Assignment 1 COMP4300

Abhaas Goyal (u7145384)

April 28, 2021

Table 1: Table abbreviations

$\mathbf{A_T}$	Advection Time
$\mathbf{N_{Send}}$	Blocking Send
$\mathbf{I_{Send}}$	Non-Blocking Send
$\mathbf{G_F}$	Gigaflops per second
$\mathbf{P_C}$	Per core (with G_F)
NP	Number of processes
$\mathbf{C_T}$	Calculated time

1 Deadlock issues

The prototype function `parAdvect()` can fail for total buffer length $> 2^{15}$ because `MPI_Send` is being called for 2 or more processes at the same time. When buffer length is small then the send is locally blocking (so it copies the message into the local buffer). But for larger value of total buffer length it becomes globally blocking (so both the messages now are stuck in the first `MPI_Send` waiting for the other process to receive). Hence, we face a deadlock situation here.
The simplest solution is to divide up the ranks into even and odd, and switch up the sending and receiving these classes.

```
if (rank % 2 == 0) {
    MPI_Send(&V(u, M_loc, 1), N_loc, MPI_DOUBLE, botProc, HALO_TAG, comm);
    MPI_Recv(&V(u, 0, 1), N_loc, MPI_DOUBLE, topProc, HALO_TAG, comm,
             MPI_STATUS_IGNORE);
    MPI_Send(&V(u, 1, 1), N_loc, MPI_DOUBLE, topProc, HALO_TAG, comm);
    MPI_Recv(&V(u, M_loc+1, 1), N_loc, MPI_DOUBLE, botProc, HALO_TAG,
             comm, MPI_STATUS_IGNORE);
} else {
    MPI_Recv(&V(u, 0, 1), N_loc, MPI_DOUBLE, topProc, HALO_TAG, comm,
             MPI_STATUS_IGNORE);
    MPI_Send(&V(u, M_loc, 1), N_loc, MPI_DOUBLE, botProc, HALO_TAG, comm);
    MPI_Recv(&V(u, M_loc+1, 1), N_loc, MPI_DOUBLE, botProc, HALO_TAG,
             comm, MPI_STATUS_IGNORE);
    MPI_Send(&V(u, 1, 1), N_loc, MPI_DOUBLE, topProc, HALO_TAG, comm);
}
```

Figure 1: Splitting and switching receiving and sending for odd and even ranks

Now it seems to be safely working for large values of M and N.

2 The effect of non-blocking communication

2.1 Implementing Non-blocking communication

- `MPI_ISend` and `MPI_IRecv` were implemented with the same style as given in the original question with `MPI_ISend` and `MPI_IRecv`, which form the basis for later questions.
- The above was done keeping in mind that there were no race conditions during top/bottom and left/right halo exchange

2.2 Timing of Blocking vs Non-blocking communication rationale

- The timing was defined by using the difference of `M_Wtime()` between top and bottom halo sending and receiving of messages **within** the `updateBoundary` function to maximize preciseness.
- In each repetition the total time accumulator was increased and in the end average time to update was taken. The source could be found in a previous commit (link). The basic methodology to reduce and put the answer is here as follows

```
avg_time = total_time / reps;
MPI_Reduce(&avg_time,&root_avg , 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
if (rank == 0) {
  printf("%d: Average time to update boundary wrt reps: %.1e\n", rank,
         root_avg/nprocs);
}
```

- The tests were done with small values of M and huge values of N since we need to focus on timings of communication here and in top/bottom exchange large values of N would mean a larger chunk of block to sent/receive.
- It is also to be noted that the value of M should atleast be the number of working nodes for all nodes participating together in alteast one send/receive. Hence, I chose M as {48, 192} for tests
- Another advanctage of keeping the value of M small is that we can still exploit the L3 cache with larger values of N
- Number of reps were taken to be 100 for making the sending/receiving average more consistent
- The tests were done 3 times and we were provided with consistent values.

2.3 Results

- For small values of N (Table 2), got varied results with in some cases blocking being faster and in others non-blocking being faster

- For large values of N (Table 3), one could clearly see that non-blocking sends/received proved to be consistently faster to a noticeable extent. Hence, we would be using that for the following set of questions.

Table 2: Blocking vs Non Blocking Send for Small values of N

M	**N**	**NP**	**N_{Send}**	**I_{Send}**	**Speedup**
48	1000	3	4.5e-06	3.2e-06	1.406
48	1000	6	3.1e-06	3.1e-06	1.
48	1000	12	1.3e-06	9.3e-07	1.397
48	1000	24	1.6e-06	8.0e-07	2.
48	1000	48	8.3e-07	4.7e-07	1.7659574
192	1000	48	4.5e-07	8.9e-07	0.505
192	1000	96	8.6e-07	3.3e-07	2.606
192	1000	192	3.7e-07	1.7e-07	2.176

Table 3: Blocking vs Non Blocking Send for Large values of N

M	**N**	**NP**	**N_{Send}**	**I_{Send}**	**Speedup**
48	100000	3	3.6e-04	1.2e-04	3.
48	100000	6	1.3e-04	7.2e-05	1.805
48	100000	12	8.2e-05	8.2e-05	1.
48	100000	24	4.2e-05	3.4e-05	1.235
48	100000	48	2.0e-05	1.2e-05	1.666
192	100000	48	2.5e-05	2.1e-05	1.190
192	100000	96	1.4e-05	8.0e-06	1.75
192	100000	192	5.7e-06	3.9e-06	1.461

3 Make Performance modelling and calibration

3.1 Determination of base values

- We need to do strong scaling in one node of dual socket, 24 core Intel Platinum Xeon 8274
- **FLOPS** = nodes * sockets * cores * ops * clock time
  = 1 * 2 * 24 * 24 * 3.2 (GHz)
  = 3.684e+09F
- $\mathbf{t_f}$ = 1/FLOPS = 2.73e-10s
- $\mathbf{t_w}$ is calculated by sending a pong message of `double` type values of length 1024 bytes 100 times and taking the average time. It was found to be 5.67e-06s
- $\mathbf{t_s}$ (which is related to latency) is calculated by sending a single byte in the pong program (inspired from Lab 01) and dividing the result by 2 It is calculated to be 1.8/2 => 9e-07s

3.2 Performance Model

- **Parallel communication**

$$\begin{aligned} T_{comm} &= T_{top/bottom} \\ &= 4(t_s + N.t_w) \end{aligned}$$

- **Sequential computation** (for width = 1) (`p` is number of processes). As of now, we have the assumption of `Q = 1`. Considering 9 floating operations in 9 point stencil to `updateAdvect` and 1 operation copy back for each points, and a 5 set way pipelined instruction level parallelism, hence to stencil compute computation ($\mathbf{t_{update}} + \mathbf{t_{copy}}$) it would take 11/5 instructions –> approximately 3 cycles

$$\begin{aligned} T_{seq} &= t_{left/right} + t_{update} + t_{copy} \\ &= 2\frac{M}{P}t_f + 3\frac{MN.t_f}{P} \end{aligned}$$

- Total time

$$\begin{aligned} T_{tot} &= r.(4(t_s + N.t_w) + 2\frac{M}{P}t_f + 3\frac{MN.t_f}{P}) \\ &= r.(4(t_s + N.t_w) + \frac{M.t_f}{P}(2 + 3N)) \end{aligned}$$

3.3 Testing Methodolgy

- Performance model was tested in one node
- The goal was to minimize top and bottom halo exchange time. Hence, like in question 2, a large value of N and small value of M was taken. In this case `M = 48` and `N = 100000`. They remain unchanged for increasing NP in this case because we want to do strong scaling.
- Number of reps was taken to be 100

Table 4: Strong scaling on single Node

NP	A_T	G_F	P_C	C_T	Speedup
3	6.01e-01	1.60e+01	5.32e+00	4.19e+00	1
6	5.11e-01	1.88e+01	3.13e+00	1.10e+00	1.176
12	5.35e-01	1.79e+01	1.50e+00	0.84e-01	1.123
24	2.40e-01	4.01e+01	1.67e+00	0.48e-01	2.504
48	1.28e-01	7.53e+01	1.57e+00	0.33e-01	4.695

3.4 Results

`M = 48 N = 100000 reps = 100`

- My calculated values are overshooting the actual values (maybe because not considering cache hits/misses). However, both the empirical and actual calculations decrease with time.
- When going from 6 to 12 processors, I was surprised to see the advection time increasing instead of decreasing and for initial values of `NP` $\mathbf{A_T}$ doesn't seem to decrease that much. My best guess would be because of the memory hierachy present in NCI nodes. As the number of processors increase, the size of the data distribution decreases in each processor, hence more data can be stored in L1 and L2 cache with more `cache hits` . This leads to $\mathbf{t_{seq}}$ being less. Initially, they don't have much effect given the size of the data and the communication of the nodes is increasing (leading to the abnormality), however from `NP` > 12, the most of the blocks are small enough to be fit into lower levels of cache hierachy.
- Other than that, we see a consistent result of $\mathbf{A_T}$ almost halving after N=12, when everything is in L3.

4 The effect of 2D process grids

4.1 Theoretical time

- Here, Q > 1 adds to more packets being transmitted in `T_comm` (in left/right halo exchange), which was sequential till the previous question.
- **Parallel communication**

$$\begin{aligned}T_{comm} &= T_{top/bottom} + T_{left/right}\\ &= 4(t_s + \frac{N}{Q}t_w) + 4(t_s + \frac{M}{P}t_w)\\ &= 8t_s + (\frac{M}{P} + \frac{N}{Q})t_w\end{aligned}$$

- **Sequential computation** (for width = 1) (`p` is number of processes).

$$\begin{aligned}T_{seq} &= t_{update} + t_{copy}\\ &= 3 * \frac{MN.t_f}{PQ}\end{aligned}$$

- **Total time**

$$T_{tot} = r.(8t_s + (\frac{M}{P} + \frac{N}{Q})t_w + 3 * \frac{MN.t_f}{PQ})$$

- Block communication > Strip communication if

$$T_{top\_bot\_block} + T_{left\_right\_block} > T_{top\_bot\_strip} + T_{left\_right\_strip}$$
$$8t_s + (\frac{M}{P} + \frac{N}{Q})t_w > 3(t_s + N.t_w) + 2\frac{M}{P}t_f$$
$$t_s > \frac{3}{8}((N - (\frac{M}{P} + \frac{N}{Q}))t_w + 2\frac{M}{P}t_f)$$

- We find a similar graph as found in Lecture 11 (22). Hence, we need to use large values of M and N to see an improvement.

4.2 Results

- `M = N = 2000 (2 * L_3 cache has around 70 MB memory)`

Table 5: Computation for 2D process grids (1 Node) with Q >= 1

P	**Q**	**NP**	$\mathbf{A_T}$	$\mathbf{G_F}$	$\mathbf{P_C}$	**Speedup**
1	12	12	2.70e-01	2.97e+01	2.47e+00	1
1	24	24	8.19e-02	9.76e+01	4.07e+00	1
1	48	48	4.42e-02	1.81e+02	3.77e+00	1
2	6	12	2.69e-01	2.98e+01	2.48e+00	1.003
2	12	24	7.92e-02	1.01e+02	4.21e+00	1.034
2	24	48	3.35e-02	2.38e+02	4.97e+00	1.319
3	4	12	2.64e-01	3.03e+01	2.53e+00	1.022
3	8	24	8.08e-02	9.90e+01	4.12e+00	1.013
3	12	48	3.30e-02	2.42e+02	5.05e+00	1.339
6	2	12	2.65e-01	3.02e+01	2.52e+00	1.018
6	4	24	8.38e-02	9.55e+01	3.98e+00	0.977
6	8	48	3.06e-02	2.61e+02	5.44e+00	1.444
12	1	12	2.65e-01	3.02e+01	2.52e+00	1
12	2	24	7.74e-02	1.03e+01	4.31e+00	1.058
12	4	48	2.91e-02	2.75e+02	5.73e+00	1.518
16	3	48	2.35e-02	3.40e+02	7.08e+00	1.880
24	2	48	3.20e-02	2.50e+02	5.51e+00	1.381

- The speedup was taken w.r.t P=1 for a particular value of `NP`
- From Table 4, best ratio of P:Q is inspired from the performance model

$$min(\frac{M}{P} + \frac{N}{Q})$$

- On further investigation this estimate was found to be true from the data being provided

- Here, speedup is taken w.r.t Q = 1 unlike in previous table
- Till Q3 `Q = 1`, so performance improvement in optimal values of P and Q (when P is close to a near square ratio) are highly impressive. On optimum values around **4x improvement** is found on using this approach for large values of M and N.

Table 6: Computation for 2D process grids (4 Nodes) with Q >=1

P	**Q**	**NP**	$\mathbf{A_T}$	$\mathbf{G_F}$	$\mathbf{P_C}$	**Speedup**
48	1	48	2.70e-01	2.97e+01	2.47e+00	1
16	3	48	2.35e-02	3.40e+02	7.08e+00	11.489
96	1	96	4.97e-02	1.61e+02	1.68e+00	1
16	6	96	1.73e-02	4.62e+02	4.81e+00	2.872
192	1	192	4.40e-02	1.81e+02	9.47e-01	1
16	12	192	1.45e-02	5.51e+02	2.87e+00	3.034
24	8	192	1.37e-02	5.85e+02	3.05e+00	3.211
32	6	192	2.19e-02s	3.66e+02	1.90e+00	2.009

- If t_w were 10 times larger, then strip partitioning would have been better (from seeing the equation and calculating the values between strip and block paritioning the condition wouldn't hold true for the large values that we have tested against)

5 Overlapping communication with computation

- In this question, we should capitalize on LR exchange since we need Q = 1 and keeping the sequential exchange part minimal we update on the rows. We take the parameters as `M = 1000000, N = NP`
- The performance model would be affected by

$$T_{comm} = 4 * (t_s + t_w) + \text{lesser time in previous question's sends and receives}$$

  In the best case scenario (the 4 is to highlight the 4 corners that I have sent before sending the messages left and right)

5.1 Results

`M = 100000 N = NP reps = 100`

Table 7: Performance comparision between normal and overlapping communication(4 Nodes)

NP	$\mathbf{A_T}$	$\mathbf{G_F}$	$\mathbf{P_C}$	$\mathbf{A_T}$ **(-o)**	$\mathbf{G_F}$	$\mathbf{P_C}$	**Speedup**
48	3.78e-01s	1.02e+02	2.12e+00	3.60e-01	1.07e+02	2.22e+00	1.05
96	1.64e-01s	2.34e+02	2.44e+00	1.21e-01	3.17e+02	3.31e+00	1.355
192	5.33e-02s	7.21e+02	3.75e+00	3.95e-02	9.72e+02	5.06e+00	1.349

- For large number of processes with high left and right halo exchange, it acts as an optimization layer and it works pretty nice (with a `1.34` speedup in 192 processes).
- Achieving overlap for 2D communication is difficult because the left-right halo exchange is dependent on top bottom halo exchange corners - I clarified a doubt on this with a diagram in Piazza(link). So synchronizing it with the number of requests to wait for is difficult (with additional checks for P=1 or Q = 1). However, I implemeted this to work for 2D process grids in optimization part of the assignment and compared it's model too with Gadi in Q9

6 Wide halo transfers

6.1 Gracefully exiting on $w > m$ || $w > n$

- For `Q or P > 1` Since the ranks which touch the right and bottom of the field may not satisfy the condition of `w > M_loc || w > N_loc` (because tho blocks size may be smaller there during division of work), normally checking this wouldn't work for all ranks.
- However, we know that for `rank == 0`, if this condition holds true then all the blocks are in danger.
- Hence, we broadcast the value to exit in this scenario
- I also changed the return type of `checkHaloSize()` to int so that all the processes could gracefully exit from `main()`

```
if (rank == 0) {
  if (w > M_loc || w > N_loc) {
    halo_error = 1;
  }
}
MPI_Bcast(&halo_error, 1, MPI_INT, 0, MPI_COMM_WORLD);
if (halo_error == 1) {
  if (rank == 0) {
    printf("%d: w=%d too large for %dx%d local field! Exiting...\n",
           rank, w, M_loc, N_loc);
  }
  return -1; // Could have used exit(0); here but didn't to gracefully exit from main
}
```

6.2 Performance Model

- **Parallel communication**: With increased size of w, 2 * w extra rows and columns are sent

$$\begin{aligned} T_{comm} &= T_{top/bottom} + T_{left/right} \\ &= 4(t_s + (\frac{N}{Q} + 2w)t_w) + 4(t_s + (\frac{M}{P} + 2w)t_w) \\ &= 8t_s + 2(\frac{M}{P} + \frac{N}{Q}).w.t_w \end{aligned}$$

- **Sequential computation** - The inner part updates w times and the 4 edges also update w times with different sizes of `n + 2w -2, n + 2w -4, ... n`

$$\begin{aligned} T_{seq} &= t_{updates} + t_{copy} \\ &= 3 * [\frac{(M+2w-2)(N+2w-2).t_f}{PQ} + \frac{(M+2w-4)(N+2w-4).t_f}{PQ} + \ldots.\frac{MN.t_f}{PQ}] \\ &= 3 * \frac{MN.t_f}{PQ} + O(w(M+N)) \end{aligned}$$

- Total time (when `r%w == 0`)

$$T_{tot} = \frac{r}{w}.2(8t_s + (\frac{M}{P} + \frac{N}{Q}).w.t_w + 3 * \frac{MN.t_f}{PQ} + O(w(M+N)))$$

6.3 Implementation of wide halo technique (discussing the impact on performance)

- A wide halo technique would be useful when it's applied in conjunction with overlapping (since you need to do some extra computation) or when a large amount of data can be stored and used in cache at one point during the sequential updation process. However, this comes at a price of increased `O(w(M + N))` computations (affecting $\mathbf{t_f}$) with increasing values of w in 4 different corners. Functionally and algorithmically as of now it is correct however there are a few bottlenecks or some extra wait calling that I have not noticed.

- One of the bottlenecks and extra overhead that my non-optimized code has is that when executing this section of the code:

```
int reps_left = reps % w;
if (reps_left > 0) {
  // Doing w updates isn't good, the number should be reps_left
  // But won't work
  updateBoundary(u, ldu, w);
  for (w_i = 1; w_i <= reps_left; w_i++) {
    int UR_size = M_loc + (2 * w) - (2 * w_i);
    int UC_size = N_loc + (2 * w) - (2 * w_i);
    updateAdvectField(UR_size, UC_size, &V(u,w_i,w_i), ldu, &V(v,w_i,w_i), ldv);
    copyField(UR_size, UC_size, &V(v,w_i,w_i), ldv, &V(u,w_i,w_i), ldu);
  }
}
```

- Extra updates to the advect field are being taken here which are not needed. To improve this, I tried doing with `updateBoundary(u, ldu, *n_reps*);` but it didn't work because the whole domain of u has been passed. Hence, the code needs to be changed a lot (including other files like `serAdvect` to bring in this functionality. This leads to the fact that `reps % w == 0` would give the optimum performance as of now.

6.4 Results

`M = N = 2000 reps = 100`

Table 8: Performance for various lengths of width for optimized division of 2D grids

w	P	Q	NP	G_R	G_R	P_C	Speedup
1	3	4	12	2.64e-01	3.03e+01	2.53e+00	1
2	3	4	12	2.70e-01	2.97e+01	2.47e+00	0.977
3	3	4	12	2.70e-01	2.97e+01	2.47e+00	0.977
4	3	4	12	2.70e-01	2.97e+01	2.47e+00	0.977
1	16	3	48	2.35e-02	3.40e+02	7.08e+00	1
2	16	3	48	3.01e-02	2.66e+02	5.55e+00	0.780
3	16	3	48	3.14e-02	2.55e+02	5.31e+00	0.748
4	16	3	48	2.99e-02	2.68e+02	5.58e+00	0.785
1	24	8	192	1.37e-02	5.85e+02	3.05e+00	1
2	24	8	192	2.66e-01	3.01e+01	2.51e+00	0.515
3	24	8	192	2.68e-01	2.99e+01	2.49e+00	0.511
4	24	8	192	2.70e-01	2.96e+01	2.47e+00	0.507

- Sample values of tiled stencil with optimum values of P and Q in Q4
- Can suprisingly see increasing trend of time with increase of of w because of the reason above and also because cache locality is not sustained in increasing order of lines. Maybe the implementation is lacking but my assumption is that the following approach is limited.
- The values of w = 2 and 3 are taken from the fact that `reps % 3 != 0` and `reps % 2 == 0` (see the bottleneck point mentioned above)

7 Tiled Stencil Technique

While the processing power of CPUs in stencil computations have been consistently growing, the actual performance of computation is often bottlenecked by the speed at which the processor can access data from the memory[1]. This is a potential problem that we have been facing since the last question. The main goal is that we need to achieve rearranging and grouping the order of execution (or iteration space) while preserving the information flow we need to achieve 2 conclusions:

1. Not violating any sort data dependencies
2. Increasing the segment of data in cache which is needed

The main goal of tiling transformation can be used to improve locality of data, and hence improving cache performance by a huge margin. In our advection solver (where a 9 point stencil has been used) the distance to be traveled between the central to bottom right corner is N_{loc} + 2 * w + 1. Hence as w increases, the probability of the next element in the same cache decreases.
Recent work have shown promise to use modified forms of overlapped tiling to achieve this goal. For eg, a very similar problem of 2D Jacobi, whose 9 point stencil form uses Hierarchical Overlapped tiling[2].It's goal is to adapt to the memory hierarchy of the target machine. It mentions that tiling techniques could improve reuse, with array padding.
Selecting Tile sizes and the process of determining the abstract tiles is even more useful in higher dimension stencils (such as 3D Jacobi iterations) with results in very low cost functions to calculate. It's transformation is reference in [3] with the following code and illustrations (Figure 4 and 7 of [3]):

```
A(N,N,N), B(N,N,N)
    do K=2,N-1
        do J=2,N-1
            do I=2,N-1
                A(I,J,K) = C*(B(I-1,J,K)+B(I+1,J,K)+
                           B(I,J-1,K)+B(I,J+1,K)+
                           B(I,J,K-1)+B(I,J,K+1))
```

Figure 2: Simplified Stencil code

```
do JJ=2,N-1,TJ
    do II=2,N-1,TI
        do K=2,N-1
            do J=JJ,min(JJ+TJ-1,N-1)
                do I=II,min(II+TI-1,N-1)
                    A(I,J,K) = C*(B(I-1,J,K)+B(I+1,J,K) +
                               B(I,J-1,K)+B(I,J+1,K) +
                               B(I,J,K-1)+B(I,J,K+1))
```

Figure 3: Tiled Stencil code for 3D Jacobi iterations

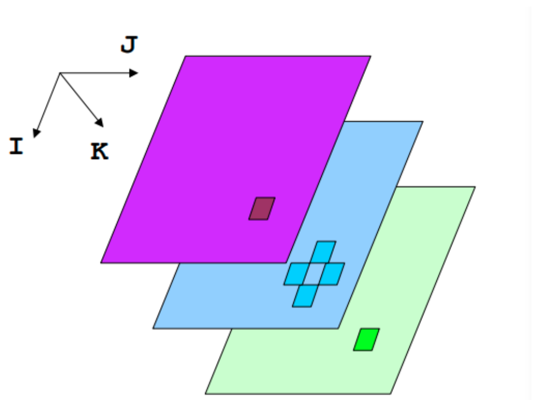

Figure 4: Access pattern of normal 3D Jacobi

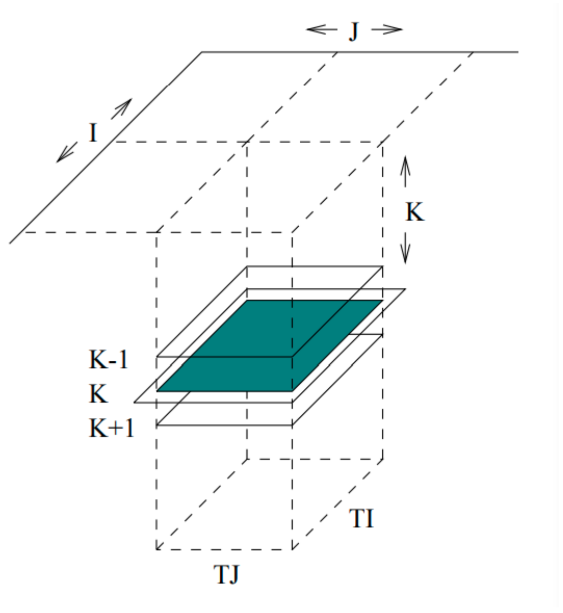

Figure 5: Access Pattern of Tiled 3D Jacobi

8 Combination of Wide halo and stencil technique

A combination of these might alleviate the `O(w(M+N))` extra computations to a certain extent by having good cache coherency. I would hypothesize that it would work for small values of w since it has a space complexity of `O((w + M)(w + N))` thus the memory performance would be detrimented as w increases.

9 Extra optimizations

- A potential optimization was seen when I saw an opportunity to explore Q5 further to work with values of $Q > 1$. For that, I also referenced my doubts and got it clarified on Piazza (link).
- The algorithm works to overlap both row and column exchange cost with computation cost in the case that
  1. The left and right exchange only happens when the values of the corners of the top and bottom halos are exchanged
  2. The final updation of the inner halo only happens after receiving all the messages.

9.1 Results

`M = N = 2000 reps = 100` We see a Similar performance to Q4 with it being slightly slow at some points

Table 9: Performance for optimized division of 2D grids and overlapping

P	**Q**	**NP**	$\mathbf{A_T}$	$\mathbf{G_R}$	$\mathbf{P_C}$	**Speedup**
16	3	48	2.37e-02	3.38e+02	7.05e+00	0.99156118
16	6	96	1.69e-02	4.68e+02	4.81e+00	1.0236686
16	12	192	1.44e-02	5.57e+02	2.87e+00	1.0069444
24	8	192	1.33e-02	5.97e+02	3.11e+00	1.0300752

(but and improved version of Q5 nonetheless). I would have liked to know how I could have improved this model w.r.t the part on exchanging corners.

10 References

[1] Zhou, T., 2016. Factors Affecting Stencil Code Performance. [online] Scholarship.tricolib.brynmawr.edu. Available at: `https://scholarship.tricolib.brynmawr.edu/bitstream/handle/10066/18674/2016ZhouT.pdf?sequence=1&isAllowed=y` [Accessed 26 April 2021].

[2] Holewinski, J., 2021. High-Performance Code Generation for Stencil Computations on GPU Architectures. [online] `http://web.cs.ucla.edu/~pouchet/doc/ics-article.12.pdf`. Available at: `http://web.cs.ucla.edu/~pouchet/doc/ics-article.12.pdf` [Accessed 26 April 2021].

[3] Rivera, G. and Tseng, C., 2000. Tiling optimizations for 3D scientific computations | Proceedings of the 2000 ACM/IEEE conference on Supercomputing. [online] Dl.acm.org. Available at: `https://dl.acm.org/doi/10.5555/370049.370403` [Accessed 26 April 2021].

11 Acknowledgements

- Done individually and with help from Piazza/COMP4300 Practicals.

- One thing which could be improved was providing some sort of visualization to the proposed question which asks the algorithm to be implemented. Rather than being asked in Piazza (like the examples of Q5 and Q6), which would save the student so much time in understanding and executing the correct algorithm.