Code Parallelisation in MPI

Andreas Hadjiantoni[1]

[1] *1611368*

December 14, 2018

In this report a number of techniques to parallelise serial code are presented. The nature of the problem that the code solves belongs to the "Structured Grids" class/dwarf. The techniques used are referred as "Synchronous Data Exchange", "Asynchronous Data Exchange" and "Cartesian Topology Synchronous Data Exchange", and each of them are discussed in individual sections. The initial serial code used, on which the three parallelisation methods were applied is assumed to be efficient, given the limited hardware resources that it was using, and it is therefore sensible to consider using multiple cores.

1 Initial Optimisations

Firstly, some trivial optimisations were added which are not parallelisations. The **-Ofast** compiler flag was added, which disregards strict standards, and the compiler is therefore free to do more optimisations. Even though it ought to be used with caution, in this case the code works as it is supposed to, and so it will be kept. Furthermore the option **-genv I_MPI_PIN_PROCESSOR_LIST='grain=cache3,shift=sock'** is given to the **mpirun** command. This will ensure that the first eight ranks will be pinned to the one socket, and the other eight to the other eight. Hence, a $rank \rightarrow socket$ mapping is now established. This will be exploited and discussed in the next section.

2 Synchronous Data Exchange

The first approach is the most conceptually simple, and it works by splitting the grid to n distinct columns, where n is the number of available cores. Each core will then process one column, and as the problem is such that the next state of an arbitrary cell in the grid depends on its value in the current state as well as on the values of its neighbours, the cores will need to exchange data. Figures 1 and 2 show graphical representations of the running times achieved with this solution. Furthermore, the columns that the ranks were assigned are such that the ranks which are run on the same socket process the one half of the image, and the other ranks on the other socket process the other half. As the communication required is only between ranks which process adjacent image parts, this scheme ensures that most of the communication occurs within the sockets and not between them. Table 1 shows the running times of this approach for the images 1024 X 1024, 4096 X 4096 and 8000 X 8000 using 16 cores.

Memory Bandwidth Limitations The thread pinning technique allows us to reason about the anomaly in Figure 2, which is more obvious in the 8000 x 8000 image. As each socket has its own memory bus, the memory bandwidth is doubled when the ninth core is used and hence making the problem less memory bandwidth bound. In specific, the limitations imposed by memory bandwidth are obvious when 4-8 cores are used in Fig-

1024 X 1024	0.0074
4096 X 4096	0.52
8000 X 8000	1.98

Table 1: *Running times for the synchronous data exchange scheme using sixteen cores. The speedup compared to the serial case is not proportional to the number of cores, as memory bandwidth is now a bigger limitation due to the higher memory IO demand. Also network bandwidth affects the speedup negatively, which was not present in the serial case at all.*

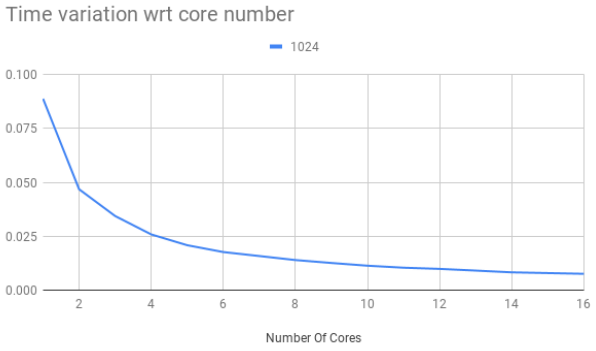

Figure 1: *In this figure, it is observed that the time needed to process the 1024 X 1024 size image decreases non linearly, and an asymptote starts to form. This is due to the fact that as the number of cores increases, the communication overheads also increase, with respect to the same image size. Also, more data IO is required to and from the memory with respect to the same memory bandwidth. Hence a memory and network bandwidth bound setup is created.*

ure 2, where no speedup is observed. The ninth core is therefore the next "useful" core, resulting in substantial overall speedup, as it uses the entire memory bandwidth of the second socket. The fact that the time decreases asymptotically in all three input sizes, but this anomaly is more obvious as the input size increases, implies that the problem becomes more memory bandwidth bound compared to communication bandwidth bound in larger input sizes. This is due to the fact that the memory and communication bandwidths required increase like $O(n^2)$ and $O(n)$ respectively, where n is the size of the side of the image.

Roofline Model The Intel Advisor 2019 is used to derive a roofline model of this approach. This allows to reason about the operational intensity

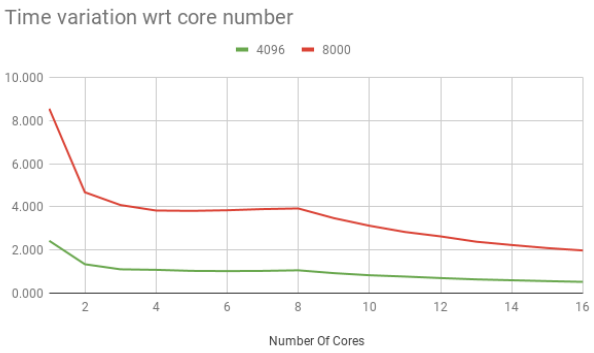

Figure 2: *Here, the same asymptotic decrease of time with respect to a linear increase of core number is observed. Additionally, in both lines a decreasing variation at nine cores is observed which does not follow the pattern.*

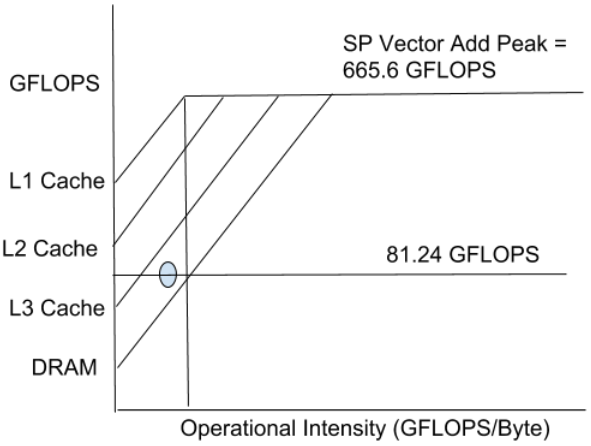

Figure 3: *In this roofline model it can be deducted that the program is memory bandwidth bound, as inferred from the running times in Figure 2. This is because the operational intensity of the for loop is on the left side of the ridge point.*

of the program. The result is shown in Figure 3.

3 Asynchronous Data Exchange

As communication adds time overheads in the program, it is attempted to minimise those overheads in the following way. Firstly, the image is logically divided in the same way (i.e. in equally sized columns). Secondly, the same data will need to be communicated, but the initiated communication is now asynchronous and non-blocking. This means that the program will not stop to perform communications at the point where the **MPI_Isend(...)** and **MPI_Irecv(...)** instructions are executed. Instead, the data will be copied in a buffer for the receiving rank to get it when it needs to and the

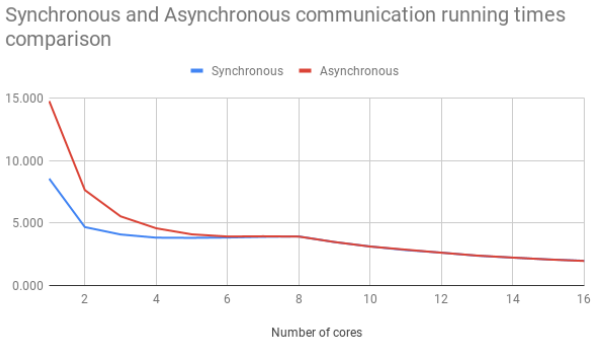

Figure 4: *In this figure, the running times for the synchronous and asynchronous schemes are contrasted, when the image sizes are 8000 X 8000 in both cases.*

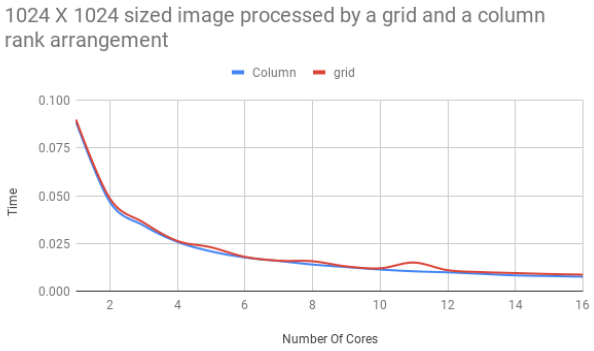

Figure 5: *Even though the grid structure should in theory perform better than the column structure for a larger number of cores, this is practically not the case, as memory bandwidth is the limiting factor when the number of cores increases.*

flow of the program will continue. This communication scheme can be exploited in the following way. We first initiate this communication for all ranks, for all of the data that needs to be exchanged in the current stencil iteration. As the program flow continues, the inner cells which do not depend on the data that is due to be transferred can now be processed. When their processing is done, the program pauses and waits for any communication to finish, and then processes the outer cells. This scheme relies on the fact that the asynchronous communication occurs during the processing of the inner cells, and hence minimises the overheads needed for communication. Even though this theoretical explanation implies a speedup and a network overhead minimisation, Figure 4 shows that this is not the case. The reason for not observing any speedup is that the non-blocking operations within a node would require a separate thread used explicitly for the communication, in order to achieve truly asynchronous data exchange. In this case, the thread performing the communications is the same as the one performing the computations, and due to that limitation, the data exchanges are non-blocking, but they are not truly asynchronous. Looking at a lower level, since all of the communication takes place within the same node, the communication is essentially one or more memory copy operations, which may happen at any point between the **MPI_Isend(...)** or **MPI_Irecv(...)** and **MPI_Wait(...)**.

4 Cartesian Topology Synchronous Data Exchange

In this section, the nature of the communications will not change. I.e. synchronous and blocking communications will be used. However, the segmentation of the image changes. In particular, a grid structure will be used. Hence, each rank will now need to send and receive data not only to the ranks that are logically left and right to it, but also above and below it. The running times for the 1024 X 1024 image are shown in Figure 5.

Theoretical evaluation of this approach The grid based approach scales better compared to the column based approach in terms of the amount of communication needed, with the number of cores being the degree of freedom. This statement is formalised in the following Lemma:

Lemma 1. *The amount of communication needed to update the state of a structured grid problem in the column and grid based approaches scales like $O(n)$ and $O(\sqrt{n})$ respectively, where n is the number of cores.*

Proof. Let n be the number of cores used.
Without loss of generality, assume that the grid is a square.
Since the grid is a square, let the size of its side be m.

We first consider the column based approach. The amount of communication per core can be

m or $2m$ depending on whether the core has a neighbor or not. In either case the communication can be generalised to be $O(m)$ and $O(mn)$ if all of the cores are considered.

The grid based approach is now considered. There will be $\sqrt{n}$ cores in each side of the grid. The amount of communication per core can be $\frac{4m}{\sqrt{n}}$ or $\frac{3m}{\sqrt{n}}$ or $\frac{2m}{\sqrt{n}}$ depending on whether the core is surrounded by other cores, whether it is adjacent to the edge of the grid, or whether it is one of the four corners. In any of those cases the communication can be generalised to be $O(\frac{m}{\sqrt{n}})$ and $O(\frac{nm}{\sqrt{n}}) = O(\sqrt{n}m)$ is the communication if all of the cores are considered.

Since the degree of freedom was assumed to be the number of workers, m can be considered to be a constant factor, and therefore: $O(mn) = O(n)$ and $O(\sqrt{n}m) = O(\sqrt{n})$. This result proves the lemma. □

Corollary 1. *The amount of communication needed to update the state of a structured grid problem in the column and grid based approaches scales linearly in both cases with respect to the size of the side of the image.*

Proof. In the above proof it was derived that in the grid and column based approaches, the communication scales like $O(\sqrt{n}m)$ and $O(mn)$ respectively. If the image size is now considered to be the degree of freedom, n is therefore a constant factor and thus $O(mn) = O(\sqrt{n}m) = O(m)$. □

5 Future Directions

In this section, further experiments are proposed, which will relax the limitations of the approaches presented above.

Grid based approach Even though the grid based approach was proven to require asymptotically less communication than the column based approach, in order to verify **Lemma 1** in a practical setting, one should consider solving a structured grid problem using both approaches. Also, it must be ensured that:

- There is no memory bandwidth bound.
- The number of cores used is large enough.

Both of the above requirements can be ensured by using a large number of nodes, and using only one core per socket. This will ensure maximum memory bandwidth per core, and the communication throughput will be decreased, as the data is transferred between the nodes rather than within, and thus it needs to travel through the network. Additionally, a slower communication medium can be used, which will amplify the impact of data transfer across the network.

Asynchronous Data Exchange It is possible to explore the asynchronous communication approach by ensuring that one core is used solely for communication, or by requiring inter node communication, in which case the network card will need to make the data transfer, instead of the core itself.

Memory Bandwidth As shown in Figure 2, the program becomes memory bandwidth bound very quickly as more cores are used. One approach to explore the memory bandwidth and the limitations it imposes would be to make it the degree of freedom, and fix the number of cores. Hence for a number of cores n, the program could be run on 1 to n sockets, (i.e. 1 to $n/2$ nodes), assuming that each socket contains at least n cores. Since each socket has its own memory bus, the memory bandwidth will vary depending on the number of sockets and it will therefore be the degree of freedom.

References

[1] *OpenMPI Documentation* https://www.open-mpi.org/doc/.

[2] *Intel Process Pinning* https://software.intel.com/en-us/mpi-developer-reference-linux-environment-variables-for-process-pinning