Introduction to High Performance Computing
Assignment 2: MPI

user id: ya17227 name: Yash Agarwal

Introduction

Previously for the serially-optimised stencil code implemented in assignment 1, was executed on a single core of the node. Only one instruction was executed at a time.

Here the stencil code is run from one up to all the 56 cores of 2 nodes. Here to implement parallel computing, the problem is broken down into discrete parts. Then instructions on each part is executed simultaneously on different cores. Three input grid vales (1024x1024, 4096x4096, 8000x8000) were used to derive the run times of the code and thus measure the efficiency of the algorithm.

Implementation:

Libraries and Concepts

The Single Program Multiple Data (SPMD) technique was used to achieve parallelism. Single Program meaning all processing units (cores) execute the same instruction at any given clock cycle. Multiple Data meaning each processing core can operate on a different data elements. The SIMD has synchronous and deterministic execution. This approach is best suited for specialised problems characterised by a high degree of regularity, such as graphics/ image processing. Since stencil is an image problem SIMD would be perfect here.

The Message Passing Interface (MPI) protocol was used to achieve point-to-point communication among processes. I used the Intel MPI with Intel C++ Compiler.

Distribution of workload

In serial optimisation the code was run on a single processor creating a bottleneck-like situation for computation and taking loads of time to run the code. Here multiple cores are used to exponentially boost the run times. Now to use multiple cores at once we need to distribute different sets of data to be worked on to the respective cores.
The scatter approach was used to distribute and gather approach to collect the equal sets of data to and from the cores as shown in fig 1.

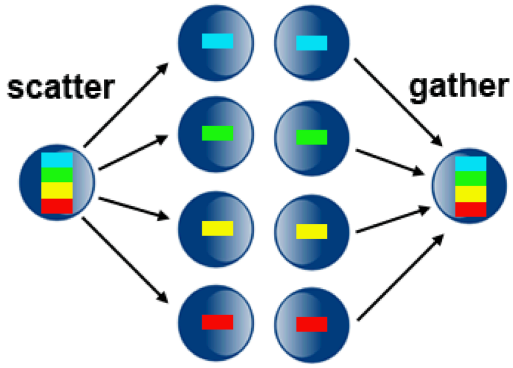

Fig 1: distribution and collection of the stencil grid

The first or the rank 0 core splits the original grid as equally as possible depending on availability of the no of cores to avoid load imbalance. Then assigns the sub-grids (have row-major layout as shown in fig 2.) to the cores.

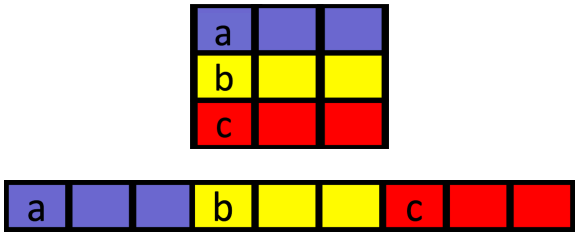

Fig 2: logical layout (above) & physical layout (below)

The data assigned to each core acts as an individual grid itself for every core or as a sub-grid. Then the computation is done on each sub-grid by the respective cores. After the computation the processed sub-grids are collected by the rank 0 core and assembled into a single grid as shown in fig 1. The assemble grid is then sent as output as if it were processed by a single processor.

Halo Exchange

This stencil code implements a weighted 5-point stencil on a square grid. The value in each cell of the grid is updated based on an average of the values in the neighbouring elements. Hence each cell element requires its neighbouring data to compute.

In fig 3, all the cells shown in purple have access to all their neighbouring values in the sub-grids and can be easily computed. Their adjacent neighbours are mapped into the same core. But for the cells in yellow the adjacent neighbouring values are in different cores: one core above and one core below.

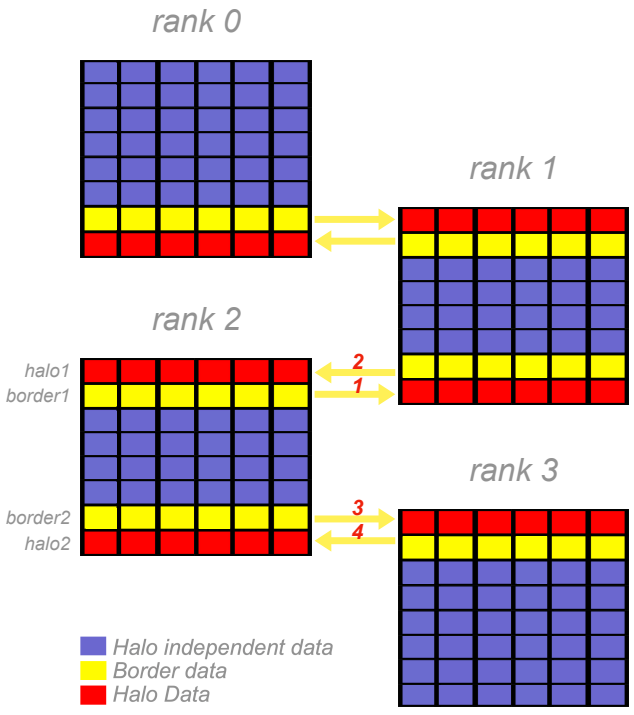

Figure 3: Halo exchange for 4 processor cores

To solve this issue an extra row of elements was added on either side of each local grid. These are shown in red in fig 3, and are called Halo or ghost cells. The halo cells store a copy of the border cells of the adjacent cores. The halo exchange is a 4 step process as shown for rank 2 core in fig 3.

1. *Border1* data is sent to the *halo2* cells of the previous rank1 core.
2. *Halo1* cells receive the *border2* data from the previous rank1 core.
3. *Border2* data is sent to the *halo1* cells of the next rank3 core.
4. *Halo2* cells receive the *border1* data from the next rank3 core.

This halo algorithm is updated every iteration to get the latest neighbour data every time before computation. With the introduction of halo cells the sub-grids behave as if they were one big continuous grid and at the same time racking up much faster computation speeds.

MPI techniques:

To share halo data values between the cores the MPI techniques were used. MPI provides multiple modes for sending messages: blocking techniques like *MPI_Send*, *MPI_Ssend*, *MPI_Rsend*, *MPI_Bsend*, *MPI_Sendrecv*, etc and non-blocking techniques like *MPI_Isend*.

Blocking

- *MPI_Send* : does not post send request until the system buffer is full and the command is not completed until the buffer is empty. Sometimes the send request could be posted before the receive request was posted by *MPI_Recv* which holds the data in the system buffer occupying extra memory and blocking other requests. This is the most basic mode and one of the slowest.

- *MPI_Ssend* : is a more synchronous mode. The send does not complete until a matching receive has begun. It is slightly faster than *MPI_Send*. Like

MPI_Send, using this mode can deadlock unsafe programs.

- *MPI_Bsend* : In this mode the user has to supply a buffer to the system for use. This is less likely to deadlock since the user can allocate enough memory to make an unsafe program safe.

- *MPI_Rsend* : In this mode for every send request the user has to guarantee that a matching receive request has been posted. This can result in undefined behaviour if the matching request is never posted.

- *MPI_Recv* : receives the message sent in any of the above modes.

- *MPI_Sendrecv* : This supplies simultaneous send and receive. It supplies receive buffer at the same time as send buffer. This does not hold the data unnecessarily in the system buffer and saves a lot of time and memory. Upon testing this was found to be the fastest blocking technique for the program as shown in fig 4.

Grid size	MPI_Send runtime(s)	MPI_Ssend runtime(s)	MPI_Sendrecv runtime(s)
1024 x 1024	0.06	0.03	0.01
4096 x 4096	0.25	0.19	0.13
8000 x 8000	1.10	1.36	0.94

Fig 4: Comparing the runtimes of the program (on 56 cores) using different MPI blocking techniques.

Non-Blocking

- *MPI_Isend* : This is a non-blocking technique, i.e. the requests are posted simultaneously oblivious of all the other requests. These non-blocking operations return “request handles” which can be tested and waited on to check whether the command was successfully executed. These are asynchronous and safe from deadlocks.

Grid size	MPI_Sendrecv runtime(s)	MPI_Isend runtime(s)
1024 x 1024	0.01	0.03
4096 x 4096	0.13	0.28
8000 x 8000	0.94	1.87

Fig 5: Comparing the runtime of the program (on 56 cores) for blocking vs non-blocking technique.

In theory *MPI_Isend* should be faster than the *MPI_Sendrecv* but upon testing it was found to be slower. *MPI_Sendrecv* was found to be providing with the fastest runtimes on all 56 cores for all the 3 input grids as shown in fig 5.

MPI_Isend could be running slower because the background communications for *MPI_Isend* will be performed on an additional thread on BCp4 since we are already using all the available hardware threads for main computation. This can cause usage of resources in turn increasing the time taken to run.

Since the stencil program has to run synchronously every iteration maybe the *MPI_Isend* is not very useful in this case and may not be running purely asynchronously here.

The MPI_Sendrecv was embraced being the fastest for our code.

Roofline Model

The roofline model analysis was made for the program running on 56 for all the three input sizes as shown in the fig 7. First the Operational Intensity (OI) was calculated as following. OI is the same for all three inputs. Then the achieved Memory bandwidth foe each input. From there we find the Peak floating point Performance which is the product of bandwidth times OI.

Operational Intensity (OI):

$$= \frac{9}{(5r + 1w) \times 4_{byte/FLOPS}}$$

$$= 0.375_{byte/FLOPS}$$

Achieved Memory bandwidth:

$$= \frac{2 \cdot nx \cdot nx \cdot 2 \cdot niters \cdot 4}{runtime}$$

Peak performance:

$$= Memory\ Bandwidth \times OI$$

Grid size	Sendrecv, 56 cores runtime(s)	Memory Bandwidth (GB/s)	Peak Performance (GFLOP/s)
1024 x 1024	0.01	167.7	62.8
4096 x 4096	0.13	12.9	4.83
8000 x 8000	0.94	1.78	0.66

Fig 6: Roofline Calculations

As shown in the above fig 7, we can observe that the program is memory bandwidth bound for all three input variables.

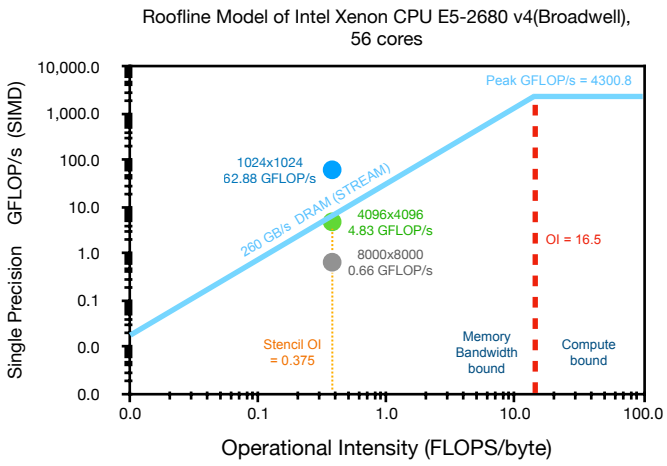

Fig 7: Roofline model using 56 cores.

Conclusion

The fig 8 plots all the runtimes of the program running on 1 core upto 56 cores. From this figure we can conclude that no. of cores used is inversely promotional to the runtime, and it is directly proportional to the speed of the program. More the cores used more the speed.

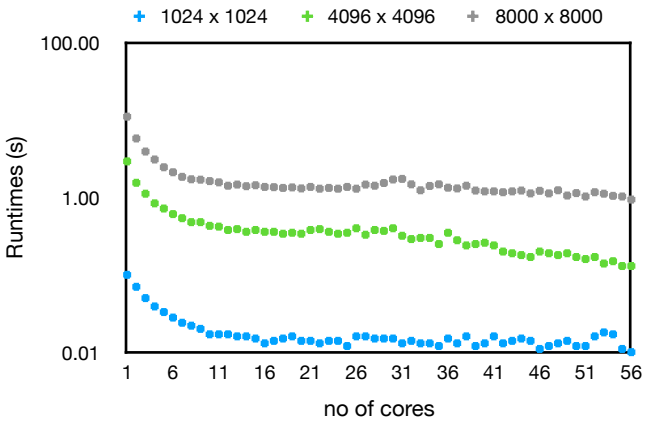

Fig 8: graph plotting speedup of program vs no of cores used.

The program was significantly speeded up by the deployment of parallel computing techniques as evident in fig 9. The code passes the *check.py* test for all the cores used.

Grid size	Runtime 1 core (in sec)	Runtime 56 cores (in sec)	Speedup (x times)
1024 x 1024	0.10	0.01	10.0x
4096 x 4096	2.93	0.13	22.5x
8000 x 8000	11.10	0.88	12.6x

Fig 9: Runtime serial optimisation (1 core) vs parallelisation (56 cores)

References

- http://wgropp.cs.illinois.edu/courses/cs598-s15/lectures/lecture23.pdf
- https://computing.llnl.gov/tutorials/parallel_comp/
- http://www.mathcs.emory.edu/~cheung/Courses/355/Syllabus/92-MPI/async.html
- https://www.rookiehpc.com/mpi/docs/mpi_isend.php