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 boundary.

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

After I run the code with $N = 10000$ in which dead lock doesn't happen and $N = 100000$ where dead lock happens; so 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 the 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 becoming 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 more easy that we can use a simple example to figure out the reason of deadlock:
Assume that we have 3 processes, they are P_0, P_1, and P_2.
When the N become 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 give code will cause the dead waiting-loop that every process is halted to wait other one moving 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 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:

```c
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:

```c
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 blocking method.

3. Task 3 Performance modelling and calibration

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)

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)

6. Task 6 Wide halo transfers

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

   Code has been updated in **parAdvectWide()** in file **parAdvect.c**.
   Also add a helper function in file **parAdvect.c**:

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

   (b)

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

8. Task 8 Performance outcome via combination of optimization techniques

9. Task 9 Implement an optimization technique