COMP4300 Assignment 1

Jackson Kilrain-Mottram (u6940136)

April 2023

1 Question 1

The parallel advection solver deadlocks for values of N greater than or equal to 2^{15} (32768). The `MPI_Send` function has different behaviour depending on the total count of elements to send. In this case 2^15 is the upper limit to internally buffered, non-blocking behaviour. Beyond this threshold, the data is not copied into an internal buffer and as such must wait for the send to complete (a matching `MPI_Recv` is invoked) before returning control to the application. Given that all of the processes are using the same model of sending before waiting to receive, they will all send in a blocking manner and none will initiate a receive as it hasn't finished sending yet. To fix this, we can either start with a send or a receive depending on the rank modulo 2. This means that half of the processes will start sending and the other half will start by receiving, ensuring no deadlocks. MPI provides a neat abstraction in the form of `MPI_Sendrecv` to handle this per-rank organisation still using `MPI_Send` and `MPI_Recv` under the hood.

2 Question 2

In order to determine the impact of the non-blocking communication compared to the blocking communication, we need to devise a parameter set that maximises communication overhead. Given the nature of the solver is implemented via domain decomposition, the optimal method for increasing communication is to maximise the number of local domains.

That is to say, minimising `M_loc` and `N_loc` such that $P \times Q$ approaches $M \times N$. In the maximal case, there is a single process associated with each element of the matrix $M \times N$, with halo width one. This optimises for communication, where every element performs 8 individual exchanges (8 sends, 8 receives) along each of the four sides and the four corners. This results in the following equation modelling the total communications per timestep:

$$C = \left(4 + \frac{2M}{P} + \frac{2N}{Q}\right) \cdot P \cdot Q$$

We can look at solving for P and Q for some M, N, by maximising the value of C with respect to these variables. More formally:

$$\forall M, N \operatorname*{argmax}_{P,Q} \left\{ \left(4 + \frac{2M}{P} + \frac{2N}{Q}\right) PQ \;\middle|\; P, Q, M, N \in \mathbb{Z}^+ \wedge P \leq M \wedge Q \leq N \wedge (PQ) \bmod (MN) = 0 \right\}$$
$$\implies P = M, Q = N$$

Inspecting the function, we can see that it is monotonic due to the restriction of the modulo between PQ and MN along with the multiplicity requirement of the $\frac{2M}{P}$ and $\frac{2N}{Q}$ constraints in the equation.

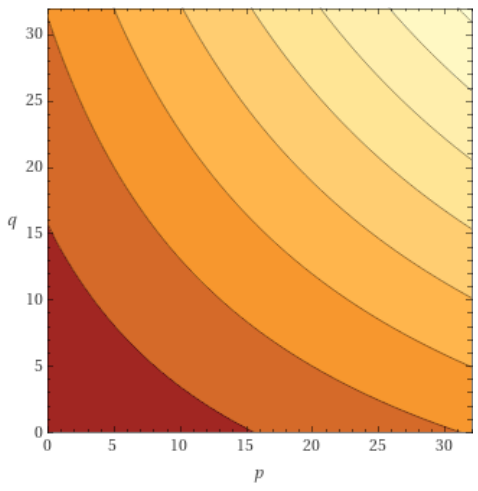

Figure 1: Contour plot of C with $M = N = 32$ without the modulo constraint

From figure 1 we can see that contour of this function strictly increases monotonically along the diagonal from the minimum values of $P = 1, Q = 1$ to the maximum values $P = M, Q = N$. All solutions of the argmax will lie at the maximum of this function, which is indicated to be the maximum of the diagonal. This shows that the optimal values for P, Q are M, N respectively for maximising communication. (This is not rigorous, more of a sketch as the focus is not on this statement of maximisation).

For a 1D process grid, we can maximise communication by choosing $P = M$ such that every process affords two top/bottom border exchanges as an entire row is done in a single communication. Additionally, two left/right border and four corner exchanges are done as column exchanges on the local advection field. This yields maximum communication while also minimising the flops per timestep, as each process is computing the minimum field evolution, thereby further skewing the overall performance towards the communication overhead.

We can't make the value of P, Q, M, N arbitrarily large, due to the limiting factor that process count scales exponentially with matrix size. We evaluate the following values for the parameters:

- $Q = 1, P = M = N = 192$, advection field size $36,864$, comms 384
- $Q = 1, P = M = N = 256$, advection field size $65,536$, comms 512
- $Q = 1, P = M = N = 512$, advection field size $262,144$, comms $1,024$

Blocking(GFLOPS)	**Non-Blocking (GFLOPS)**	**Blocking(Time)**	**Non-Blocking(Time)**
8.04e+00 4.19e-02 (PC)	3.71e+01 1.93e-01 (PC)	9.17e-03s	1.99e-03s

Table 1: Results of first configuration $Q = 1, P = M = N = 192$

Blocking(GFLOPS)	**Non-Blocking (GFLOPS)**	**Blocking(Time)**	**Non-Blocking(Time)**
9.06e+01 3.54e-01 (PC)	1.27e+024.97e-01 (PC)	1.45e-03s	1.03e-03s

Table 2: Results of second configuration $Q = 1, P = M = N = 256$

Blocking(GFLOPS)	**Non-Blocking (GFLOPS)**	**Blocking(Time)**	**Non-Blocking(Time)**
2.08e+02 4.06e-01 (PC)	2.13e+02 4.17e-01 (PC)	2.52e-03s	2.46e-03s

Table 3: Results of third configuration $Q = 1, P = M = N = 512$

From these results, we can gather that the non-blocking implementation is definitely better. It outperforms the blocking implementation in both time and GFLOPS. Interestingly, with higher field sizes and processor counts, the difference grows smaller, it's hard to say conclusively from these results whether this is absolute or an artefact of the limited scope here. In any case, the non-blocking variation can be seen to outperform the blocking.

3 Question 3

3.1 1D Model Derivation

In the 1D model, each process performs two communications of the same form, which can be modelled as

$$\frac{M}{P} \cdot 4t_w + t_s$$

Where the $\frac{M}{P}$ value denotes the row dimension of the local advection grid, each element of which contains doubles. Each double is contained in 4 words, so the $4t_w$ factor account for this. Lastly, the t_s supplies the startup overhead for communicating the row. Given that each process performs this twice, we multiply this expression by two:

$$2 \cdot \left(\frac{4Mt_w}{p} + t_w \right)$$

We need to include the cost of computing the advection field after the communication occurs. Given that the local advection field dimensions are factors of the process grid, we can express the local grid size as the product of the dimension ratios

$$\frac{M}{P} \cdot \frac{N}{Q}$$

Given that each element of the advection grid employs as cost of t_f (as given by the question statement), we multiply the ratio expression by this to result in the total computation cost for the local advection grid. Combining this with the expression for the communications and accounting for r iterations, the final model is

$$r \cdot \left(2 \cdot \left(\frac{4Mt_w}{P} + t_s\right) + \frac{MNt_f}{PQ}\right)$$

3.2 Measuring Coefficient Values

To gather experimental values for the time valued variables (t_s, t_w, t_f), we can use timers and numerical methods based on the timing requirement.

3.2.1 Evaluating t_f

The most straightforward to capture is t_f which can be done by wrapping the double loop within the `updateAdvectField()` function invocation with `MPI_Wtime()` before and after, then computing the average of all iterations averaged over invocations. For this we use, $48, 96$ and 192 cores in distinct runs over $r = 100$ iterations, with field dimensions $M = N = 1000$.

Core Count	**Average t_f Duration**
48	$1.6338549417164163e-09$
96	$1.8313747096703432e-09$
192	$1.9773684770638774e-09$

Table 4: Average measured times for a single field element update.

Examining the results, we can see that on average the duration for a local field element update is $\approx 1.8141993761502122e-09$ seconds or $\approx 1.8141993761502122062$ nanoseconds.

3.2.2 Evaluating t_s

If we consider the composite form of a communication cost as an expression, it is comprised of the startup time t_s and the cost of sending n words of data as nt_w:

$$t_{\text{comm}} = t_s + nt_w$$

So, we can imagine, that if we send no data for a given communication, then the cost is simply

$$t_{\text{comm}} = t_s$$

This is precisely how we can measure the value of t_s. Implementation wise, this can be done by a ping-pong message exchange between two processes with a message of size zero. In order to determine a value accurately, we need to account of the overhead of the timers themselves. This can be done by taking the average of a timer that is started and immediately stopped several times then dividing the value by two. We reduce the final t_s value measured by this amount to achieve an accurate measurement. (See `startup.c` for more details on the implementation).

Buffer size	**Measurement**
100	$2.7e-04$
1000	$3.35e-04$
10000	$2.9e-04$
100000	$2.96e-04$
1000000	$2.46e-04$
10000000	$3.34e-04$
Average	$2.951666666667e-04$
Average (One Way)	$1.47583e-05$

Table 5: Measurements of t_s without timer overheads with given buffer sizes. Note that the `MPI_Send` and `MPI_Recv` calls specify a size of 0 explicitly.

Computing the average of the averages, we see that t_s is approximately $1.47583e-05$ seconds, or ≈ 14.7583 microseconds.

3.2.3 Evaluating t_w

In the previous section, we characterised the value of t_s through experimental analysis. That is an important stepping stone to evaluating t_w, as we need to eliminate the overhead from our measurements. Given that we have already performed the necessary measurements to determine the ping-pong time for various word sizes in previous lab tasks, we can use the values to calculate the cost of t_w.

In prior, we evaluated several transfers of varying word counts from 1 to 4194304. Each of which gave slightly varying values for duration for ping-pong to complete on average. In the below table, the raw data is set out, including only the ones with large enough buffer sizes that the measurements can be used accurately.

Word Count	**Duration**	t_w
16384	$1.51e-05$	$2.0855712890624925e-11$
65536	$5.56e-05$	$6.231948852539063e-10$
262144	$2.06e-04$	$7.295291900634766e-10$
1048576	$8.05e-04$	$7.536332130432129e-10$
4194304	$4.66e-03$	$1.1075119256973267e-09$

Table 6: Raw measurements of ping-pong for various sizes payloads (words) [Sourced from course site LAB1 notes]

The values of t_w computed in the table above is done according to the following formula $\frac{\text{duration}-t_s}{\text{word count}}$. We can take an average of all of these values to find an approximation for the value of $t_w \approx 6.469449853897094e-10$. Note that t_s substantially (several orders of magnitude) larger than the value of t_w. This gives a good indication that smaller message sizes will incur a heavier overhead as the actual transfer time may well be less than that of the startup time. This would suggest that to effectively armortise the startup time over communication, one should choose to use larger messages that ideally incur the same cost for transfer in total as that of the startup time.

3.3 Strong Scaling Analysis

From the output of `lscpu` on Gadi, we can see that the L3 cache size is 36608KiB (binary units, not SI). As such, the equivalent in bytes is 37486592, which implies a maximum of 4685824 doubles (8 bytes per double) stored contiguously, can fit in the L3 cache. Therefore our maximum field dimensions, that are closest to being square are 2048×2288. In the interest of making this easier to handle, we will opt for a purely square variant of a slightly smaller set of dimensions 2048×2048, which fills $\approx 89.51\%$ of the L3 cache. Given that this is a power of two, it also allows for cleaner scaling of the process grid dimensions.

4 Question 4

4.1 2D Model Derivation

In order to account for the extra dimension of communication in the 2D model, we need to express the communication of the columns and corners in this model. Notice that the computation cost does not change, as it is not bound by any communication factors.

To remodel the equation of communication, we consider the extended case of replicated communications about the diagonal of the local advection field. That is to say, that we have a single column and a single row exchange with two corners.

In our existing expression, we can include a similar expression for the column cost as have done for the row. Here we exchange the dimension specific terms and include it as follows

$$2 \cdot \left(\frac{4Mt_w}{P} + \frac{4Nt_w}{Q} + t_s \right)$$

For each corner, we need to send a single double which is modelled as the cost of four words. Within the same expression, we include this as a separate term

$$2 \cdot \left(\frac{4Mt_w}{P} + \frac{4Nt_w}{Q} + 8t_w + t_s \right)$$

Note that we now have a total of four distinct communications occurring, so we account for this by scaling the t_s factor by four to model each communication startup overhead cost individually.

$$2 \cdot \left(\frac{4Mt_w}{P} + \frac{4Nt_w}{Q} + 8t_w + 4t_s \right)$$

Using this new expression for communication, we can replace the original in our model, with some factoring of terms to achieve the following model.

$$r \cdot \left(8 \cdot \left(\frac{Mt_w}{P} + \frac{Nt_w}{Q} + 2t_w + t_s\right) + \frac{MNt_f}{PQ}\right)$$

5 Question 5

Notes

- 2D hard because of corners
- Need to either exchange corners explicitly or add a second layer of overlapping comms after top/bottom exchanges received to send/recv left/right with corners
- Balance between communication cost and computational cost needs to be drawn. Ideally should be as equal as possible relative to the problem size.
- Performance model
  - Need to compute max of computation and communication that overlap
  - Take computation and communication to be of the same duration and treat as single operation
  - Supposing a computation heavy problem with small comms of large content, we would use the computation. In the inverse condition, consider the communication as the value
  - Useful: https://www.osti.gov/servlets/purl/944757

Performance tests
M=N=1000,P=100,Q=10
M=N=10000,P=

6 Question 6

6.1 Wide Halo 2D Model Derivation

Accounting for halo width within our existing 2D model, can be done be re-evaluating the basis for which unique operations are performed within the r iterations. Notice that, we are performing communication of the halo every w iterations, and within these iterations successively smaller advection computations are combined to form the advection field updates.

With this in mind, we can change our model to be based on the accumulation of this chunk of work repeated $\left\lfloor \frac{r}{w} \right\rfloor$ times. It is important to recognise that we still get complete solutions even if iterations terminate mid-way through a chunk. Including this in our new model, we have the following form

$$\left\lfloor \frac{r}{w} \right\rfloor (t_{\text{comm}} + t_{\text{comp}})$$

Given that our model now considers chunks, we know that for a given chunk, communication only occurs once. As such we can directly use the form of the communication cost derived in the original 2D model. Including this results in the following

$$\left\lfloor \frac{r}{w} \right\rfloor \left(8 \cdot \left(\frac{Mt_w}{P} + \frac{Nt_w}{Q} + 2t_w + t_s\right) + t_{\text{comp}}\right)$$

A given chunk, will perform w successive advection field computations, where each computation is incrementally smaller in its size to evaluate the full halo over the independent centre region. Let us first consider our original 2D model expression for the computation cost

$$\frac{M}{P} \cdot \frac{N}{Q} \cdot t_f$$

For each of the local field dimensions, expressed here as ratios, we need to include an additional number of variable halo cells specified by w (Let D be a given field dimension ratio as $\frac{M}{P}$ or $\frac{N}{Q}$). Given that the halo is around the whole local field, we extend the notion of dimension to include this

$$D + 2w$$

Assuming we have some index i which identifies which iteration within the current chunk is being performed, we can adjust the expression uniformly based on this index to decrease the halo for each iteration

$$D + 2w - 2i$$

If we replace the original dimension ratios with this new expression, we have a form that expresses the computation cost for any given iteration within a chunk. Given that our new 2D model is in terms of chunks, we wrap this with a sum over the w iterations within a chunk for the final model

$$\left\lfloor \frac{r}{w} \right\rfloor \left(8 \cdot \left(\frac{Mt_w}{P} + \frac{Nt_w}{Q} + 2t_w + t_s \right) + \sum_{i=1}^{w} \left[\left(\frac{M}{P} + 2w - 2i \right) \cdot \left(\frac{N}{Q} + 2w - 2i \right) \cdot t_f \right] \right)$$

6.2 Discussion

The use of wide halos is definitely beneficial, in that provides a means to reduce the overhead of communication at every iteration. In theory this could provide better scaling performance for wider halos, though a balance needs to be maintained between the overhead of large communications versus the cost of computation of larger local advection fields. With smaller processor availability this is a viable solution to catering for performance with communication hungry configurations, around smaller values of P and Q.

However, for larger process grids, this becomes harder to optimise as there is a limit to effective halo size, being that of the local advection field size. At this point, the overlap is absolute so over-computation becomes dominant, leading to inefficiency of a different kind.

7 Question 7

[Note: Optimal ghost zone size performance model: M. Ripeanu, A. Iamnitchi, and I. Foster. Cactus application: Performance predictions in a grid environment. In EuroPar'01, 2001.]

Iterative Stencil Loops (ISLs) are computational models designed to evolve a grid with some given transition function for every cell based on considering a defined neighbourhood pattern. These are highly parallelisable computations at the behest of requiring communication and synchronisation between the threads to ensure consistency within the processing loops. The overhead introduced by the synchronisation and communication between the parallelised units of work can impede the performance of the solver.

Tiled stencil computations (TSCs) were introduced to address the overheads of communication and synchronisation by breaking up the problem space into an atomic unit of execution known as a tile [https://www.sciencedirect.com/science/article/pii/ During the execution of a tile, no communication or synchronisation is used to amortise communication start-up cost over the larger execution time of a given tile [https://www.sciencedirect.com/science/article/pii/074373159290027K]. Over the problem space, tiles are required to be identical everywhere except for the boundary. Boundary tiles should be regular in the pattern as a consequence of regularity and identicality in the wider space. This is to address the complexities of partitioning the problem space efficiently and deterministically to ensure a balanced execution load for tiles. In order to reduce the communication and synchronisation even further, ghost zones can be used to partially overlap computation between tiles to avoid the need for communicating boundary context between tiles [https://link.springer.com/article/10.1007/s10766-010-0142-5]. One can look to overlap the boundary communication with the computation of the interior tile elements in order to amortise the boundary communication time.

TODO: EFFECTIVENESS
2D TSCs are well defined and provide highly optimised data profiles in terms of array access and partitioning the data amongst tile sets. This extends to the choice of tile size which is relative to the problem space and infrastructure available [TODO REF]. Prior work has explored the usage of automatic tile size determination to optimise this case.

Moving to the 3D and higher dimensional problem spaces introduces new issues. Suppose a 3D tiled Jacobi iteration with Von Neumann neighbourhood 1, cache accesses are limited by the extra dimensional constraint which bridges tile hyperplanes in the problem space. This introduces overhead in the processing time and also limits the access behaviour, noting that cache lines are replaced frequently where the array of data in the strip of a particular hyperplane is actively replacing the same cache line. This causes access conflicts and poor cache locality [https://dl.acm.org/doi/pdf/10.5555/370049.370403].

[https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=arnumber=6569833] present the in-plane loading memory access method to incrementally update the output element on the halo with partial reads from neighbour data. This is opposed to the forward-plane methods that fetches all neighbour values in one batch for computing each element of the halo.

Optimisations: https://dl.acm.org/doi/pdf/10.5555/1413370.1413375 https://ipcc.cs.uoregon.edu/lectures/lecture-8-stencil.pdf

To look at:

- FFT accelerated stencil computation: https://arxiv.org/pdf/2105.06676.pdf
- Accelerating stencil computations with kernel convolutions: https://ieeexplore-ieee-org.virtual.anu.edu.au/stamp/stamp.jsp?