HPC - Parallelisation with MPI

Martin Dimitrov
md16024 1644531

1 Introduction

The objective of this report is to outline my research on optimisation via parallelisation. In particular, it focuses on the improvements I have made for my serially optimised *stencil* code following the MPI standard. MPI falls under the Single Program Multiple Data (SPMD) model.

The hardware resources we have been given are 2 nodes each with 2 Xeon Intel E5-2680 CPU's. As each CPU has 14 cores, these give us a total of 56 cores available. The report will include results and analysis of the code running from one core up to all cores of both nodes.

2 Architecture

The first step was to decide the model of communication for the program. The final workflow follows as:

1. the first worker (master) initialises the whole image
2. distribute it across all cores including itself (with *MPI_Scatter*)
3. stencil function gets executed in all ranks
4. master collects (with *MPI_Gather*) and outputs the result

This architecture is not initialising the whole image in each thread, saving otherwise duplicated data. It also involves fewer messages sent than the master-slave paradigm. Note that each worker depends on its neighbours. In order to update the data needed between them, each iteration of step 3 consists of *halo exchanges*.

Throughout this project I have compiled using Intel's *icc* compiler with the *-Ofast* and *-xBROADWELL* flags.

3 Domain decomposition

There are multiple ways of splitting the data between workers. This report analyses the row and column splits.

The image in my serially optimised *stencil* code is a one-dimensional array initialised in row-major order. Splitting the data into columns instead of rows proves less useful as workers do not operate on contiguous regions in memory. This raises two issues. First, it lowers cache hits during the stencil function as the double loop accesses the array in a row-major manner. Second, it increases the memory access overhead whilst halo-exchanging as each cell is located *slice_width* apart in memory. In general, a good domain decomposition directly relates to how the data is initialised and accessed. In the case of my *stencil* code, row decomposition fits better.

4 Row Decomposition

To cater for images with non-divisor cohort sizes, I initially allocated all of the remaining rows to the last core. This, however, means that the program is as fast as that core. A better distribution is to allocate the remaining rows evenly between all ranks. In the case of this project where the remainder can only increase up to 56, there is not a noticeable difference between both approaches. I stuck with the latter as it scales better with more cores.

Using a row decomposition with the *Intel* implementation of MPI, I ran all core configurations from 1 to 56 and averaged it across 10 runs. Table 1 displays the best speeds we get (in bold) across all images with their corresponding core configuration.

Table 1: Core configuration that yield the best runtimes (in seconds) for each image in bold. Using a row decomposition

	1024	4096	8000
1	0.09	3	11
32	**0.008**	0.3	1.2
54	0.03	**0.13**	0.9
56	0.04	0.14	**0.8**

Ultimately, we get runtimes down to 0.009s, 0.14s and 0.8s for the three corresponding images. Compared to the initial serial code with times 0.09s, 3s and 11s, we get increases of 10x, 21x and 13x.

Fig.1 displays more accurately how *stencil* scales using different core counts. All three resolutions follow the same steadily-increasing speedup per core curvature up until 30 cores. Note that all images get a 2x speedup (100% increase) using two cores over one. Isolating the distinct speedup/core curvatures per resolution problem gives us the strong scaling.

4.1 1024 analysis

The 1024x1024 reaches peak performance of 12x or 0.008s at 30-32 cores. Further increasing core count worsens the speedup ratio, going back down to 2x using all 56 cores. The plummet comes from the fact that by increasing cores we increase the competition for memory bandwidth. Because of the split, we already have low rows per core, so we cannot take advantage of further splitting up the data. For example, when running with 56 cores, each rank operates on only 17 rows. Comparing this with a version of *stencil* that only does halo exchanges shows that the stencil function makes little to no difference on runtime. Therefore making the 56 core configuration an overkill.

4.2 4096 analysis

The 4096x4096 image has the best scaling ratio out of all resolutions. It reaches 0.13s or 23x speedup when using 54 cores versus the serial version.

When having more than 30 cores, the speedup curvature resembles to be linearly increasing compared to the linearly decreasing one for the 1k image.

4.3 8000 analysis

The 8k image, on the other hand, seems to be following the trend of a diminishing return. It is most optimal at around 48-56 cores at 0.8s. This results in a rough 13x speedup over the serial code. Compared to the 4k image, the scaling ratios are far lower after the 30th core mark. This issue can be derived from improper caching. The CPU that we are using: Xeon Intel E5-2680 CPU has a 35MB L3 cache shared across 14 cores. In order to completely cache the image, the size per core should not exceed 2.5MB. We can calculate the image size per core using this formula:

$$image\_pixels/cores * data\_type * 2$$

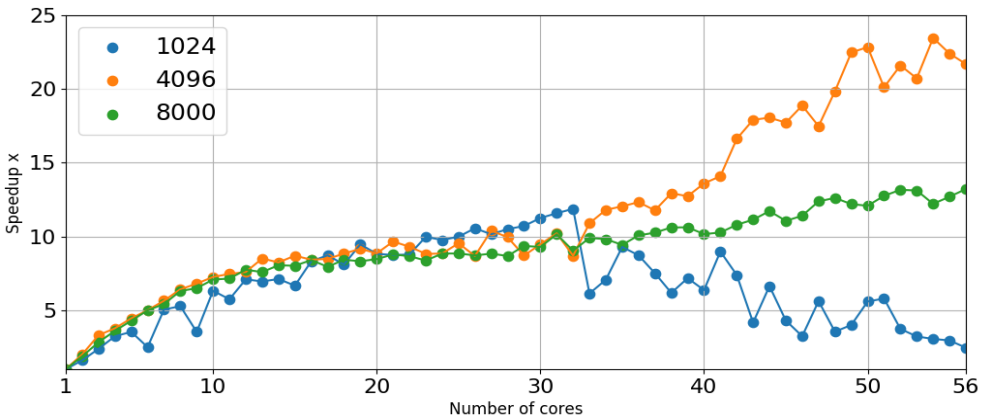

Figure 1: Speedup ratio for each core configuration compared to a single core.

In the case of 8k, 56 cores, the split will result in 9MB per core. As this far exceeds the cache space, the program needs to go to DRAM, hence, the slowdown. The 4k image on the other hand has 2.3MB per core. This falls within the L3 cache bounds, allowing the complete cache of the image and therefore better speedup ratios.

5 Message communication

The MPI standard supplies several methods for rank-to-rank communication. The implementation mentioned above uses *MPI_Sendrecv* for halo-exchanges.

Unpacking *MPI_Sendrecv* to the two separate calls *MPI_Send* and *MPI_Recv* proved to be almost 1.3x slower.

The stricter *MPI_SSend* is a synchronous call that blocks until the message has been received, unlike *MPI_Send* which utilises a buffer. Using it resulted in a 1.5x marginal decrease over the initial *MPI_SendRecv* as expected.

MPI_ISend is similar to *MPI_Send*, but returns immediately (it is non-blocking/asynchronous). This communication is performed via shared memory on a separate thread. In high core cases, it will only increase the competition for memory bandwidth. As such [fig. 2] displays the comparison between a (*MPI_ISend* - *MPI_Recv*) implementation to the original *MPI_SendRecv*. The idea is that *ISend* enforces the send to be asynchronous whilst we still block when receiving. For rank counts lower than 30 there is not a significant difference. As rank count increase we see that *MPI_ISend* has better speedups up until the 50th core mark, where the competition for memory bandwidth starts to slow down the program. (Note that this might also be caused by the bc4 slowdown) Using the Tau profiler we also see that the 1024 image spends a bit less time communicating (being blocked) using ISend.

6 OpenMP

The scaling for MPI-only code tails off at high core counts as the cost of communication increases relative to the compute (especially when strong scaling). By using a hybrid of MPI and OpenMP we can lower that cost. Essentially we reduce the communication amount by lowering the number of MPI ranks, hence, increasing the problem per rank. Each MPI rank then can spawn OpenMP threads which run in parallel and split up the problem within the rank. Launching 1 MPI rank per socket (aka 2 per node) and 14 OpenMP threads per rank allows all corresponding threads to run on one NUMA region. In other words, this configuration would best fit the available hardware and avoid cross socket communication within the NUMA architecture.

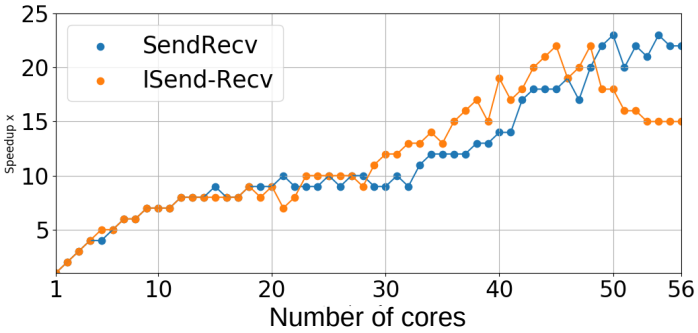

Figure 2: Speedup ratio comparison between SendRecv and ISend-Recv for the 4k image. Each data point comes from an average across 10 runs

I compiled the hybrid OpenMP and MPI (*MPICH*) configuration above (4 MPI ranks, 14 threads each) with close thread pinning using Intel's compiler *mpiicc* with the *Ofast* flag. Table 2 compares the runtimes to the initial serial version and the best timings we get from the MPI-only configuration from section four.

Table 2: Runtimes (in seconds) compared between serial, MPI-only and MPI/OpenMP versions of stencil using Intel's icc compiler and the flag Ofast

	1024	**4096**	**8000**
Serial	0.09	3	11
MPI only	0.008	0.13	0.8
Hybrid	0.009	0.1	1

We see that both MPI-only and hybrid approaches yield very similar results

7 Performance Analysis

We examine that it is not always the case that having more cores equates to faster runtimes. As seen from [fig. 1] and said in section 4, 1024 runs fastest with 32 cores, 4k with 54 and 8k with 56. The final runtimes (per best configuration)

using Intel's MPI implementation, Intel's compiler *icc* with the *-Ofast,-xBROADWELL* flags and the *SendRecv* version for halo-exchanges are 0.008, 0.13 and 0.8 seconds. These correspond to speeds of 210, 206 and 128 GFLOP/s respectively. Compared to the performance of the serially optimised code of 18, 9 and 9 GFLOP/s, we examine a 12x, 22x and 14x increases.

I have constructed the roofline model for the 56 core configuration [fig. 3]. The runtimes using all cores from both nodes are: 0.03s, 0.13s, 0.8s or 56, 206 and 128 GFLOP/s. Note that the model was constructed using these bandwidth values[1]:

1. *L1* bandwidth: 151GB/s * number of cores
2. *L2* bandwidth: 54GB/s * number of cores
3. *L3* bandwidth: 13GB/s * number of cores
4. *DRAM* bandwidth: 65GB/s * sockets

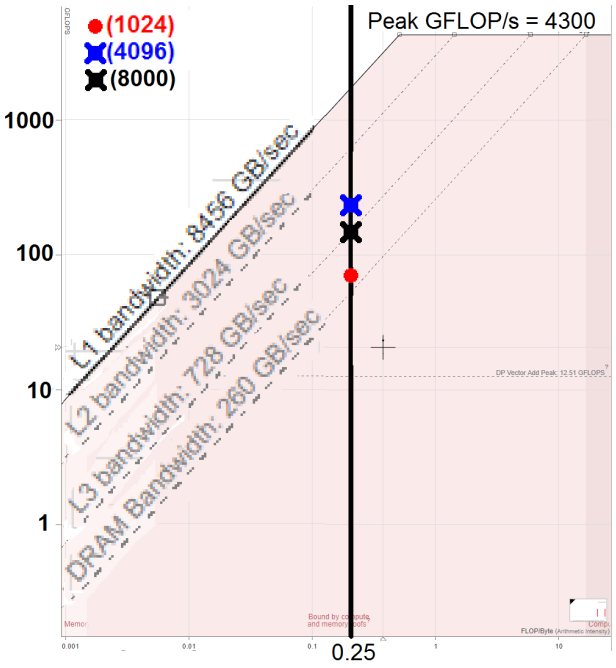

Figure 3: Roofline model for 56 cores

The IO has been calculated to be 0.25 as we have got 5 reads, 1 write and 6 operations of floats. As mentioned in section 4, we see that the 4k problem scales better with 56 cores than the 8k as of the cache size. The 1k image on the other hand scales a lot worse (down to a 2x speedup than the serial) as the high number of cores emphasise the communication overhead.

[1]These values were given by a TA and were confirmed on the forum