Student Name: Jeremiah Hsing
Student ID: u7439274

COMP4300 Parallel Systems

Assignment 1

1. Task 1 Deadlock issues

   (a) **What value of N does it deadlock?**

   The deadlock happens when $N >= 32768$.

   Methodology I used to get this N: binary search
   We can first give a guess to N's upper and lower boundaries.

   i. N_{lower}: lower boundary of N.
   ii. N_{upper}: upper boundary of N.

   After I run the code with $N = 10000$ in which deadlock doesn't happen and $N = 100000$ where deadlock happens; we can say that $N_{lower} = 10000$ and $N_{upper} = 100000$.

```
//binary search to get N pesudo code
N_lower = 10000;
N_upper = 100000;
while (N_upper - N_lower > 1) {
  N = (N_upper + N_lower) / 2;
  if (deadlock happens) {
    N_upper = N;
  } else {
    N_lower = N;
  }
}
```

   After this binary search, I find that $N >= 32768$ where deadlock happens on Gadi login node.

   (b) **Explaination of deadlock:**

   In this given code, it uses **MPI_Send()** and **MPI_Recv()** to send and receive the halo data. The **MPI_Send()** and **MPI_Recv()** have blocking semantics.
   This means that when the message size is large enough (that it cannot take the advantage of inner buffer anymore) **MPI_Send()** will not return until the message data has been copied by the receiving process,
   When the $Q = 1$ and $P = np$, as the N becomes bigger, the size of halo messages that need passed and communicate between processes will become larger and larger.

   We can have a look at the original code snips that update the boundary (when $P > 1$) in the **updateBoundar()** function.

```c
static void updateBoundary(double *u, int ldu) {
  ... ...
  if (P == 1) {
    ... ...
  } else {
    //Here is the snips we are talking about
    int topProc = (rank + 1) % nprocs, botProc = (rank - 1 + nprocs) % nprocs;
    MPI_Send(&V(u, M_loc, 1), N_loc, MPI_DOUBLE, topProc, HALO_TAG, comm);
    MPI_Recv(&V(u, 0, 1), N_loc, MPI_DOUBLE, botProc, HALO_TAG, comm,
        MPI_STATUS_IGNORE);
    MPI_Send(&V(u, 1, 1), N_loc, MPI_DOUBLE, botProc, HALO_TAG, comm);
    MPI_Recv(&V(u, M_loc+1, 1), N_loc, MPI_DOUBLE, topProc, HALO_TAG,
        comm, MPI_STATUS_IGNORE);
    // snips end
  }
  ... ...
} //updateBoundary()
```

It's easier that we can use a simple example to figure out the reason for deadlock:
Assume that we have 3 processes, they are P_0, P_1, and P_2.
When the N becomes large enough that cannot use the inner buffer for message passing ($N = N_{large}$)
The P_0 sends message to P_1;
and P_1 sends message to P_2;
and P_2 sends message to P_0

Then due to large N and the blocking semantics as I mentioned above, all the processes are halted and waiting for the message to be received.
The P_0 waits for P_1 (P_1 is halted to wait for P_2);
and P_1 waits for P_2 (P_2 is halted to wait for P_0);
and P_2 waits for P_0 (P_0 is halted to wait for P_1)
So we can see that the processes are in a deadlock state.

And actually, when N is large enough if there is more than one process ($P > 1$) the given code will cause the dead waiting loop that every process is halted to wait for other one to move the first step first (which never happens)

(c) fix the halo-exchange code in **parAdvect.c**:

Methodology:

We can use the *rank* of each process to divide two groups: *rank* is odd and *rank* is even, and we let 2 groups perform message-passing communication operations in a different order to make sure everyone won't be blocked at the same time.
For the even rank processes ($rank\%2 == 0$) they send message to the **topProc**, receive message from **botProc**; send message to the **botProc** then receive message from **topProc**;
For the odd rank processes ($rank\%2 == 1$) they receive message from **topProc**, send message to **botProc**; receive message from **botProc** then send message to **topProc**;

Here is the code of new **updateBoundary()** function:

```
static void updateBoundary(double *u, int ldu) {
  int i, j;

  //top and bottom halo
  //note: we get the left/right neighbour's corner elements from each end
  if (P == 1) {
    for (j = 1; j < N_loc+1; j++) {
      V(u, 0, j) = V(u, M_loc, j);
      V(u, M_loc+1, j) = V(u, 1, j);
    }
  } else {
    int topProc = (rank + 1) % nprocs, botProc = (rank - 1 + nprocs) % nprocs;
    //>>>>>>>>  there I replaced the original code snips  <<<<<<<<
    if (rank % 2 == 0){
    MPI_Send(&V(u, M_loc, 1), N_loc, MPI_DOUBLE, topProc, HALO_TAG, comm);
    MPI_Recv(&V(u, 0, 1), N_loc, MPI_DOUBLE, botProc, HALO_TAG, comm,
      MPI_STATUS_IGNORE);
    MPI_Send(&V(u, 1, 1), N_loc, MPI_DOUBLE, botProc, HALO_TAG, comm);
    MPI_Recv(&V(u, M_loc+1, 1), N_loc, MPI_DOUBLE, topProc, HALO_TAG,
      comm, MPI_STATUS_IGNORE);
    } else {
      MPI_Recv(&V(u, 0, 1), N_loc, MPI_DOUBLE, botProc, HALO_TAG, comm,
        MPI_STATUS_IGNORE);
      MPI_Send(&V(u, M_loc, 1), N_loc, MPI_DOUBLE, topProc, HALO_TAG, comm);
      MPI_Recv(&V(u, M_loc+1, 1), N_loc, MPI_DOUBLE, topProc, HALO_TAG,
        comm, MPI_STATUS_IGNORE);
      MPI_Send(&V(u, 1, 1), N_loc, MPI_DOUBLE, botProc, HALO_TAG, comm);
    }
    //>>>>>>>>  replacement end  <<<<<<<<
  }

  // left and right sides of halo
  if (Q == 1) {
    for (i = 0; i < M_loc+2; i++) {
      V(u, i, 0) = V(u, i, N_loc);
      V(u, i, N_loc+1) = V(u, i, 1);
    }
  } else {
  }
} //updateBoundary()
```

2. Task 2 The effect of non-blocking communication

    (a) Update **updateBoundary()** with unblocking method:

```
static void updateBoundary(double *u, int ldu) {
  int i, j;
  //top and bottom halo
  //note: we get the left/right neighbour's corner elements from each end
  if (P == 1) {
    for (j = 1; j < N_loc+1; j++) {
      V(u, 0, j) = V(u, M_loc, j);
      V(u, M_loc+1, j) = V(u, 1, j);
    }
```

```
    } else {
      int topProc = (rank + 1) % nprocs, botProc = (rank - 1 + nprocs) % nprocs;
      //Task_2 solution start
      MPI_Request req[4];
      MPI_Status stat[4];
      MPI_Isend(&V(u, M_loc, 1), N_loc, MPI_DOUBLE, topProc,
        HALO_TAG, comm, &req[0]);
      MPI_Irecv(&V(u, 0, 1), N_loc, MPI_DOUBLE, botProc,
        HALO_TAG, comm, &req[1]);
      MPI_Isend(&V(u, 1, 1), N_loc, MPI_DOUBLE, botProc,
        HALO_TAG, comm, &req[2]);
      MPI_Irecv(&V(u, M_loc+1, 1), N_loc, MPI_DOUBLE, topProc,
        HALO_TAG, comm, &req[3]);
      MPI_Waitall(4, req, stat);
      //Task_2 solution end
    }
    // left and right sides of halo
    if (Q == 1) {
      for (i = 0; i < M_loc+2; i++) {
        V(u, i, 0) = V(u, i, N_loc);
        V(u, i, N_loc+1) = V(u, i, 1);
      }
    } else {

    }
  } //updateBoundary()
```

(b) Compare two methods:
Here is the running time (unit: seconds) two methods take for $r = 100$, $M = 10000$, $N = 10000$ with np as [1, 3, 6, 12, 24, 48, 96, 192]

Table 1: Blocking and Unblocking Comparation

np	1	3	6	12	24	48	96	192
blocking	29.5	12.2	9.09	8.56	4.3	2.15	1.05	0.495
unblocking	29.3	12.2	9.09	8.54	4.3	2.14	1.05	0.495

We can say that the unblocking method is slightly better than the blocking method.

3. Task 3 Performance modeling and calibration

(a) Preparation: first we need to write some measuring code to measure the t_s, t_w, and t_f:

i. t_s is the communication startup time, we can set the message size that we send in the measuring program to be 0 so that we can defuse the influence of message size in communication and let the startup time be the significant factor.
Here is the measuring program for t_s:

```
// measure t_s pseudo code

int main(int argc, char *argv[])
{
  int rank;
  double communication_time = 0;
```

```
    int msg_size = 0;
    int reps = 1000;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0)
    {
      MPI_Barrier(MPI_COMM_WORLD);
      for (int i = 0; i < reps; i++)
      {
        start_time = MPI_Wtime();
        MPI_Send(NULL, msg_size, 1);
        MPI_Recv(NULL, msg_size, 1);
        end_time = MPI_Wtime();
        communication_time += end_time - start_time;
      }
    }
    else if (rank == 1)
    {
      MPI_Barrier(MPI_COMM_WORLD);
      for (int i = 0; i < reps; i++)
      {
        start_time = MPI_Wtime();
        MPI_Recv(NULL, msg_size, 0);
        MPI_Send(NULL, msg_size, 0);
        end_time = MPI_Wtime();
        communication_time += end_time - start_time;
      }
    }
    printf("Communication startup time: %.3e seconds (rank %d)\n",
      communication_time / reps / 2, rank);
    return 0;
  }
```

ii. t_w is the communication per-word time, we can set the message size that we send in the measuring program to be significantly large (such as 1000000 * **sizeof(double)**) so that we can defuse the influence of startup time in communication and let the per-word time be the significant factor.
Here is the measuring program for t_w:

```
  // measure tw pseudo code

  int main()
  {
    int rank, size, tag = 0;
    int msg_size = 1000000;
    double communication_time;
    MPI_Request req;
    MPI_Init();
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    double *send_buffer = (double *)malloc(msg_size * sizeof(double));
    double *recv_buffer = (double *)malloc(msg_size * sizeof(double));

    if (rank == 0)
    {
```

```c
    MPI_Barrier(MPI_COMM_WORLD);
    communication_time = MPI_Wtime();
    MPI_Send(send_buffer, msg_size, 1);
    MPI_Recv(recv_buffer, msg_size, 1);
    communication_time = MPI_Wtime() - communication_time;
  }
  else if (rank == 1)
  {
    MPI_Barrier(MPI_COMM_WORLD);
    communication_time = MPI_Wtime();
    MPI_Recv(recv_buffer, msg_size, 0);
    MPI_Send(send_buffer, msg_size, 0);
    communication_time = MPI_Wtime() - communication_time;
  }
  free(send_buffer);
  free(recv_buffer);
  printf("Communication per-word time: %.3e seconds (rank %d)\n",
    communication_time / msg_size / 2, rank);
  return 0;
}
```

iii. t_f is the float computation time, here is the measuring program:

```c
// measure the float computation time pseudo code

int main()
{
  double num = 2.76;
  for (int i = 0; i < reps; i++)
  {
    start_time = MPI_Wtime();
    num = num * num;
    num = num + num;
    num = num - num;
    num = num + num;
    end_time = MPI_Wtime();
    computation_time += end_time - start_time;
  }
  printf("Float computation time: %.3e seconds \n",
    computation_time / 4 / reps);
  return 0;
}
```

After running the measuring program, we can get this output:

```
mpirun -np 2 ./measure_ts
Communication startup time: 6.911e-07 seconds (rank 0)
Communication startup time: 7.414e-07 seconds (rank 1)

mpirun -np 2 ./measure_tw
Communication per-word time: 1.533e-09 seconds (rank 1)
Communication per-word time: 1.533e-09 seconds (rank 0)

mpirun -np 1 ./measure_tf
```

```
Float computation time: 7.602e-10 seconds
```

We get that(rounding results):
$t_s = 6.911 \times 10^{-7}$ (second per startup)
$t_w = 1.533 \times 10^{-9}$ (second per double type message)
$t_f = 7.602 \times 10^{-10}$ (second per float computation)

(b) write a performance model for the computation:

$$T_{para} = r \times (T_{comm} + T_{comp}) \tag{1}$$

$$T_{para} = r \times (4 \times t_s + 4 \times N \times t_w + 20 \times \frac{M \times N}{P} \times t_f) \tag{2}$$

Now let me explain how we can get the equations above:

By reading the code in **parAdvect.c**, we get that the total time T_{para} contains two parts: communication time T_{comm} and computation time T_{comp}.
For the communication time T_{comm}, we can get that:

i. Each process doing 4 message passing communication operations:
   A. **Send(msg, topProc)**
   B. **Recv(msg, topProc)**
   C. **Send(msg, botProc)**
   D. **Recv(msg, botProc)**

   So the coifficient of t_s is 4

ii. The communication per-word time t_w is the same for all the processes. And each process has 4 message passing operations (as I show above).
   for a $1D$ grid module that we are now analyzing, each message's size is **N * sizeof(double)**; which is .
   So we can get that t_w's coifficient is $4 \times N$

iii. The computation time T_{comp}:
   According to the code in **updateAdvectField()**, for one cell it needs to do 20 float computation.
   And there are $M \times N$ cells in the whole grid, which are divided into P processes to compute parallelly(for a 1D grid)
   So the coifficient of t_f is $20 \times \frac{M \times N}{P}$

iv. There are r iterations so the total time $T_{para} = r \times (T_{comm} + T_{comp})$

(c) Run experiments to determine the values of these parameters on Gadi, and justify my methodology:

i. First we let $M = N = 1000, r = 100, np = 192$
   Then we can get the result as follows:

```
mpirun -np 192 ./testAdvect 1000 1000 100
Advection of a 1000x1000 global field over 192x1 processes for 100 steps.
Advection time 8.69e-03s, GFLOPs rate=2.30e+02 (per core 1.20e+00)
```

Using the equation above and the values we measured we can get that:
T_{para}
$= 100 \times (4 \times 6.911 \times 10^{-7} + 4 \times 1000 \times 1.533 \times 10^{-9} + 20 \times \frac{1000\times1000}{192} \times 7.602 \times 10^{-10})$
$= 8.88083 \times 10^{-3}$
$\approx 8.69 \times 10^{-3}s$

ii. Verify t_s's coifficient methodology:
We can use a very small M and N (such as $M = N = 2$) to significantly reduce the computation and message passing workload; and use the executing time to measure the communication startup time.
In this case we use $np = 2$, $M = N = 2$ and $r = 100$; and then we can say that $T_{para} \approx 4 \times r \times t_s = 400t_s$
We can get the result as follows:

```
\$ mpirun -np 2 ./testAdvect 2 2 100
Advection of a 2x2 global field over 2x1 processes for 100 steps.
Advection time 3.20e-04s, GFLOPs rate=2.50e-02 (per core 1.25e-02)
```

The t_s we measure is 6.911×10^{-7}
$400 \times t_s = 400 \times 6.911 \times 10^{-7} = 2.7644 \times 10^{-4} \approx 3.2 \times 10^{-4}$
So we can say that the t_s value and its coefficient are correct.

iii. Verify t_f's coifficient methodology:
We can use a large M and N (large but still keep the field in L3 Cache Lake size, such as $M = N = 1000$); we keep$M$, N, r and we use different np to run the program. As the $P = np$ in the 1D grid module, we can compare the difference of running time to verify that if $\frac{1}{P}$ is t_f's coefficient.
We can get the result as follows:

```
mpirun -np 96 ./testAdvect 1000 1000 100
Advection of a 1000x1000 global field over 96x1 processes for 100 steps.
Advection time 1.14e-02s, GFLOPs rate=2.01e+02 (per core 2.09e+00)

mpirun -np 192 ./testAdvect 1000 1000 100
Advection of a 1000x1000 global field over 192x1 processes for 100 steps.
Advection time 8.69e-03s, GFLOPs rate=2.23e+02 (per core 1.16e+00)
```

According to the experiments, $T_{np=96} - T_{np=192} = 2.71 \times 10^{-3}s$
According to the equation above $T_{np=96} - T_{np=192} = 100 \times 20 \times 1000 \times 1000 \times t_f \times (\frac{1}{96} - \frac{1}{192}) = 2.71 \times 10^{-3}$
then we can get that $t_f = 2.71 \times 10^{-3} \times \frac{1}{100\times20\times1000\times1000\times(\frac{1}{96}-\frac{1}{192})} = 2.6016 \times 10^{-10} \approx 7.602 \times 10^{-10}$

(d) Within one Gadi node, perform a strong scaling analysis and compare predicted vs actual execution time for various numbers of processes p Use the same value of M, N, and r throughout. Justify why you think those values are suitable:
strong scaling: the process number is increasing while the problem size is fixed.
$S_p = \frac{T_{seq}}{T_{para}}$
$= \frac{4t_s+4N\times t_w+20MN\times t_f}{4t_s+4N\times t_w+\frac{4t_s+4N\times t_w+20MN\times t_f}{P}}$

$\lim_{P\to\infty} S_p$
$= \frac{4t_s+4N\times t_w+20MN\times t_f}{4t_s+4N\times t_w}$
$= 1 + \frac{20MN\times t_f}{4t_s+4N\times t_w}$

Let $M = N = 1000, r = 100, np = [1, 3, 6, 12, 24, 48]$
$t_s = 6.911 \times 10^{-7}$ (second per startup)
$t_w = 1.533 \times 10^{-9}$ (second per double type message)
$t_f = 7.602 \times 10^{-10}$ (second per float computation)
Using the M, N value and the t_s, t_w and t_f we measured above, we can get the S_p value when $P \to \infty$:
$\lim_{P\to\infty} S_p = 1 + \frac{20MN\times t_f}{4t_s+4N\times t_w} \approx 24.327$

The experiment result of the running time of different np in one node is as follows:

Table 2: Running time of different np in one node

np	1	3	6	12	24	48
time (sec)	2.29e-01s	9.10e-02s	4.21e-02s	2.01e-02s	1.10e-02s	1.21e-02s

We can see that at first the running reduced when np increasing; however, when $np > 24$ the running time keeps almost the same(even increase a little for $np = 48$ and $np = 24$).
That's because when np is large enough the parallel part (computation part) becomes a much less significant factor that affects the running time. In this case, we can say that when $np = 24$ the speed up is close to the upper boundary of speed up.

The speed-up of $np = 24$ is:
$S_{24} = \frac{T_{np=1}}{T_{np=24}} = \frac{2.29e-01}{1.10e-02} = 20$ this is also close to the theretical value $S_\infty = 24.327$ which we get from the strong scaling analysis above.

4. Task 4 The effect of 2D process grids

   (a) Update **updateBoundary()** with 2D process grids ($Q >= 1$):

   Code has been updated in **updateBoundary()** in file **parAdvect.c**.

   (b) Conduct experiments with fixed $M = N = 1000, r = 100, np = 36$ on one node:
   experiment output:

Table 3: task 4 experiment 1 output

P × Q	1 × 36	3 × 12	4 × 9	6 × 6	12 × 3	18 × 2	36 × 1
time (sec)	1.98e-02s	6.84e-03s	7.91e-03s	6.36e-03s	5.99e-03s	7.21e-03s	1.08e-02s

   i. whether a (near) square ratio has a different effect to the default ($Q = 1$)?
   Yes, when we keep the np and use a near square ratio($P = Q$), the running time is much less than the default ($Q = 1$).

ii. What is the optimum aspect ratio predicted by my model?

A. The new performance model is:

$$T_{para} = r \times (8 \times t_s + 4 \times (N_{loc} + M_{loc} + 2) \times t_w + 20 \times \frac{M \times N}{P \times Q} \times t_f)$$
$$T_{para} = r \times (8 \times t_s + 4 \times (\frac{N}{Q} + \frac{M}{P} + 2) \times t_w + 20 \times \frac{M \times N}{P \times Q} \times t_f)$$

where M_{loc}, N_{loc} are local field length and width within a grid.

B. The optimum aspect ratio:

If we don't consider any other factors and assume that accessing continuous elements in a 2D array takes the same time as accessing non-continuous elements (which is not usually what it is), then when $(M = N)$ the optimum aspect ratio is: when $P = Q$

Explanation:
According to AM-GM Inequality: $M_{loc} + N_{loc} \geq \sqrt{M_{loc} \times N_{loc}}$ (equal iif $M_{loc} = N_{loc}$). This means when $M = N, r, np$ keep the same, if $P = Q$, the $M_{loc} = N_{loc}$, and each grid is a square; the $(M_{loc} + N_{loc})$ is smaller than that of $P \neq Q$, so the coefficients of t_w in $P = Q$ is smaller, so it's faster.

The experiment output explanation:
However, as we can see, the experiment output above doesn't show that $P \times Q = 12 \times 3$ is faster than $P \times Q = 6 \times 6$. And after repeating the experiment several times the outputs are basically the same. So it's not an accident error.
Why does the real-world result diverge from our theoretical result?
This is because, the compiler can take the advantage of locality and in a 2D array, the accessing time for continuous elements is usually faster than accessing non-continuous elements. $P \times Q = 12 \times 3$ grid has more continuous elements (row longer than column) than $P \times Q = 6 \times 6$ grid, which makes it run faster.

As a result, we can give an actual optimum aspect ratio for the real-world case, which needs two factors to be considered (1) the sidelength of the grid; (2)continuous elements percentage in the halo (locality advantage). We need to figure out what factor is dominating and get the real result.
However, if we don't consider factor (2), then $P = Q$ is the optimum aspect ratio.

iii. Predict how much difference there would be if the coefficient t_w was 10 times larger.
The difference of the new module and the old module when coefficients of t_w 10 times larger is:

$$T_{para}^{old} = r \times (4 \times t_s + 4 \times 10 \times N \times t_w + 20 \times \frac{M \times N}{P} \times t_f)$$
$$T_{para} = r \times (8 \times t_s + 4 \times 10 \times (\frac{N}{Q} + \frac{M}{P'} + 2) \times t_w + 20 \times \frac{M \times N}{P' \times Q} \times t_f)$$
$$P = P' \times Q$$
$$\Delta T_{para} = T_{para}^{old} - T_{para}$$
$$\Delta T_{para} = 4 \times 10 \times r \times (\frac{N}{Q} + \frac{M}{P'} + 2 - N) \times t_w - 4 \times r \times t_s$$

iv. conduct experiments with fixed $M = N = 1000, r = 100, P = 12$ on 1 to 4 nodes (48 to 192 cores)
experiment output:

Table 4: task 4 experiment 2 output

np	48	96	144	192
time (sec)	5.41e-03s	2.79e-02s	2.97e-02s	2.92e-02s

5. Task 5 Overlapping communication with computation

(a) Update **parAdvectOverlap()** using overlapping communication

Code has been updated in **parAdvectOverlap()** in file **parAdvect.c**
Also add two helper functions in file **parAdvect.c**:

```
static void overlapUpdateBoundaryTB(double *u, int ldu, MPI_Request *req);
static void overlapUpdateBoundaryLR(double *u, int ldu);
```

(b) Discuss what the performance impact of this technique might be, and describe how it would affect your performance model.

i. Using the overlapping technique, the message passing time is overlapped with the computation time, so the total running time is less than the time without overlapping.

ii. New performance model is: (when $Q = 1$)

$$T_{para} = r(4t_s + \mathbf{Max(}4Nt_w \text{ , } \frac{20(M-2)(N-2)}{P}t_f\mathbf{)} + 20(\frac{2M}{P} + 2N - 4)t_f)$$

(c) conduct experiments with fixed $M = N = 1000, r = 100, np = 4, P = 2$
experiment output:

(d) Run experiments
conduct experiments with fixed $M = N = 1000, r = 100$ on 1 to 4 nodes (48 to 192 cores) using the overlapping and 1D grid($P = np$)
experiment output:

Table 5: task 4 experiment 2 output

opts	None	"-o"
time (sec)	6.78e-02s	6.54e-02s

Table 6: task 4 experiment 3 output

np	48	96	144	192
time (sec)	1.24e-02s	1.42e-02s	3.32e-02s	9.61e-03s

(e) Why achieving overlap for 2D communication is difficult?

Explanation:
In 2D grid communication, the left & right direction message passing has data dependency on the top & bottom direction message passing. This means in a 2D grid module, the left & right direction message passing operations must wait til top & bottom direction message passing is finished;
so the left & right direction message passing operations cannot be overlapped. And this makes achieving overlap for 2D communication difficult.

6. Task 6 Wide halo transfers

(a) Update **parAdvectWide()** using wide halo

Code has been updated in **parAdvectWide()** in file **parAdvect.c**.
Update the **checkHalo()** function as requested.
Also, add a helper function in file **parAdvect.c**:

```
static void wideUpdateBoundary(double *u, int ldu, int w);
```

(b) What are the potential advantages of wide halos?

The potential advantages of wide halos are:
By using wide halo, the number of communication has been reduced to $\frac{r}{w}$, this will highly reduce the communication startup time; and according to the data we measure above the communication startup t_s is significantly larger than t_w and t_f.

(c) How it would affect your performance models?

$$T_{para} = \frac{8rt_s}{w} + 4r(\frac{N}{Q} + \frac{M}{P} + 2)t_w + \sum_{i=0}^{w-1} \frac{20r}{w}(\frac{M}{P} + 2i)(\frac{N}{Q} + 2i)t_f$$

(d) conduct experiments with fixed $M = N = 1000, r = 100, np = 4, P = 2$
experiment output:

(e) Run appropriate experiments
conduct experiments with fixed $M = N = 1000, r = 100, P = 12, w = 5$ on 1 to 4 nodes (48 to

Table 7: task 4 experiment 2 output

opts	None	"-w 2"
time (sec)	6.78e-02s	6.59e-02s

192 cores)
experiment output:

Table 8: task 4 experiment 3 output

np	48	96	144	192
time (sec)	6.19e-03s	2.91e-02s	2.96e-02s	1.85e-02s

7. Task 7 Literature Review (optimization techniques for stencil computations)

   Stencil computation is a kind of iterative algorithm that is widely used in scientific computing, such as solving partial differential equations (PDE). It involves updating array elements based on the values of their neighbors, usually in a regular pattern. The performance of stencil computation can be affected by factors like memory system performance. One approach to address this is the tiled stencil technique. The tiled stencil technique is an optimization technique that aims to improve the performance of stencil computation by enhancing cache locality and reducing memory access overhead. However, there are still lots of aspects that can be improved in the tiled stencil computation. In this literature review, we will focus on the implementations based on the primary tiled stencil computation.

   Current automatic tiling frameworks frequently select hyperplanes that result in pipelined initialization and uneven workload distribution. Driven by this consideration, researchers have employed a novel approach called diamond tiling[1], which aims to enable concurrent start-up initialization and achieve optimal workload balance whenever feasible. According to their experiments outputs, the new technique they applied can outperform previous methods by a factor of 1.3 to 10.0 (Bondhugula et al., 2017).

   Usually, stencil computation needs to access a lot of memory, so in some processors like Intel Xeon Phi x200 special high-speed memory called high-bandwidth memory (HBM) to improve the speed of memory system access. However, when handling computation with a large problem size that is over the HBM size limitation, the processor will have to use regular bandwidth memory instead and cannot take the advantage of the HBM. To address this problem and make as full use of the hardware resource as possible, a new method called temporal wave-front tiling is driven by researchers[2]. This method introduces an extra layer of cache-blocking to enable the effective utilization of both the HBM bandwidth and the memory capacity. According to their experiments outputs, it has been shown that temporal wave-front tiling can offer a 2.4x speed increase when compared to using HBM cache without temporal tiling, and a 3.3x speed boost compared to relying solely on regular memory for solving large problem sizes (Yount & Duran, 2016).

8. Task 8 Performance outcome via a combination of optimization techniques

   Combining diamond tiling and temporal wave-front tiling techniques may alleviate the limitations imposed by memory system performance in stencil computations.

Diamond tiling is designed to enable concurrent start-up initialization and achieve optimal workload balance. This technique enhances cache locality, reduces memory access overhead, and improves overall computation efficiency. By optimizing cache utilization and evenly distributing the workload, diamond tiling can significantly speed up stencil computations.

Temporal wave-front tiling, on the other hand, addresses the problem of handling large problem sizes that exceed the capacity of high-bandwidth memory (HBM). It introduces an extra layer of cache-blocking to enable effective utilization of both HBM bandwidth and memory capacity. This technique allows for better handling of large problem sizes and further improves the performance of stencil computations.

The combination of diamond tiling and temporal wave-front tiling could potentially improve memory system performance by exploiting cache locality, workload balance, and efficient use of HBM bandwidth and memory capacity.

However, there are some trade-offs to consider when implementing these techniques together:

(a) Implementation Complexity: Combining these two techniques might increase the complexity of the implementation, which may require more development time and resources to effectively integrate both methods.

(b) Additional Overhead: The additional layer of cache-blocking introduced by temporal wave-front tiling may impose extra overhead in the form of increased memory access latency and cache management complexity.

9. Task 9 Implement an optimization technique

(a) Update **parAdvectExtra()** using wide halo

Code has been updated in **parAdvectExtra()** in file **parAdvect.c**.
The optimization is to reduce the number of **copyField()** function calls.

(b) conduct experiments with fixed $M = N = 1000, r = 100, np = 4, P = 2$
experiment output:

Table 9: task 4 experiment 2 output

opts	None	"-x"
time (sec)	6.78e-02s	4.86e-02s

References

[1] Bandishti V. Bondhugula, U. and I. Pananilath. Diamond tiling: Tiling techniques to maximize parallelism for stencil computations. *IEEE Transactions on Parallel and Distributed Systems*, 28:1285–1298, 2017.

[2] C. Yount and A. Duran. Effective use of large high-bandwidth memory caches in HPC stencil computation via temporal wave-front tiling. *2016 7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2016.