Distributed Memory Parallelism with MPI

Samia Mohinta
Department of Computer Science
University of Bristol
User ID : dc18393
Candidate No. : 97293

***Abstract*—The objective of this assignment was to parallelize the serially optimised 5-point Stencil algorithm using MPI `Single Program Multiple Data' technique and observe how the application scales, in terms of reduction in run-time across 1 to 16 cores of one BlueCrystal Phase 3 node. This report will detail the approaches undertaken to implement parallelism, the effects on the run-time as the code ran on multiple cores as well as discuss the observations.**

Keywords*— *High Performance Computing, MPI, Single Program Multiple Data (SPMD), Stencil, Parallelism.

I. SERIAL OPTIMISATIONS

As part of the first assignment, we applied different optimisations such as choosing the right compiler flags and code modifications for single precision floating point calculations and contiguous memory access to enable vectorisation and thereby improve performance in terms of speed on a single core of BCp3.

Intel Compiler (`intel-compiler-16-u2`) with *-O2* and *-xHOST* optimisation flags was used to compile the code. The optimised run-times achieved for 1024*1024, 4096*4096 and 8000*8000 input image sizes were as in *Table 1*.

Input Image Size	1024*1024	4096*4096	8000*8000
Run-Times	0.08s	2.47s	8.51s

Table 1: *Run-Times Achieved after Serial Optimisations using Intel Compiler (Version 16, Update 2).*

II. PARALLELISM USING MPI : METHODOLOGY

A. Problem Understanding and Defining the Methodology

Our problem involves a grid a cells (input image matrix) and a time-stepping simulation (100 iterations of `stencil`) that we aim to split across multiple cores for faster computation. To achieve this, we require a domain decomposition into smaller grids. These small data grids are owned by different ranks in the cohort, which now run `stencil` on their local data subset. This technique helps to reach the solution quicker in terms of wall-clock time, because now each rank in itself has less work load as it works only on a subset of the input matrix simultaneously along with other initialised ranks in the cohort.

However, we have a specific type of memory access pattern (*stencil*), wherein to calculate the values for each cell, the program requires to look at the values of its neighbouring East, West, North and South cells. As a result, some cells need to access data held on a different rank. In the distributed memory architecture, no rank has access to data held on other ranks. To solve this issue, addition of halos around the sub-domains on each rank needs to be performed. Halo-exchange between ranks is accomplished by a point-to-point communication pattern between ranks using a set of MPI send and receive functions.

Also, since it is a time-stepping problem, we have to repeat this halo-exchange between the ranks after each iteration, so that each rank has updated cell values in the halos to use for further calculation.

With the above understanding, the serially optimised code was modified to incorporate different forms of domain decomposition (row-based, column-based) with message passing (halo-exchange) between ranks in the initialised cohort to develop a parallel program that scaled across 16 cores of a node. Additionally, as I had achieved fastest run-times in the serial code optimisation assignment with Intel compiler (`intel-compiler-16-u2`), I chose to use it for this assignment as well. Intel compilers take advantage of the underlying Intel architecture of BlueCrystal. Moreover, this version of the Intel compiler also prevented me from loading any extra `open-mpi` modules, because it came with inbuilt `mpi` libraries.

B. Domain Decomposition by Columns

1. Approach

The input image grid was partitioned into sub-grids such that each rank got all the rows, but a subset of the number of columns (Figure 1). The decomposition method divided the input grid into equal number of columns for each rank and assigned any remainder columns to the last rank in the cohort. Two extra columns were added to the local sub-domains of each rank to hold the halos it received from its neigbouring ranks. `MPI_Sendrecv` function was used to establish a halo-exchange pattern between the ranks, wherein each rank first sent data to the rank on its left while receiving data from the rank on its right and then sent data to the right while receiving from its left. `MPI_Sendrecv` was used so that I need not bother about ordering the blocking sends and receives, while the communication subsystem took care of any cyclic dependencies that might lead to deadlock [2]. However, it should be noted that `MPI_Sendrecv` still relied on a send from a rank to complete a receive operation and if that did not happen, the system deadlocked. Thus, in this case, assigning ‘left’ and ‘right’ to incorrect ranks caused deadlocks.

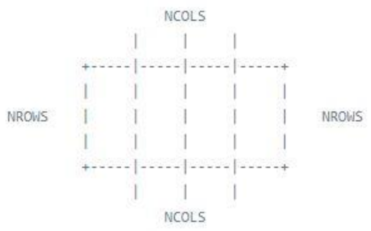

Figure 1*: Column-based Domain Decomposition.*

Halo-exchange was done after every iteration to send updated cell values to the neighbouring ranks for further processing. Code was compiled as `mpiicc -O3 -Wall -std=c99 stencil.c -o stencil` command.

2. *Results*

With the above approach, the run-times dropped to `0.03s`, `1.09s` and `3.9s` for 1024, 4096 and 8000 input image sizes when the program ran on 16 cores. Even though the program execution became approximately 3x faster for the smaller input grid (1024), it did not scale very well for the larger input sizes (4096 and 8000), which ran just twice as fast as on a single core.

Such results for large input sizes may be attributed to the fact that this memory layout might not be efficient with respect to C's arrangement of being a row major language. A series of memory jumps to access each cell will be involved in this data arrangement and therefore degrade the performance. Moreover, since the halos are columns, with larger input sizes there is a possibility that the halos don't entirely fit into the cache and as halo-exchange is carried out after every time-step, column halo reconstruction in memory can turn-out to be expensive.

C. Domain Decomposition by Rows

1. *Approach*

Since the ballpark run-times were not reached after running *`stencil`* on the distributed column chunks on 16 cores, the code was modified to decompose the input image matrix into chunks of rows instead of columns. This was done because accessing memory sequentially in the order it is stored is more efficient than iterating through a row-major ordered data structure in a column-major order.

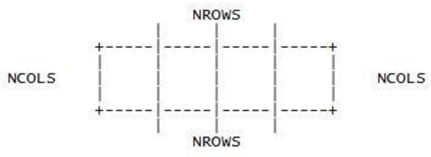

Figure 2: *Domain Decomposition into Row Sub-grids.*

The code structure remained similar to the column-based approach with a few modifications. Since `stencil` application on the first and the last row of the input grid does not require any halo, therefore the chunks containing the first and last rows were assigned a halo width of 1 row. All other chunks held by ranks between `0` and `Size-1`, were assigned a halo width of 2 rows. Blocking `MPI_Send` and `MPI_Recv` functions were ordered to establish a point-to-point message passing communication pattern, wherein each rank exchanged halos with a rank above or below it.

2. *Results*

The run-times recorded after running this parallel code on 16 cores of one node in BCp3 are in *Table 2*.

Input Size	1024*1024	4096*4096	8000*8000
Run-times	*0.01s*	*0.54s*	*2.03s*

Table 2*: Optimised Run-times Achieved with Domain Decomposition by Rows.*

This implementation of the parallel code recorded an approximate speedup of 8x (0.08/0.01) for 1024*1024 input size and a speedup of 4x (2.47/0.54, 8.51/2.03) for both 4096 and 8000 inputs when compared with the run-times attained with serial optimisations. This performance with respect to speed boost can be attributed to efficient sequential memory access pattern during `stencil` computations.

III. Observations and Discussion

A. Scaling

A graphical representation of the code scaling from 1 to 16 cores, in increments of one core, is shown in **Figures 3, 4 and 5**. It can be observed from the Figures that the maximum speed up is recorded as soon as the program is launched to run on 2 cores. The run-times were nearly reduced by half, from *0.08s to 0.05s* (**speedup=1.6**) for 1024*1024, *2.47s to 1.37s* (**speedup=1.8**) for 4096*4096 and *8.5s to 4.3s* (**speedup=1.9**) for 8000*8000 image sizes. Thereafter, with subsequent additions of cores (one at a time), the performance improved sub-linearly and plateaued after the addition of the 10th core to the system for all three input sizes.

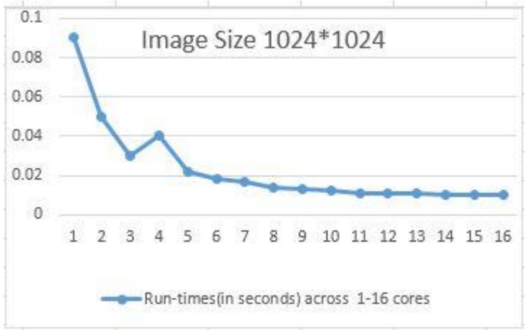

Figure 3: *Run-times recorded on Cores 1 to 16 for Image Size 1024*1024. Run-time on 1 Core= 0.09s, Run-time on 16 Cores= 0.01s.*

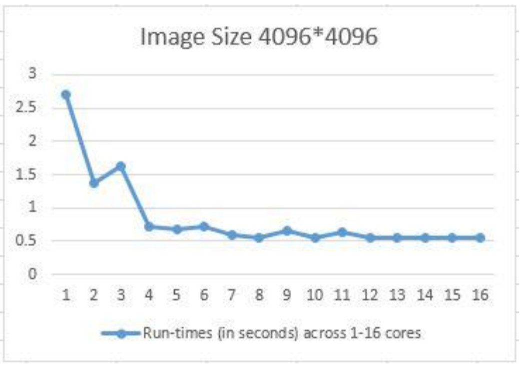

Figure 4: *Run-times recorded on Cores 1 to 16 for Image Size 4096*4096. Run-time on 1 Core= 2.7s, Run-time on 16 Cores= 0.5s.*

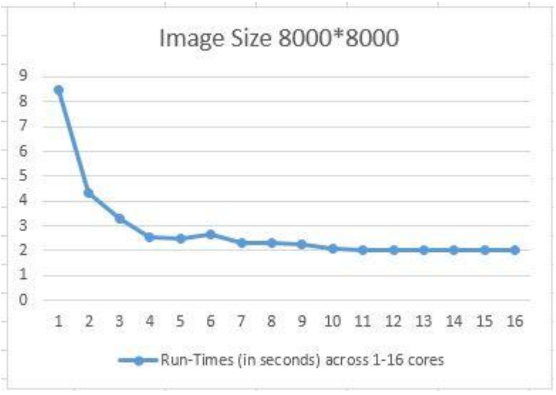

Figure 5: *Run-times recorded on Cores 1 to 16 for Image Size 8000*8000. Run-time on 1 Core= 8.5s, Run-time on 16 Cores= 2.1s.*

No further improvement in speed even after adding more cores (11th to 16th core) was because now the speed of communication is slower than the speed of computation. With each additional core added, the amount of computations to be performed by each rank is reduced because the size of the local sub-grid becomes smaller, but every rank still has to exchange large packets of row halos with its neighbouring ranks. The time taken to transfer and copy the data as halos after each iteration between ranks now becomes the determining factor for the run-time. As a result, even when the computations happen a lot faster, the ranks have to wait to receive updated halos to continue processing and this affects the speed, which shows no improvement and plateaus after a certain point. In other words, the communication time adds up and stalls the performance growth.

Also, with addition of more cores, we are ideally not increasing the memory bandwidth. *Stencil* being a memory bandwidth bound problem does not achieve optimum scaling performance when more compute resources are added. Moreover, the cores may start sharing the memory bandwidth as we add more of those to the cohort. In order to attain optimum scaling for *Stencil*, adding more nodes will scale better than adding more cores in the same node, because with more nodes we add more memory bandwidth.

B. *Process Pinning*

On taking a closer look at the scaling graphs above for the stencil algorithm running on 1 to 16 cores, a few spikes in the run-times can be noticed upon addition of more cores. In other words, after launching the program to run on 4 cores for input size 1024, the run-time increased from what was recorded when the program ran on 3 cores. Nevertheless, the run-time again decreased after addition of one more core. This abrupt increase in the run-time (run-times should drop when program runs on more cores) is believed to be dependent on how far the cores are located inside the node. That is, since there are two sockets per node in BCp3 and 8 cores per socket, it is possible that resource manager in BCp3 allots two cores per socket. As a result, not all the cores are connected to a single socket and therefore need to communicate with the other two cores connected via the second socket. This communication via sockets degrades the speed because it takes more time to transfer data to cores located farther and the cores sit idle waiting for data.

The problem was overcome by pinning processes to run on designated cores that are located within a close boundary [4], possibly allocating all cores connected to one socket, when the program was launched on <=8 cores . Option `–binding order: compact` was used with `mpirun` to direct the resource manager to assign adjacent cores that are closest to one another in the multi-core topology [3]. Also, all 16 cores of the node were reserved even when the program was launched to run on less than 16 cores to prevent sharing of cores in one node with other users.

The scaling graphs for 1024*1024, 4096*4096 and 8000*8000 after invoking `mpirun` with `– binding order:compact` option are given in **Figures 6, 7 and 8**.

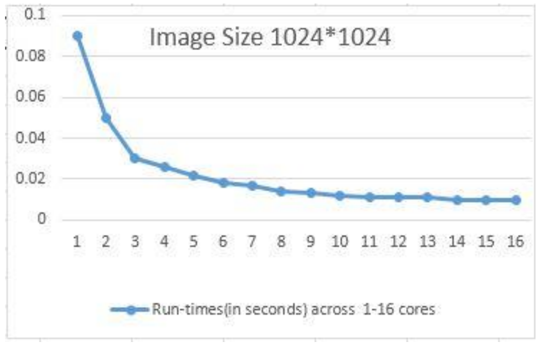

Figure 6: *The performance now scales evenly in a sub-linear manner without abrupt speed decrease on increasing the number of cores for image size 1024*1024.*

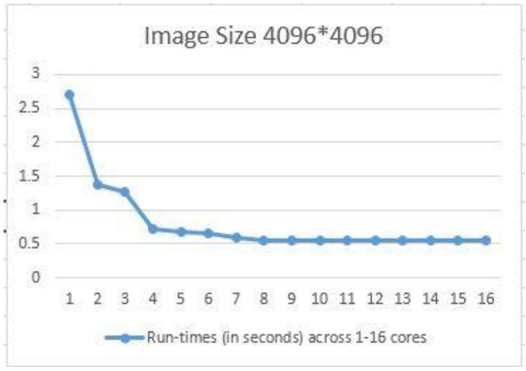

Figure 7: *The performance now scales evenly in a sub-linear manner without abrupt speed decrease on increasing the number of cores for image size 4096*4096.*

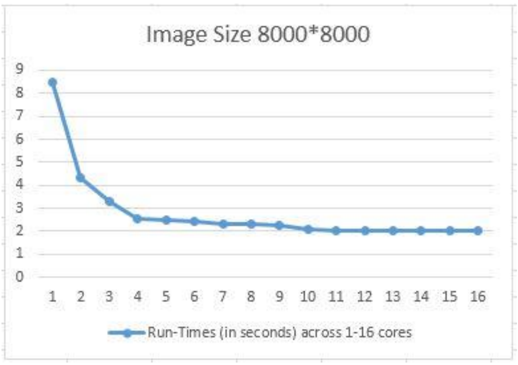

Figure 8: *The performance now scales evenly in a sub-linear manner without abrupt speed decrease on increasing the number of cores for image size 8000*8000.*

C. Roofline Model

After calculating the floating point performance (measured in GFLOPS/s) for the three different input image sizes, it is observed that 1024*1024 image size is bound by the L2 cache, while the image sizes of 4096 and 8000 are bound by the L3 cache. These observations have been made by plotting the GFLOPS/s values to the roofline graph (**Figure 9**). The calculated floating point performance for this parallel version is approximately 118 GFLOPS/s, 38 GFLOPS/s and 36 GFLOPS/s for 1024, 4096 and 8000 image sizes respectively.

There has been a significant improvement with this parallel implementation when compared to the floating point performances of the serially optimized code, which were 21.97 GFLOPS/s, 11.4 GFLOPS/s and 12.6 GFLOPS/s for inputs 1024, 4096 and 8000 respectively. In spite of the improvements, the problem still remains memory bandwidth bound because the image matrices are still unable to fit completely into the higher level caches. Therefore, additional computing power in the form of more cores is not being optimally utilized, as speed of computation is limited by memory bandwidth.

The operational intensity was calculated by counting the number of floating operations divided by the total amount of bytes loaded or stored in a single loop iteration. Based on my `stencil` block, I know that there will be 6 floating point operations in each iteration, because I have changed the code to implement 2 multiply and 4 add operations. Also, there are 5 reads and 1 write operation every iteration, each of 4bytes. Hence, the OI is 6/24 =0.25. When SIMD GFLOPS/s values are plotted against this OI, the points lie on the left of the ridge of the roofline model, thereby showing that the problem still lies on the memory bandwidth bound zone.

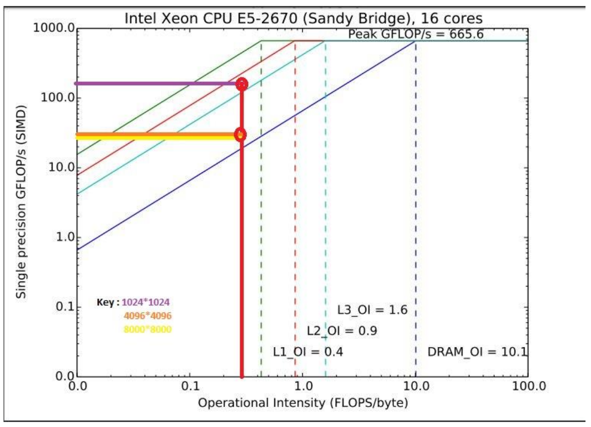

Figure 9: *Roofline model showing floating point performance with the parallel code running on 16 cores for 1024, 4096 and 8000 inputs.*

IV. Additional optimisations

A few other optimisations were tried with respect to using non-blocking communication modes like `MPI_Isend` and `MPI_Irecv` to make the code more robust against deadlocks. However, the problems faced were that the program on a few ranks commenced on incorrect data and to ensure valid data receipt wait calls had to be placed in appropriate positions in the code. These wait calls led to some processes waiting for data longer than others. Given the format of `stencil` implementation, no core could commence processing until it had valid data in the halos and therefore this asynchronous communication pattern did not lead to any performance improvement and was reverted to use `MPI_Send` and `MPI_Recv` with row-based domain decomposition.

The GCC compiler (`gcc-7.1.0`) with –O3 flag was also used to compile the column-wise decomposed grid parallel code, so as to gauge any compiler level performance benefit and compare it with ICC compiled performance gains. The run-times achieved were 0.04s for 1024,1.09s for 4096 and 3.7s for 8000 image sizes. These times were similar to what was achieved with the ICC compiler on the same code. Hence, I carried on using the ICC compiler (Version 16, Update 2).

Dynamic memory alignment options `like _mm_malloc (a, 64)` and `__assume_aligned(a, 64)` were explored as well. However, this did not lead to any appreciable performance improvement. This may be because the computation speed on the smaller sub-grids of the data on each core overshadowed the memory alignment improvements.

V. Conclusion

The fastest timings achieved after porting the serial code to run on parallel cores with MPI are highlighted in bold in Table 3. The code was compiled `with mpiicc -O3 -xHOST -std=c99 -Wall stencil.c -o stencil` before launching to run on 16 cores.

Image Size	1024*1024	4096*4096	8000*8000
Parallel Code Run-times	**0.01s**	**0.5s**	**2.03s**
Speed-Up	8x	5x	4x

Table 3: *Final Optimisation Results with speedups calculated against serial optimized run-times.*

On the whole, this parallelized code (domain decomposition by rows) is on average 80%, 50% and 40% faster than the serially optimized version for 1024, 4096 and 8000 image sizes.

References

[1] Intel Compiler Optimisations - https://software.intel.com/en-us/articles/step-by-step-optimizing-with-intel-c-compiler

[2] https://blogs.cisco.com/performance/send_isend_or_sendrecv

[3] Intel Library Developer Reference- https://software.intel.com/en-us/mpi-developer-reference-windows-global-options

[4] https://www.open-mpi.org/doc/v2.0/man1/mpirun.1.php