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.

```
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:

```
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 blocking method.

3. Task 3 Performance modelling 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 the communication and let the startup time be the significant factor.
Here is the measuring program for t_s:

```
    // measure t_s pesudo 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 the communication and let the per-word time be the significant factor.
Here is the measuring program for t_w:

```
// measure tw pesudo 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)
  {
```

```
    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:

```
// measure the float computation time pesudo 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 the 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 commputation 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}
$= 100 \times (4 \times 6.911 \times 10^{-7} + 4 \times 1000 \times 1.533 \times 10^{-9} + 20 \times \frac{1000 \times 1000}{192} \times 7.602 \times 10^{-10})$
$= 8.88083 \times 10^{-3}$
$\approx 8.69 \times 10^{-3} s$

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 $T_{para} \approx 4 \times r \times t_s = 400 t_s$
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 t_s value and its coifficient is 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 1D grid module, we can compare the difference of running time to verify that if $\frac{1}{P}$ is t_f's coifficient.
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 $t_f = 2.71 \times 10^{-3} \times \frac{1}{100 \times 20 \times 1000 \times 1000 \times (\frac{1}{96} - \frac{1}{192})} = 2.6016 \times 10^{-10} \approx 7.602 \times 10^{-10}$

(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 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 effact 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)

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