Intro to HPC Assignment 2 Report

sw16437, Jerry Wang

Row Decomposition

The program was first implemented with row decomposition because images are stored in row-major format. This means all partitions will be in contiguous memory space, therefore enabling shorter access time. Images are sliced horizontally by the number of cores to reduce the overhead caused by the lack of parallelism or load imbalance. If the image cannot be evenly divided, the remainder rows (smaller than quotient) will be allocated to the last core. This ensures the remainder core has smaller data chunk than the last core, so that the overall performance won't be limited by the remainder. Halo exchange is completed by a blocking send and receive operation MPI_Sendrecv. This configuration plateaued and reached ballpark time with all 56 cores at 0.02s, 0.14s, 1.03s for image size 1024*1024, 4096*4096 and 8000*8000 respectively as shown in figure 1.

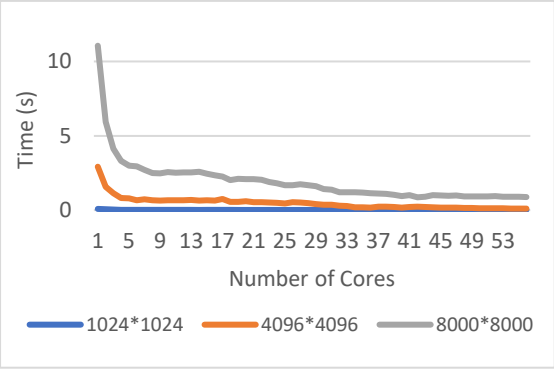

Figure 1 - Runtime of Row Decomposition using configurations of 1 to 56 cores

Column Decomposition

Using column decomposition, the stencil operation performs much slower than row composition because all halo exchanges will access non-contiguous memory. As illustrated in figure 2, the column decomposition approach performed slightly slower, about 5% slower on average than row decomposition. (Timing of row column implementation was not taken on the same day, the actual difference may be larger if the performance fluctuation of Blue Crystal is taken into account)

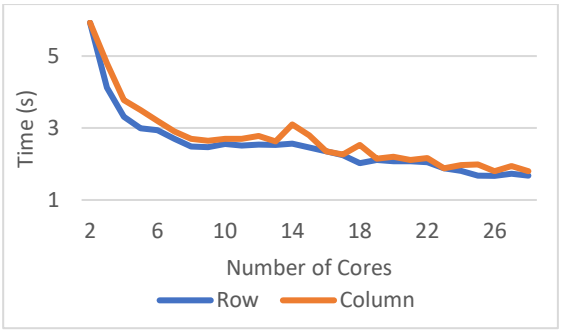

Figure 2 - Runtime of Row Composition and Column Decomposition

Tile Decomposition

A tiling approach is implemented using a 2-D MPI cartesian grid topology. The grid is set up with MPI_Cart_create and each tile finds its neighbour through MPI_Cart_shift. The grid is then mapped onto the image using the same distribution method used in the row decomposition implementation. For communication among tiles, MPI_Neighbor_alltoallv is used because this allows halo exchanges with one single datatype but multiple displacements. This is a blocking call that lets each tile to send and receive their halos with 4 neighbours. Surprisingly, tile decomposition was only on average 1.2x faster than row decomposition for size 4096*4096 and 8000*8000, and no obvious improvement for size 1024*1024 if noise is taken into consideration. The runtime for this implementation using 56 nodes are 0.01s, 0.10s, 0.88s respectively.

It is observed that tile decomposition outperforms row decomposition especially when the column numbers in the grid is lower. It is consistently faster than row decomposition when given a prime number core count (which leads to a grid width of 1) but struggles to match row decomposition with wider grids such as a 7*8 grid for a 56-core configuration. Although executing MPI_Neighbor_alltoallv in a 2D cartesian grid produces smaller buffer sizes, it will need to access both contiguous and non-contiguous memory for horizontal and vertical halos. Furthermore, the exchange of vertical halos in row-major images will degrade performance significantly due to the access of non-contiguous memory. Intel Trace Analyzer extracted the collective runtime of each function call. It showed that in a prime

number configuration of 17 cores, MPI_Neighbor_alltoallv and MPI_Sendrecv produced similar runtime, whereas a 16-core configuration witnessed 5% faster runtime in favour of MPI_Sendrecv.

Further experiment with non-square input to compare the advantages of both row decomposition and tile decomposition shows for extremely narrow images (e.g. 10*100000), row decomposition outperforms tile decomposition consistently due to tile implementation's large-scale access of non-contiguous data. However, tile decomposition becomes the winner when given extremely wide images (e.g. 100000*10) due to its ability to maintain smaller send and receive buffers.

This result clearly demonstrates the different advantages of different topologies when processing arrays with different aspect ratios.

Atomic vs Collective Communication

The influence of messages size on runtime was investigated by implementing a variation of my row decomposition implementation. Instead of sending chunks of filled buffers in MPI_Sendrecv, pairs of MPI_Send and MPI_Recv are used instead to send and receive one element in the array at a time. Unsurprisingly, this caused a massive slowdown. In the case of larger images, this implementation was on average 8x slower than the original. As the message size started to increase, the performance starts to recover. This makes sense since MPI_Send and MPI_Recv are blocking calls. And the waits for blocking point-to-point communication can result in significant synchronisation overhead, which can also amplify the "random noise" effects caused by OS interrupt. Packing buffers collectively instead of one-by-one however, did not have a noticeable impact on the runtime.

MPI Type Vector

As an attempt to address the disadvantage of accessing non-contiguous memory of the above tile decomposition implementation, alternative implementation was done using Neighbor_alltoallw as the communication method. Comparing to its "v variant", this call allows each tile to send and receive multiple datatypes with multiple displacements. In a row-major image, a horizontal halo can be defined using MPI_Type_contiguous, and a vertical halo can be defined using MPI_Type_vector. These two constructors sum up the characteristics of the memory location of the elements in the halos into collective type vectors, while using the pointer of the start of the tile as the anchor of its send and receive buffer. However, testing showed that this did not produce any noticeable improvement over my original implementation. In order to test if MPI_Type_vector was useful in reducing non-contiguous memory access. As an experiment, the worst-performing column decomposition implementation changed to use send and receive MPI_Type_vector instead of packed buffers. The result showed very limited improvement over the original, which may come from the implicit buffer filling by MPI_Neighbor_alltoallw. This indicates that MPI_Type_vector does not reduce the access of non-contiguous memory. Although the combination of Neighbor_alltoallw and MPI_Type_vector eliminates the need for send and receive buffers by avoiding one extra copy of the halos as well as handling halo exchange in one line, there will always be a cost of accessing non-contiguous memory.

Blocking vs Non-Blocking

Another experiment done was comparing blocking and non-blocking communications. Row decomposition was modified to use MPI_Isend and MPI_Irecv. Tile decomposition was modified into two variants using MPI_Neighbor_alltoallv and MPI_Neighbor_alltoallw respectively. These non-blocking calls return immediately and do not wait for the communication to finish. Timings of the new variants showed no sign of faster performance. Intel Trace Analyzer shows that 6.37% of the program runtime was spent waiting for barrier. This makes sense because in order to make sure the correctness of the output, the stencil process needs to be synced so that each tile is running the same iteration. In addition, MPI_Barrier, which are very expensive to

use, had to be put before and after MPI_Neighbor_alltoall[v/w] to ensure the buffers are filled and valid before being received into each tile or slice.

OpenMP

OpenMP supports shared memory multiprocessing programming while MPI programs run on distributed memory across cores. This means the parallel thread can have access to the same data. Upon an omp parallel pragma, the program forks into separate threads and join back together at the end of the parallel region. Comparison between the OpenMP implementation and MPI row decomposition implementation using configurations from 2 cores to 28 cores shows very little difference in the peak performance of the two implementations. However, the runtimes of the OpenMP stencil do present an interesting feature.

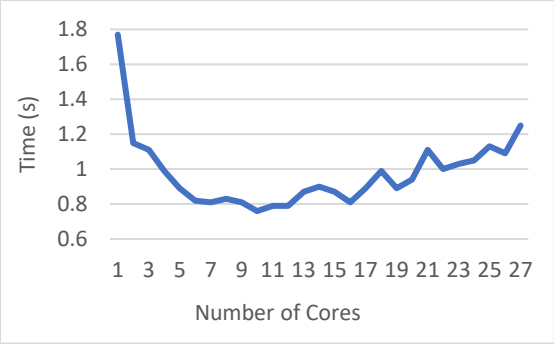

*Figure 3 - Runtime of OpenMP implementation for image size 4096*4096*

Figure 3 shows increasing thread number using OpenMP to perform stencil sees a convex trend in timing. Its performance first improves, then deteriorates. This makes sense because the bandwidth utilization of the OpenMP stencil scales positively up to a number of threads. However, as the total utilization bandwidth reaches the physical memory bandwidth limit, the performance will plateau. Adding more threads further will likely cause a competition in memory bandwidth and sees a drop in performance, as is shown in figure 3.

Upon closer examination, it can be seen that the MPI timing curves possess the same characteristic caused by the limitation of physical bandwidth. The performance degrades as core counts exceeds a certain number. This feature is more easily observed on bigger image including 4096*4096 and 8000*8000.

MPI+OpenMP

As a comparison, an implementation that combines MPI and OpenMP by only parallelising the stencil loops with pragma did not provide any performance gain over the fastest MPI implementation as shown in Figure 4.

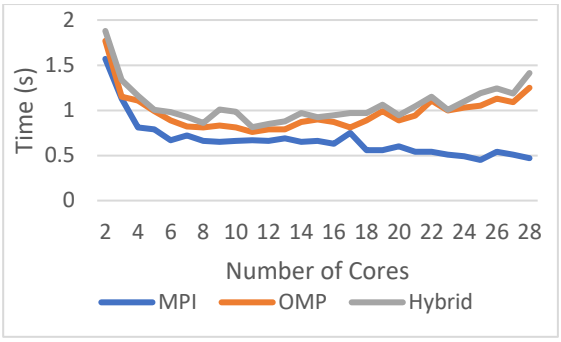

Figure 4 - Runtime comparison of MPI, OpenMP and Hybrid

Thread Pinning

Another technique available was thread pinning. Thread pinning provides a way to pin a group of processes to a specific node and avoid undesired process migration. The MPI implementations require extensive amount of communication among the cores for halo exchanges. When a thread communicates with its neighbours on another node, it creates overheads. To try and minimise the performance lost due to this reason, thread pinning could bring potential improvement. Using I_MPI_DEBUG 4 enables the display of the node assignment for each rank. For example, for a configuration of –nodes 2, --ntasks-per-node 8. Rank 0-7 are located on compute node A, while rank 8-15 are located on compute node B. After determining each rank's neighbour. MPI_PROCESS_PIN_LIST is then used to group the adjacent ranks on the same node. Since Blue Crystal has a fixed rank node distribution policy, pinning threads should allow some improvement in performance. By pinning rank 0,1,4,5,8,9,12,13 on node A and rank 2,3,6,7,10,11,14,15 on node B, It was able to

achieve a minor performance boost of 1.1x. Although not significant, this demonstrated the importance of cache affinity and the impact of accessing data on a separate node could be significant on a much larger input.

During this experiment, configuration utilising cores on both nodes perform much faster than on a single node. For example, comparing a configuration of –nodes 2, --ntasks-per-node 8, with –nodes 1, --ntasks-per-node 16. The prior produces a runtime 1.8x faster than the latter. This is understandable since doubling the node count will double the available memory bandwidth as well, which in theory allows the program to run twice as fast.

NUMA Effect

Blue Crystal uses Non-uniform Memory Access architecture. NUMA allocates each CPU its local memory that is fast to access. That architecture avoids the performance issue that can be caused if multiple CPUs are accessing the same memory module at the same time. When a CPU is accessing non-local memory modules, an interconnection is established using the owning NUMA node. This interconnect usage slows down the memory access. Another policy of NUMA is first-touch memory allocation. In OpenMP stencil, because image initialisation is executed in serial, memory will be allocated on the master thread. This will result in the forked threads accessing non-local memory, causing slowdown. By splitting up the initialisation onto each forked thread, the improved OpenMP implementation was able to match the previous MPI implementation.

Size	1024*1024	4096*4096	8000*8000
Before	0.02	1.14	5.35
After	0.02	0.67	2.55

Table 1 - Comparison before and after parallel initialisation using configuration of 28 cores.

Roofline Analysis

Figure 5 shows a roofline graph of BC4 using 28 cores on size 1024*1024. Table 2 shows the performance of the program using 28 cores. The peak performance is bounded by L1.

Size	1024*1024	4096*4096	8000*8000
MPI	419	697	341
OpenMP	251	76	76
Hybrid	204	62	73

*Table 2- GFLOP/s Performance on image size 1024*1024*

Conclusion

Runtimes achieved by the submitted program using all a 56-core configuration are listed in table 3. Comparing the results from the serial stencil code, the parallel solutions clearly achieved a much faster runtime.

Size	1024*1024	4096*4096	8000*8000
MPI	0.01 ± 0.003	0.12 ± 0.02	0.95 ± 0.08
Serial	0.08	2.89	11.04

Table 3 - Runtime achieved by MPI (56 cores) and serial code

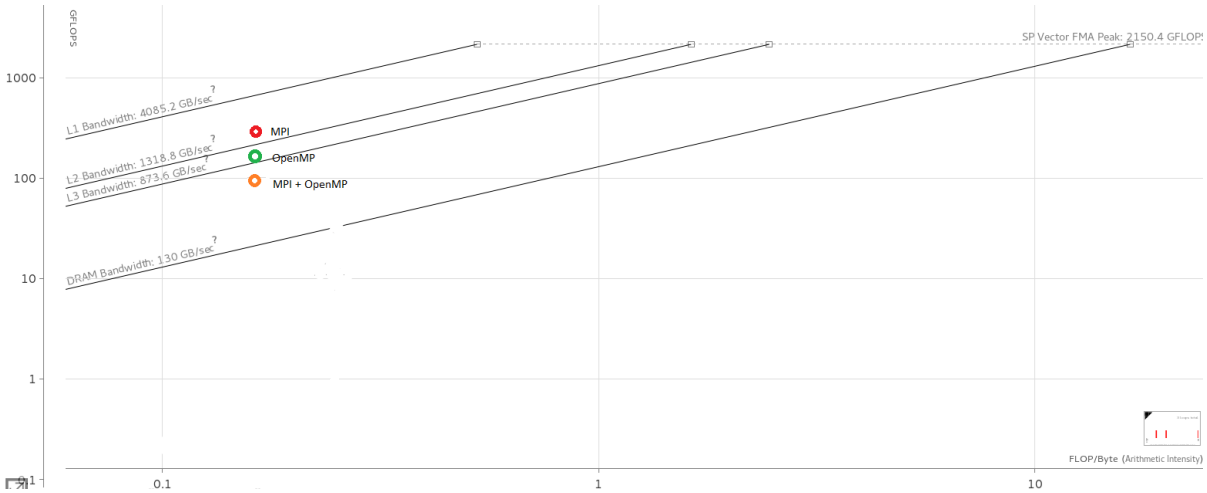

Figure 5 - Roofline model of Blue Crystal 4, 28 cores