Message Passing in High Performance Computing

Mark Ergus Nicholl[1]

University of Bristol
[1] *mn16660*

December 14, 2018

In the previous assignment, we performed serial optimisation to optimise the code running on 1 process. Now, we take advantage of the fact that each node can carry out multiple processes with their own memory space in order to speed up our implementation of stencil. The obvious area to explore in this task is the optimisation of the bandwidth of data moving between processes. For this purpose, I will be comparing and analysing my implementations throughout the duration of this coursework. For this purpose, I will first compare running times in various core and size combinations to compare implementations and then proceed to analyse the scalability of my final code from 1 core to 16, in increments of 1 core.

1 MPI Optimisation and Implementation

1.1 Domain Decomposition

In order to perform MPI optimisation, the first step to take is to decide how to divide the data among processes. The pros and cons of each technique very much depends on how our original code stores and accesses data. To give context to the arguments ahead, my serial stencil code stores and accesses data through a 1-D array. This 1-D array can be imagined as a column major 2-D array, where cell[i+1] is below cell[i] and cell[i+length of columns] is to the right of cell[i].

Column VS Row Decomposition The 2 most basic ways to decompose and distribute the data among processes is either by decomposing row-wise or column-wise. This decision has huge implications on run-time and performance. Performing row decomposition of my original stencil code yielded poor performance, and at times was detrimental to run-time. The problem was not too apparent in the smaller cases with less cores, as operating on the 1024x1024 image with 4 cores took 0.10s compared to 0.12s with the optimised stencil code. However, operating on the 8000x8000 image with 16 cores took 27.65s, roughly 7 times the time it took for the optimised stencil.
One of the reasons is a large number of messages being sent between processes. As the rows were not contiguous, it was not possible to send the image in 1 message function. As a result, a large number of messages must be sent to exchange halos. In the case

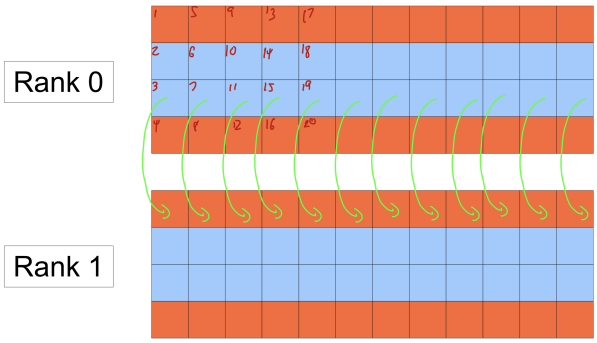

Figure 1: *Visualisation of a halo exchange with row decomposition*

of the 8000x8000 image, this was 8000 messages per

rank, and with 16 ranks performing 4 halo exchanges per iteration, 512,000 send and receive calls had to be called. This obviously affected run-time massively. Whereas in a column decomposition, only 1 message call sending a packet the size of a column is needed per rank. This reduces the messages to 64 send and receives per iteration. This improved the problem massively, and could now complete the 8000x8000 problem with 16 cores in 2.23s. This was a 12x speedup over the row decomposition code and a 4x speedup of the serial code. While this is true, amount of bits sent remained the same in both situation. It is important

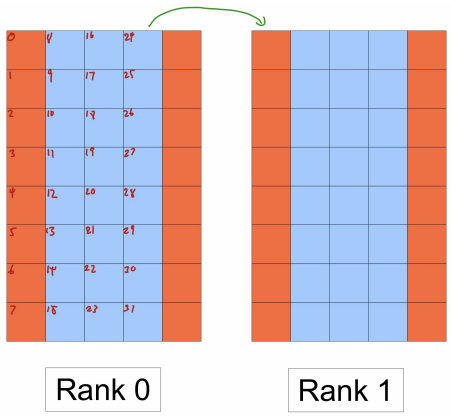

Figure 2: *Visualisation of a halo exchange with column decomposition*

to note that this is not true in all cases. It may be the case that sending a larger number of small packages works faster than singular, larger packages. In fact, the latency of a smaller package is lower than that of a larger package. In reality, a lot of factors can affect the outcome such as communication fabric, distance between communication targets, single node or multiple node or even communication type used. In this specific case, it could be slower due to congestion in a single core combined with the blocking mechanism of MPI_Ssend.

Cartesian Topology Ideally, a Cartesian topology should be able to improve performance even more. The improvement from applying this method stems from the fact that it is a 'virtual' topology. In general, MPI is allowed to reorder the ranks, so that network proximity in the cluster corresponds to proximity in the structure of the code[1][2]. With this, we get to use the full advantage of locality of data while not dealing with its complexity. Also, now having halos of different directions, we can avoid congestion among processes. Personally, I would predict that this would improve performance of the program, especially when scaling up, as load balancing and congestion will be handled better. Unfortunately, I did not have the time to implement this well hence choosing to stick with column decomposition.

1.2 Communication Method

Blocking Synchronous To begin this task, I went with the safest form of message passing, MPI_Ssend. As it blocks and is synchronous, it is easy to understand the process and to understand the correct message passing logics in the code. As this is true, it is vulnerable to deadlocks but this is useful as we know there has to be a mistake in the mechanism if a deadlock occurs. However, this method is possibly the slowest and there is more potential in the code. In order to proceed, I decided to look towards faster methods of message passing.

Blocking Asynchronous The next method tried was MPI_Send. MPI_Send is not strictly asynchronous, in fact it is only potentially asynchronous. It relies on the availability of a system buffer in order to perform asynchronous message passing. Performing tests on the same code while only changing the send methods in the time critical section, there was a moderate increase in time. For example, the 8000x8000 on 16 cores performed in 1.95s, a 1.15x speedup from before. This speedup, although small, was consistent on multiple runs. This would indicate the availability of a system buffer on the machine's architecture. This would help the code speedup slightly as processes performing a send do not have to wait for a matching receive to be posted, instead it can move on with the computations ahead. In a synchronous message send, the process will always wait until the receive is posted. This could leave processes at idle, and slower processes will be a bottleneck in the code.

Self Implemented Blocking, Asynchronous In order to maximise performance, implementing a non-blocking, asynchronous message send and receive (MPI_Isend and MPI_Irecv) seemed like the rational direction to go. In theory, this would be the best chance to overlap computation and communication and could be the key to good scaling. However, given that I have maintained the structure of the stencil code without special buffers, the code still needs to be blocked with a MPI_Wait function to allow safe data access from iteration to iteration. As this was the case, there was minimal benefit to using this. Speedups recorded were less than 1.05x in all cases. This result was probably due to the structure of the stencil code. The appropriate measure to ensure this implementation work would be to reorder the code in such a way that the process could meaningfully carry out computation without waiting for its send. This way, all processes will not maintain at idle and could probably perform faster to a significant degree.

2 Comparison of MPI Code and Serial Code Performance

cores	1024x1024	4096x4096	8000x8000
Serial	0.12s	2.55s	9.18s
1	0.16s	3.32s	10.90s
16	0.011s	0.053s	1.95s

Overall, on single processes, the serial code ran slightly better than the MPI code. However, when we allocate more processes, the performance increases significantly. The speedups for the 1024x1024, 4096x4096, and 8000x8000 are 11x, 48x and 5x respectively.
The first interesting trend is the 1 core MPI vs the serial code. This could be due to issues such as minor code manipulations to accommodate for edge cases in MPI or overhead of memory allocation for buffers or local image space.
The second interesting trend is the 16 core MPI vs the serial code. Overall, we can see that it performs much better when multiple cores are used. As we spread the load and 16 array entries can be calculated at once, there is no doubt that it is more efficient than running the code serially.
The third trend to notice is that the 4096x4096 image speed up was more than 4 times more than the 1024x1024 image, while being about 9 times of the 8000x8000 image. This could indicate a few things. First of all, this could indicate that the message passing performs more inefficient the more we scale up. This makes sense as since we are packing the whole column into 1 message, the latency of packing and sending 8000 floating points would affect the performance. Second of all, the speed up of the middle sized image compared to the small one could indicate that given the code structure and message passing algorithm, the middle sized image have a better processing speed up benefit to message passing overhead ratio. From this, we can conclude that in terms of image size scalability, the code performs at its best somewhere in between 1024x1024 and 8000x8000.

3 Scalability Analysis

As we increase the number of cores, the performance of the code will differ. Below is the graph of time achieved against the number of cores for the 8000x8000 image :

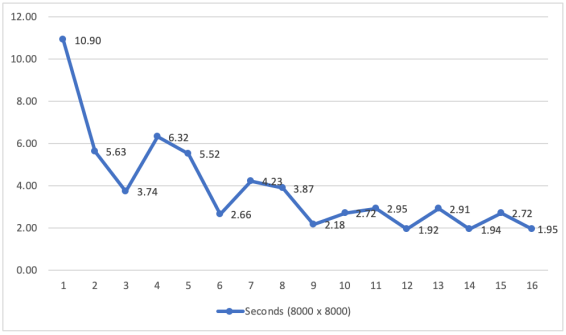

Figure 3: *graph of time achieved against the number of cores for the 8000x8000 image*

Decrease In Time As Core Increases As we increase the number of cores, the time taken for the program to run completely generally decreases. This is due to the fact that the local image in the individual rank that the process needs to compute gets smaller as we increase the number of processes. At 16 cores, each rank has 4,000,000 array entries to calculate per iteration compared to 64,000,000 with 1 core. This means that each rank only has to process 1/16 of the image, and as this is done concurrently, the program saves time. It is not quite as equivalent the time taken to run on an image 1/16 the size serially as message passing incurs overhead on the program.

Diminishing Returns The increase in performance generally decreases as the number of cores increases. After 9 cores, there is practically no increase in performance as we increase the number of cores. First of all, as we divide the columns among the cores, the number of columns reduced per rank decreases the more cores we use (from 10 cores to 11 cores, each rank holds 73 columns less, whereas from 1 core to 2 core, each rank holds 4000 columns less). Also, paired with the fact that the more ranks we have, the more messages we have to pass (as we increase a core, we need to send an additional 4 messages with constant size of 8000 floating point values), we can see why the increase in performance decreases then stagnates. The trade off of more processing power to more message passing decreases as we increase the cores, and once we saturate the node with packing, message passing and receiving, we reach a bottleneck. The code now depends on how fast messages can be sent and received instead. We have effectively nullified the benefit of message passing, and should look to improve the way in which we exchange messages to gain more benefit in performance.

Bump In Values Along The Line At certain values, there are bumps in performances. These bumps are significant in value, therefore would be far fetched to be attributed to noise in the data. Instead, there are a few ideas of where this might come from. At most odd core values, there are spikes in performance. As the load of columns per rank is distributed in a way where the last rank receives the remainder of columns if the size is not divisible by the number of cores, this could cause the last rank to be the bottleneck in each iteration. As the messages are blocking, the processes will be left idle waiting for the last rank to finish calculating.

4 Analysis and Conclusions

Taking the best performer, as we calculate our operational intensity (6/((5r * 1w)*4) is 0.25 FLOPS/byte and our single precision performance (4096*4096(array)*100(iterations)*2(calls per iteration)*6(single precision floats)) divided by the time taken (0.053s) as roughly 380GFLOP/s, we can see that we are at the best case still far from the theoretical peak performance of 665.6 GFLOP/s. Given the low operational intensity, we are still memory bandwidth bound. Hence, future development should focus on managing cache within ranks, better organisation of data and more efficient message passing.

5 Future Improvements

Techniques such as hybrid programming with OpenMP, Remote-Memory Access, Shared Memory and multiple node MPI could be explored to increase performance.

6 References

[1] http://pages.tacc.utexas.edu/ eijkhout/pcse/html/mpi-topo.htmlCartesiangridtopology
[2] https://cse.buffalo.edu/ vipin/nsf/docs/Tutorials/MPI/mpi-advanced-handout.pdf