COMS30005

Parallelism with MPI

Louis Heath

lh16421

December 12th, 2018

1 Introduction

This report aims to outline my investigation into optimisation via parallelisation. I apply various techniques to my serially optimised *stencil* code, such as Single Program Multiple Data, and also consider message passing improvements to reduce communication costs.

2 Optimisation

2.1 Domain decomposition

The first approach taken was to divide the stencil image between multiple worker nodes. After each iteration, the workers must engage in a *halo exchange* in order to update the edges of their image. We find that the way in which we make this division has a significant impact on performance.

I chose to begin by decomposing the image into columns. This made the most sense to me as the image array itself was indexed vertically, with *image[1]* being the cell output directly beneath *image[0]*. This way we are working with one contiguous region of memory, reducing access times and improving cache hits.

This method proved effective, reducing run times to 0.008, 0.53 and 1.94 seconds for the three respective images. Comparing these to the serial timings of 0.1, 2.4 and 8.4 seconds, we see speedups of 12.5X, 4.5X and 4.3X respectively.

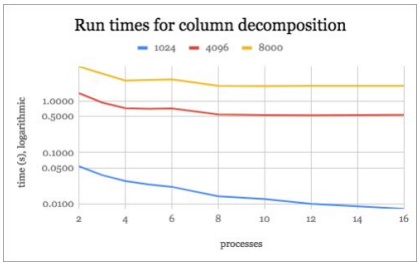
Figure 1: *Run times for column distribution (s)*

ppn	1024	4096	8000
8	0.0143	0.5449	1.9454
10	0.0127	0.5296	1.9416
12	0.0102	0.5282	1.9577
14	0.0092	0.5329	1.9447
16	0.0082	0.5362	1.9527

From *Figure 1* we see that the run times for the smallest image were best at 16 processes, whereas the larger two were most optimal at around 10 processes, marginally losing performance from there on. This plateau is visualised in *Figure 2*, which suggests that the 1024 image could benefit from more processes than the Xeon E5-2670 can provide.

We also see a kink in the curve when six processes are applied. This kink is more prominent on larger images, suggesting that it is due to my method of catering for non-divisor cohort sizes. I simply take the remaining *n* columns and share them amongst the first *n* ranked processes. If I divided these remaining columns horizontally as well vertically, I could have allowed a more even spread of work, resulting in less idle communication.

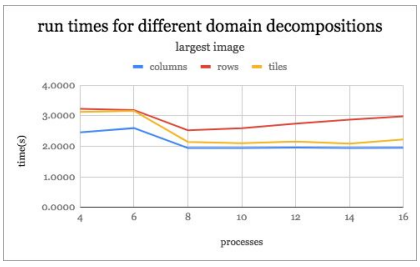
Figure 2



For sake of comparison, I produced a second implementation using row decomposition, and a third which utilising a cartesian grid to divide the image into tiles. A plot of their performances for 4 to 16 cores is shown in *Figure 3*, from which we see that the column decomposition was most effective.

The plot for rows can be seen to peak at 8 processes, consistently degrading from then on. This could be because as the row divisions get narrower, each sub-column of contiguous memory gets smaller, resulting in gradually more cache-misses each time.

Figure 3



The row decomposition also has a higher minimum time of 2.52 seconds, 1.3X slower than columns. In addition to the cache-hits factor during computation, this could also be due to larger memory access overheads when preparing a send buffer, as each cell is located *numy* apart in memory.

The tile implementation performs similarly to rows, though plateaus marginally higher at around 2.1 seconds. We also see its time increase at 16 processes, as this is the first case of division into 4 rows. For example division by 14 would result in 2 rows and 7 columns. Similarly to the degradation of the row decomposition, this decrease in performance could be due to an increase in cache-misses.

We consider the ratios of cells computed on vs cells communicated, assuming *width == height*. We find the ratio for column and row decompositions to be

$$\frac{(\frac{width}{cohort\ size}+2)*\ height}{height*2} \approx \frac{width}{cohort\ size}$$

And the ratio for a cartesian grid of *sx* columns and *sy* rows, i.e. *cohort size = sx * sy*, to be

$$\frac{(\frac{width}{sx}+2)(\frac{height}{sy}+2)}{2\frac{width}{sx}+2\frac{height}{sy}} \approx \frac{width}{2(sx+sy)}$$

This means that for small cohort size, row and column decompositions are coarsely grained, becoming less so linearly with more processes. In the case where *cohort = sx * sy > 2(sx + sy)*, tiles are more coarsely grained than columns or rows. Considering all even cohort sizes from 2 to 16, this only holds for 16. In all other cases tiles are more finely grained.

This is supported by *Intel Trace Analyser*, which finds that the column decomposition spends the least time on communication, and that tiles spend the most, as seen in *Figure 4*. The row implementation is in the middle, likely due to its non contiguous message buffers as mentioned before.

Figure 4: *Time spent on code vs MPI*

	columns	rows	tiles
code	90.30%	87.50%	83.70%
mpi	9.60%	12.40%	16.20%

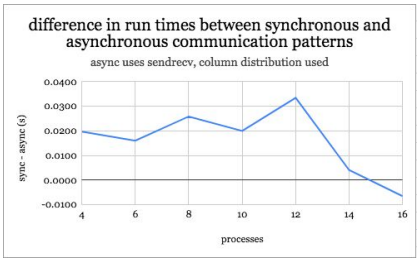
2.2 Message options

The MPI standard provides many options for point-to-point messaging. In the implementations mentioned so far I have used synchronous *Send*, *Recv* and *Sendrecv*.

These functions utilise a buffer to hold the message while in transit, allowing the sender to continue executing code. Synchronous functions however are blocking, as they wait for the message to be received. This comes at the benefit of portability and reduced memory overheads, as the message does not have to be copied an extra time.

Implementing a synchronous halo exchange on my row decomposition code resulted in a very marginal decrease in performance, as can be seen in *Figure 5*. The synchronous code performed around 0.02 seconds faster on average, though there was a large amount of noise in the recordings.

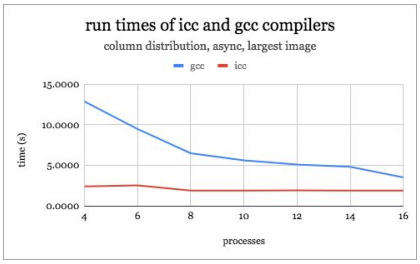
Figure 5



2.3 Compiler options

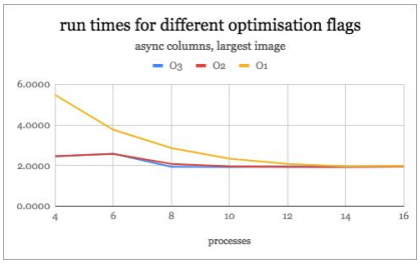
Similarly to the last assignment, compiler choice had a large impact on performance, as can be seen in *Figure 6*. Intel's compiler evidently makes a larger impact on more finely grained systems, suggesting that it includes more optimisations for communication.

Figure 6



Optimisation flags also play a key role in MPI applications. As can be seen in *Figure 7*, the O1 flag looks very similar to the curve drawn by *gcc* in *Figure 6*. This suggests that the two configurations provide a similar level of communication optimisations.

Figure 7



In attempt to yield further optimisations I applied various pragmas provided by Intel, such as *unroll and jam*. These pragmas simply override certain decisions made by the O3 flag, however, and so failed to improve my run times.

3 OpenMP

OpenMP allows us to parallelise code within one process. A serial program can split into multiple threads, which have both private and shared memory. This allows for quick parallelisation however is less scalable, as memory is shared.

I was able to apply OpenMP to my serial code in two ways; firstly by splitting up the image into columns and allowing each thread to work on its portion, and secondly by simply adding a for loop pragma to the innermost loop of the *stencil* function.

Figure 8: *Run times for serial and OMP*

method	1024	4096	8000
serial	0.1000	2.3900	8.3900
columns	0.0084	1.9970	4.6244
for loop	0.1027	1.0194	3.7719

From *Figure 8* we see that the columns split yielded similar speedup to that of MPI for small images, but was a lot slower for large images. This is because OMP is limited by memory, and so struggles to operate efficiently on the larger 8000x8000 image. The for loop pragma proved more effective than the column decomposition, and both were faster than serial.

The for loop pragma provides different scheduling options, such as *static*, *dynamic*, and *guided*. The timing recorded in *Figure 8* utilised guided scheduling, which aids with load balancing. Each thread takes a chunk of iterations from a queue. Once complete it takes a new chunk, smaller than the first. This downward scaling maximises the chance of all threads finishing at similar times.

Thread safety is very important when working with OpenMP, as two threads could operate on the same region of memory simultaneously. To avoid this in my columns decomposition I used the *barrier* pragma between each *stencil* call, ensuring that each iteration was fully complete before using the data again.

4 Derived Types

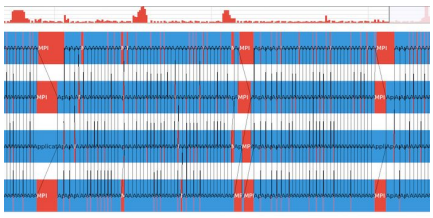
In attempt to reduce the time taken to construct halo message buffers, particularly in the row decomposition where the buffer was non-contiguous, I constructed an *MPI_Type_vector* to automate the packing process. This assumes one regular data type and a constant stride between each element, which was ideal for halo exchange. The change yielded no speedup, however.

Similar functions, such as *MPI_Type_struct* allow for multiple data types of varying size, and displacement, however were not useful in this case.

5 Performance analysis

Though happy with my optimisations, there is definitely room for improvement. *Intel Trace Analyser* diagnosed my problem to be excessive wait times in-between *Send* and *Recv* calls. In particular there were occasional regions of excessive computation, perhaps when data is moved between caches.

Figure 9: *Red blocks indicate MPI waiting time*



Throughout the assignment I found there to be huge variance in my recordings. This is likely because each process is distributed randomly across the chip, resulting in different latencies each time. Finding a way to bind processes to cores by rank would provide more reliably fast timings.

My final times were 0.008, 0.53 and 1.94 seconds for the three respective images, corresponding to speeds of 236, 57 and 59 GFLOP/s.