HPC MPI Report

Chanthru Uthaya-Shankar

Cu17780

Introduction

In this coursework I use the MPI standard and the OpenMP API to parallelise my stencil code to run onto all the cores of 2 nodes in BlueCrystal4 and run the stencil kernel faster.

Domain Decomposition

When deciding a strategy to parallelise my code I decided that the approach would be to create a strategy that incorporates halo exchange. The exchange would involve sending halo data between neighbouring ranks, I presumed that this method would be the most efficient form of data exchange as it would involve less sending and receiving compared to if all the communication was done via a master process.

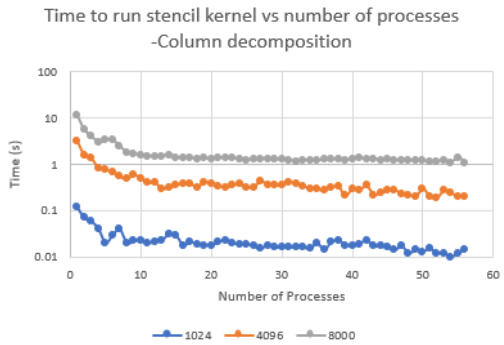
Column Decomposition

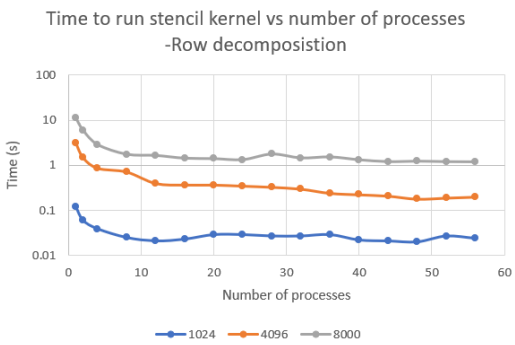
Initially I implemented the code using column decomposition, which I thought would be the quickest as the image was stored in row major form meaning that elements going down the row would be contiguous in memory. This meant I could send large sections of the array in one go without the need to store the data in a separate array buffer first. I could just specify the start address of the first element of the buffer as well as the length when sending halo data. My strategy for column decomposition was to divide up evenly the image space by however many processes there are available, with each process getting that amount of columns to work on, when the image size was not divisible by the amount of processes I added an extra column to each processes sub grid to account for the extra leftover columns.

Row Decomposition

The next step was to see which domain decomposition technique was better, so I changed my code so that it used row decomposition. Using row decomposition compared to column decomposition required the same amount of sends and receives however elements in array to be sent to the halo regions are no longer contiguous in memory so when filling the send buffers with data the process has to jump to different locations in memory to fetch the required data.

Comparison





Above shows the times achieved when running the stencil kernel on the different image sizes going from 1 process to 56 processes for both implementations of domain decomposition, column and row. As shown by the graphs using column decomposition was faster than using row decomposition. This difference in timings are mostly due to the reasons mentioned above, as in row decomposition the halo data to be packed into the buffer was not contiguous in memory, there was a less likely chance of that data being in cache when packing the buffer. This resulted in a higher cache miss rate for the row decomposition code. This meant that this data was in other storage mediums with higher access times. Overall this accounts for the increase in time between

For all images when using a low number of cores between 0 and 10 the runtime falls very quickly in a negative logarithmic fashion and then starts to plateau off as the number of cores keeps increasing. This shows that there's a limit to how many cores we can use and still get a speed up. A good application of this data would be to find the minimum number of cores to use and still achieve optimum runtimes.

Below shows the speedup from 1 core to 56 cores for each of the images.

Image	Speedup
1024	8.57
4096	15.9
8000	10.2

For each image the speedups were achieved at a lower amount of cores. The 1024 image achieved this speed up at 48 cores, the same with the 4096 image. The 8000 image achieves close to this speed up around 52 cores. This makes sense as the time for halo communication increases as the number of cores increases, eventually this time equals the time saved by splitting the grid to run on multiple cores and computing the stencil kernel on these sub grids. This causes the time to start to plateau. The 8000 image starts to plateau after more cores due to an increase in overall image area, but still never exactly reaches the same runtimes achieved on 56 cores. This tells us that possibly the runtime on 8000 could be increased further if more cores were available.

Message sending

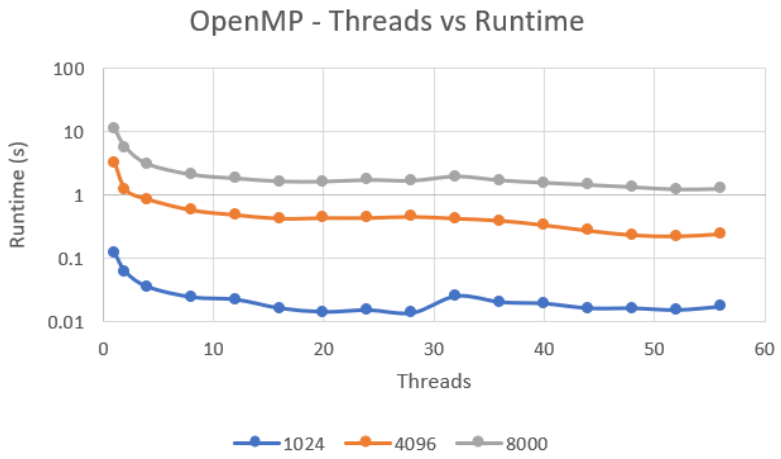
Once I had a fully working solution for both row and column decomposition, I tried to find other ways of speeding up my code. One possible bottleneck was possibly the strategy I was using to send and receive halo data. In my solution mentioned above I used `MPI_Sendrecv` to handle my halo exchange. I used this function twice in one `halo_exchange` function call, where on the first go ranks send their halo data to the left and receive from the right. The next use would be to do the opposite send to the right process and receive from the left process. Another solution that I implemented is to use the blocking API calls `MPI_Send` and `MPI_Recieve` to carry out halo exchange. I found that by doing this it gave me very similar runtimes when run on 56 cores:

Image size	Time (s)
1024 x 1024	0.017
4096 x 4096	0.21
8000 x 8000	1.3

As shown above the runtimes were very similar to the times achieved by using `MPI_Sendrecv`. This was due to overall the same amount of send and receives were occurring.

OpenMP

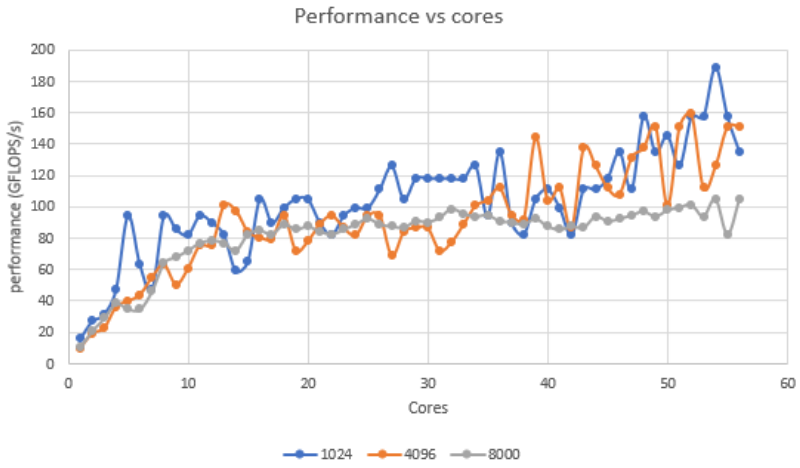
I then explored different parallelisation techniques, the alternate solution I came up with utilized the OpenMP API. This involved adding a pragma to parallelise the main loop in the stencil function.



What can be noted is that the times achieved changed very little by parallelising the code, but for some images such as the 1024 x 1024 image the fastest time was achieved using 28 cores at a runtime of 0.013 seconds, which was faster than when running on 56 cores which achieved a runtime of 0.017 seconds. A possible explanation for this is that there is an associated overhead when forking and joining threads when parallelising the code. Also, what can be noted is that after 28 cores for 1024 image the performance starts to slow down. This is because after a certain amount of processes the overhead of forking and joining starts to take up more time compared to the execution of the stencil code.

Performance

To compare performance, I calculated the amount of GFLOPS/s for each image using an operational intensity of 0.375.



As you can see by the graph for all cores the performance rises and steadily plateaus off, this is an example of sublinear plateau. This coincides with my times, where initially between small numbers of cores the times decreased a lot almost in a negatively exponential way and then plateaued off.

Conclusion

Below shows the final runtimes for each of my images using the column decomposition method and using `MPI_Sendrecv` for halo exchange.

Image	Run time (s)
1024	0.014
4096	0.21
8000	1.1