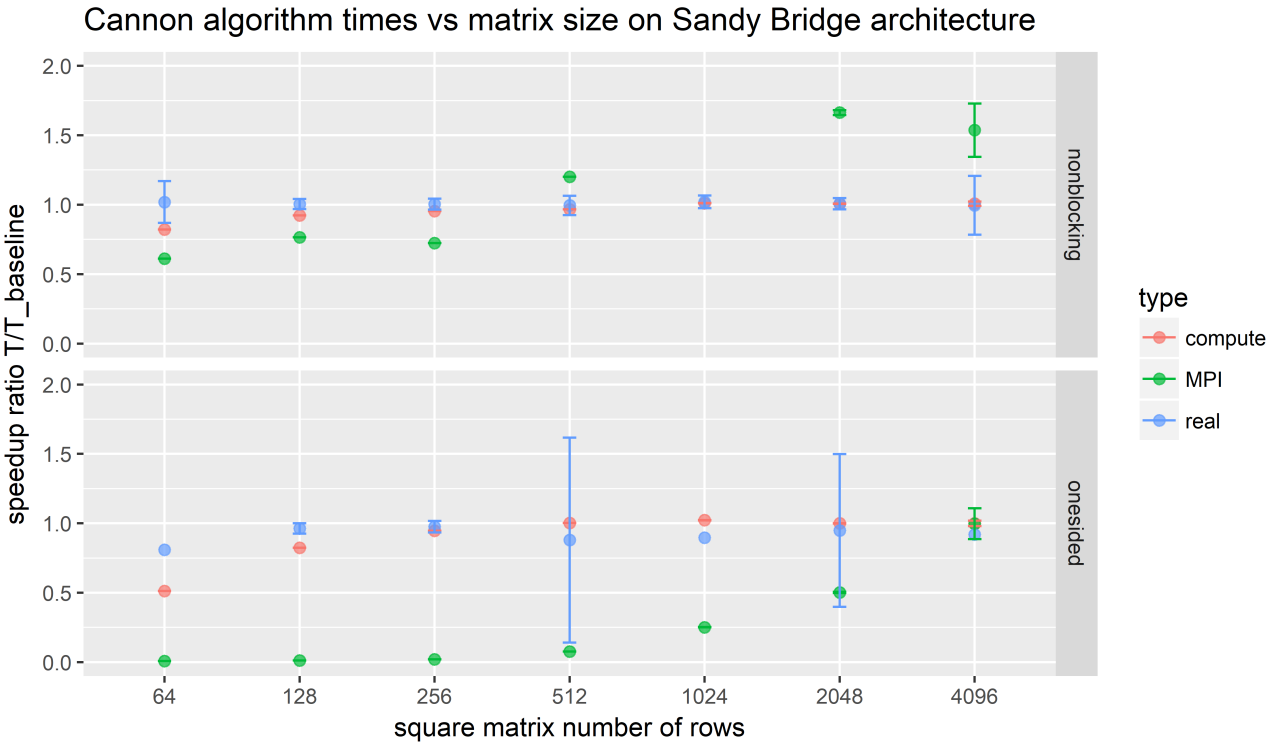

Figure 7: Speedup vs kernel version on Sandy Bridge architecture

2.3 One Sided Communication

The idea behind this program is very simple. Instead of establishing a point to point communication, a "window" is allocated to each processor. All the "windows" are connected and can be accessed by any processor in the communicator with a MPI()GET command. Before the MPI()GET or MPI()PUT commands, a barrier is created so that all processes about to access the window are synchronous. Effectively, all the processes put something in the respective barriers of their counterparts. Wait until the processes has been synchronised. Take what was put into their window and resume calculations. Every put and get was prefaced by a barrier (fence) to negate possible race conditions.

2.4 Answers to section 5.2

1. : The following commands were used.

   MPI()Win()create which is a collective call executed by all processes in the group of comm. It returns a window object that can be used by these processes to perform RMA operations MPI()Win()fence which Synchronises RMA calls on a window.

   MPI()Get which allows the reading of the shared buffer

   MPI()Put which allows the writing to the shared buffer

2. :As can be seen computation and communication overlap were achieved, though nowhere near the theoretical maximum.

3. : As can be seen by figure 6 the computation and communication times were increased. The total time however remained stable within the margin of error. This implies the existence of overlapping, as the additional overhead was absorbed by the parallel execution of communication and computation.

4. :there was no observed speedup vs baseline in real time. The speedup from overlapping communication and computation was matched by increased overhead.

5. :No significant speedup versus the baseline was observed. The introduction of a buffer on the heap involves a substantial amount of computational effort and resulted in slow memory access which may have served to negate any beneficial overlapping effects.

6. : As can be seen there was a minor speedup with respect to the non blocking version. This might be due to the fact that the "window" access each processor has decreases the initialisation overhead of establishing a communication. This effect however was negligible.