Message Passing Interface and Parallelism

David Sharp : ds16797 : Candidate 36688

December 2018

1 Final Implementation

My final implementation that I coded with MPI for the 5-point stencil task distributes columns out between processes - leaving any remainder with the highest rank process - and implements a 2-D halo-exchange cyclicly using blocking combined send/receive calls. It then copies the answer from the write-too array ($curImage$) to the read-from array ($preImage$) and repeats the iteration for the required 200 times in the test cases. It checks to see whether the inputted image size has more rows than columns and makes sure it distributes over the larger by swapping the array dimensions if necessary.

This is what occurs in the case that the size of image is comparatively small, when the image is larger instead I ignore the answer copying and instead just do two iterations effectively 'in-place' similar to the serial stencil code. I think due to the higher overhead of copying an array at higher sizes this strategy performs better.

After the stencil is complete, the master rank takes charge of taking back in the answers from the 'student' processes. It allocates space for an array to store each of the student's answers in, then places its own answers in the array and sets up to receive each student's answers column by column and place them in the same array. It then feeds the array into the serial image output function. This outputting process takes by far the bulk of the actual runtime of the program, particularly for very large images, though not measured for this assignment.

1.1 Implementation Decisions

There were several implementation paths where the results surprised me, the first being that I assumed that my method for distributing rows out to the processes was flawed in that the last row would end up with more work than the rest of the processes and would thus slow the entire program down due to synchronous communication throughout the program. I made the change to make it so that since the remainder is at most one less than the number of processes in the world, I could just give each process (starting with the highest ranked process and iterating downwards) an additional row to manage. However, this implementation slowed down my code very slightly, on further thought even if it had sped up my code it would likely not have been a significant improvement since the imbalance in rows between ranks is at most one less than the number of ranks due to it being $Disparity = NumberofColumns(ModNumberofprocesses)$ This means that this disparity becomes less of an issue as the size of the image scales, and becomes more of an issue as the number of processes scales. These two factors together mean that this change would be a improvement if we were running relatively small images in 2019 on a double AMD Rome node, however with 16 core nodes on Bluecrystal3, scaling with the number of processes on a single node is not a useful change.

As for the change in strategy based on image size, I don't honestly quite understand why stencilling back and forth rather than copying the array is slower for small images since it seems to just be fewer operations. I imagine that there is some compiler optimisation that gets removed when the for loop is expanded to include two send/receive halo exchanges. However for images of around size 3000x3000, doing the double stencil starts becoming significantly quicker, at size 8000x8000 the stencil runs in nearly half the time when not copying the image array.

1.2 Carried over Serial Optimisations

Having tested both Intel's MPI compiler and the default gnu version, I'm still just using mpicc; it offers a speedup in general, some cases it did run slower but the idea of writing code to get the auto marker to go back and recompile based on the input arguments was slightly too impossible for me. I'm still using fast math optimisations as well, in most cases it offered a three or four times speedup. Once again, optimisations for doubles far outweighed any potential speedup from the lower data load of using floats. In fact, quite interestingly, it's seemingly equivalent to running the same code with one less process executing.

2 Timings

2.1 Optimised Serial Code Timings

Compiler	1024 image/s	4096 image/s	8000 image/s
gcc7.1.0 -Ofast	0.441	7.58	30.2
icc16 -Ofast	0.542	9.38	35.6

2.2 Timings with Array Copying

Stencil time/s 3sf (Mean of Three Runs)								
Processes	1	2	3	4	5	6	7	8
1024x1024	0.534	0.256	0.167	0.125	0.103	0.0852	0.0730	0.0689
4096x4096	10.6	5.56	3.77	2.88	2.69	2.77	2.37	2.14
8000x8000	35.5	17.3	13.1	9.86	9.89	10.4	8.93	7.82
Processes	9	10	11	12	13	14	15	16
1024x1024	0.0608	0.0527	0.0506	0.0475	0.0446	0.0418	0.0392	0.0373
4096x4096	2.29	2.06	2.25	2.18	2.25	2.14	2.25	2.11
8000x8000	8.59	7.73	8.45	7.86	8.45	7.84	8.38	7.90

2.3 Timings with ‘in-place‘ Double Stencil

Stencil time/s 3sf (Mean of Three Runs)								
Processes	1	2	3	4	5	6	7	8
1024x1024	1.09	0.543	0.366	0.276	0.227	0.189	0.163	0.144
4096x4096	14.3	7.43	4.73	3.83	2.94	2.54	2.17	2.02
8000x8000	31.8	20.0	13.0	12.1	10.9	7.18	7.10	6.25
Processes	9	10	11	12	13	14	15	16
1024x1024	0.133	0.121	0.111	0.104	0.104	0.0919	0.0879	0.0807
4096x4096	1.76	1.61	1.49	1.38	1.36	1.29	1.27	1.19
8000x8000	5.16	4.64	4.64	4.27	4.45	4.28	4.36	4.12

2.4 Timings with ‘in-place‘ Double Stencil and floats

Stencil time/s 3sf (Mean of Three Runs)								
Processes	1	2	3	4	5	6	7	8
8000x8000	58.5	30.0	20.2	15.2	12.5	10.5	8.98	7.91
Processes	9	10	11	12	13	14	15	16
8000x8000	7.65	6.48	5.92	5.50	5.20	4.87	4.53	4.26

2.5 Timing Analysis

These timings, especially the comparison between the two methods, show some interesting behaviour; clearly there is a significant overhead to the different strategies, looking at the timings for the 4096 image it's not until we start running six processes that we see any performance increase over the array copying approach. Also notably, the ‘in-place‘ method performs nearly half as well as array copying for the 1024 image.

Compared to the serial code, the speedup looks pretty good, for the small image and large image the MPI code nearly matches the fully serial code when running on one core, the medium image is ever problematic neither of my methods really get that close to the serial time.

Once we start using enough processes to stencil on a large enough image, we start seeing very significant performance increases with the ‘in-place‘ approach, I was surprised to see that more or less throughout it outperformed the other approach on the 8000x8000 image, I had expected the rate-limiting factor of the memory bandwidth usage to mean that the 16 core performance wouldn't be twice as fast as the array copying approach like for the 4096x4096 image.

3 Code/Timing Evaluation

It's no great surprise that at the high end of process numbers, the code is heavily memory bandwidth bound. As can be seen from the timings relative to the time for one process in that size category, while as number of processes increases the relative time for n processes $\approx \frac{1}{n}$ however when we get to using enough processes, this is no longer true. From my understanding this is due to - for the small image, or for small numbers of processes - being more comparatively compute bound, hence our times increasing proportionally with the number of processes which mainly provide more flops and less so memory bandwidth. This is most clear when looking at the times for the 8000x8000 image, while the first three or four process number increases offer us a significant performance boost when we start getting to using about eight or nine processes we have already hit near enough the time when running the code with 16 processes. I believe this specific eight or nine breakpoint is due to the specific architecture of Bluecrystal3, since Bluecrystal3 nodes contain two eight core sockets, once we start needing nine processes we are definitely using both sockets and hence at least have access to the maximum possible STREAM bandwidth. Since I am still a factor of two off the benchmarks for all the images it's clear that there is more I could have done to utilise more efficiently the memory bandwidth available.

3.1 Theoretical Maximums

In the case that we assume our message passing has no cost associated, i.e we reduce our code to cost 10 FMAs per pixel per iteration, we can obtain the following table for Bluecrystal3 theoretical maximum speeds on the stencil code, using the Blackboard roofline model.

Image Size	Giga FMAs required	1 core timing/s	16 core timing/s
1024x1034	2.1	0.05	0.0032
4096x4096	33	0.79	0.05
8000x8000	128	3.08	0.19

The first takeaway from the table is that message passing has a very real cost, the code is severely bandwidth bound, if we could manage to just ram the entire image array onto 16 cores uncaringly and suffer no errors or downsides we could reap a 10-20x speedup on my code.

The second takeaway which is somewhat unfortunate for me, is that the closest I get to the theoretical maximums is on the 8000x8000 image on a single core, where I'm *only* a factor of 10 off. This means that my code is scaling badly in terms of the cost of message passing.

3.2 Improvements

Consider my message passing(MP) scheme mathematically with processes p, and image dimensions x (width) and y (height).

The cost of passing messages is the cost of the left cyclic pass plus the cost of the right cyclic pass. Assuming the number of processes divides the width of the image, each process is managing $\frac{x}{p}$ columns and each row extends to the depth of the image.

$$\begin{aligned} MPCost &= MPLeft + MPRight \\ MPLeft &\propto py \\ MPRight &\propto pxy \\ MPCost &\propto 2py \end{aligned}$$

This means that the amount of message passing we have to do scales linearly with the number of processes and the size of the image.

If we instead consider a tiling approach we can say that,

$$MPCost = MPNorth + MPSouth + MPWest + MPEast$$

Assuming a square number of processes that divides the image dimensions, each process is managing a $\frac{x}{\sqrt{p}} \quad by \quad \frac{y}{\sqrt{p}}$ size tile.

$$\begin{aligned} MPNorth &\propto p\frac{x}{\sqrt{p}} \\ MPNorth &\propto \sqrt{p}x \\ MPSouth &\propto \sqrt{p}x \\ MPWest &\propto p\frac{y}{\sqrt{p}} \\ MPWest &\propto \sqrt{p}y \\ MPEast &\propto \sqrt{p}y \\ MPCost &\propto 2x\sqrt{p} + 2y\sqrt{p} \end{aligned}$$

Under the tiling scheme, with sufficiently sized tiles, the amount of message passing scales with the square root of the amount of processes we are using. More realistically, as the number of tiles increases on an image there comes a point where all we are doing is passing messages, but the number of processes can be selected for to counter this. Also performance may end up being better than this for the other reason of not every tile needing to pass messages in all four directions i.e the edge cases.

This means that a tiling scheme will achieve a far closer to maximal memory bandwidth, and I suspect is the difference between my code and the benchmarks.

Another implementation is to distribute rows to processes rather than columns, since if there are many more rows than columns each process will be very slow to complete its part and it may not be able to distribute work to all processes. This has been accomodated for (if a little bluntly) in my code, it checks to see which of dimensions x and y is the larger, then simply swaps them to enforce x being larger or equal to y.

My original approach was to use rank zero as a command process to farm out sections of the image to workers as was the remit of the Concurrent Computing xCore coursework from Year 2. This ran into issues. For one it is simply inefficient since you are using one less process and not gaining all that much, further the amount of message passing would be significantly higher leading to using memory bandwidth far less efficiently.