CAAM 520 – Homework 2

Due: 3/6/2020

General remarks: Please submit your homework report (as a PDF file) along with any code to your personal CAAM 520 Git repository. Please provide a makefile that can be used to compile your code, and make sure that your code compiles and runs in the CAAM 520 virtual machine.

Problem 1 *(Message passing, 15 pts.)*

Assume that the code below is executed using three MPI processes. Determine the value of `x[0]`, `x[1]`, and `x[2]` *on each rank* before and after the completion of each MPI message or broadcast. Determine the result returned by the function *on each rank*. Express your answers in terms of the inputs `a`, `b`, and `c`.

```
int foo(int a, int b, int c)
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  int x[3];
  switch (rank) {
  case 0:
    x[0] = a; x[1] = b; x[2] = c;
    MPI_Bcast(&x[0], 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&x[1], 1, MPI_INT, 1, MPI_COMM_WORLD);
    break;
  case 1:
    x[0] = b; x[1] = c; x[2] = a;
    MPI_Bcast(&x[1], 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Recv(&x[1], 1, MPI_INT, 2, 999, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Bcast(&x[0], 1, MPI_INT, 1, MPI_COMM_WORLD);
    break;
  case 2:
    x[0] = c; x[1] = a; x[2] = b;
    MPI_Bcast(&x[0], 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Send(&x[2], 1, MPI_INT, 1, 999, MPI_COMM_WORLD);
    MPI_Bcast(&x[0], 1, MPI_INT, 1, MPI_COMM_WORLD);
    break;
  }

  int result = 0;
  MPI_Reduce(&x[rank], &result, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
  return result;
}
```

Note: Feel free to compile and run the above code to verify your answers for some values of `a`, `b`, and `c`.

Problem 2 *(MPI collectives, 20 pts.)*

a) Using `MPI_Send()` and `MPI_Recv()`, implement a function

```
void my_bcast(void *buf, int count, MPI_Datatype datatype, MPI_Comm comm)
```

that broadcasts data *from the master rank* to all other ranks. Calling your function should be equivalent to calling `MPI_Bcast()` with the `root` argument equal to zero. *(5 pts.)*

b) Implement a function

```
void my_bcast_tree(void *buf, int count, MPI_Datatype datatype,
                   MPI_Comm comm)
```

that performs the same broadcast as before, but with the following optimization: Instead of calling `MPI_Send()` $p-1$ times on the master rank to broadcast the data to all p MPI ranks, each rank should participate in the broadcast once it has received the data. Specifically, implement the algorithm below, in which the number of ranks that have received the data doubles after each round of communication.

```
while not all ranks have received the data do
  if this rank has received the data then
    send the data to one of the ranks that has not received it yet
  end if
end while
```

(15 pts.)

Note: Implement both functions in the file `bcast.c` which is provided as part of this homework assignment. Do not modify the `main()` function in `bcast.c`. You can compile and run the code in `bcast.c` to test your implementation. In particular, make sure that your implementation works when using a single MPI process and when using a number of processes that is not a power of two.

Problem 3 *(Deadlocks and MPI, 15 pts.)*

a) In your own words, *briefly* describe what a deadlock is and how it can occur when using `MPI_Send()`, `MPI_Recv()` and collectives such as `MPI_Bcast()`. *(5 pts.)*

b) Can the following code result in a deadlock? If so, explain why and under which conditions a deadlock can occur. *(5 pts.)*

```
int rank;
double data;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank%2 == 0) {
  // Even rank, send data to next rank.
  MPI_Send(&data, 1, MPI_DOUBLE, rank + 1, 999, MPI_COMM_WORLD);
}
else {
  // Odd rank, receive data from previous rank.
  MPI_Recv(&data, 1, MPI_DOUBLE, rank - 1,
           999, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

c) Can the following code result in a deadlock? If so, how can the code be modified to avoid the deadlock? *(5 pts.)*

```
int rank;
double data, result;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) {
  // Print result on master rank.
  MPI_Reduce(&data, &result, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
  printf("Result: %f\n", result);
}
```

Note: For part c), it suffices to describe the necessary modifications. You do not have to submit the modified code.

Problem 4 *(The Jacobi method with MPI, 50 pts.)*

In this problem, you will implement the Jacobi method using MPI. Unlike in the first homework, you will not implement the Jacobi method for a general linear system $Ax = b$. Instead, you will implement the Jacobi method for Poisson's equation as in the example `mpi_poisson_jacobi.c` that was discussed in class. Your task is to improve the MPI communication in the algorithm implemented in the example.

a) Review the example `mpi_poisson_jacobi.c`. Recall that it distributes a grid of n^3 points among p MPI ranks by assigning a slice of $n \times n \times n/p$ grid points to each rank, i.e., by slicing the grid along the third direction (the "k-direction"). *(0 pts.)*

b) Familiarize yourself with the code template in the file `jacobi_3d_slicing.c` which is provided as part of this homework assignment. Unlike in the example `mpi_poisson_jacobi.c`, the n^3-point grid is now sliced along *each* direction, resulting in less MPI communication. Specifically, the code template uses the following parameters, data structures, and functions:

- `n` – The number of grid points in each direction
- `npi`, `npj`, `npk` – The number of parts (slices) in i-, j-, and k-direction. Assume that each of these numbers is greater than or equal to two and divides `n`. Furthermore, assume that the number of MPI ranks is `npi`·`npj`·`npk`, i.e., there are as many parts as there are ranks.
- `pi`, `pj`, `pk` – A tuple of three numbers that uniquely identify each part after the slicing. Possible values are `pi=0,...,npi-1`, `pj=0,...,npj-1`, and `pk=0,...,npk-1`.
- `ni_loc`, `nj_loc`, `nk_loc` – As a result of the slicing, each MPI rank works on a cuboidal part of dimensions `ni_loc`×`nj_loc`×`nk_loc`, where `ni_loc=n/npi` etc.
- `u_new`, `u_old`, `f` – *One-dimensional* arrays to store the solution etc. Each array is of length `(ni_loc+2)`·`(nj_loc+2)`·`(nk_loc+2)`, providing enough storage for values at all local grid points as well as for ghost layers at the top, bottom, left, right, front, and back of the cuboidal part of the grid that is stored on each rank.
- `ijk_to_index` – A function that maps *local* indices `i_loc`, `j_loc`, and `k_loc` to an entry in the arrays `u_new`, `u_old`, and `f`. For example, the value of `u_new` at the grid point with *local* coordinates `i_loc`, `j_loc`, and `k_loc` can be accessed using `u_new[ijk_to_index(i_loc, j_loc, k_loc, ni_loc, nj_loc)]`. Notice that besides the coordinates `i_loc=0,...,ni_loc-1`, coordinates of `i_loc=-1` and `i_loc=ni_loc` can be used to access the ghost layers in negative and positive i-direction etc.

Note: To get a better understanding of the different parameters and the data layout, review the figures on the last page of this assignment. *(0 pts.)*

c) The template already contains all code for the computation of the Jacobi updates. All that is left for you to implement is the halo exchange. As a preparation, implement a function

```
int part_to_rank(int pi, int pj, int pk, int npi, int npj, int npk)
```

that returns the rank of the MPI process which works on the part of the grid identified by the coordinates `pi`, `pj`, `pk`. Furthermore, implement the inverse funtion

```
void rank_to_part(int rank, int npi, int npj, int npk,
                  int *pi, int *pj, int *pk)
```

which returns the coordinates `pi`, `pj`, `pk` of the part of the grid assigned to the given rank. *(10 pts.)*

d) Describe how you would implement the halo exchange, i.e., which data each rank must send to and receive from its neighbors, how you would identify the neighbors using the functions implemented in part c), and which MPI functions you would call to communicate the data. *(15 pts.)*

e) Implement the halo exchange in the `jacobi_update()` function. Test your implementation by running

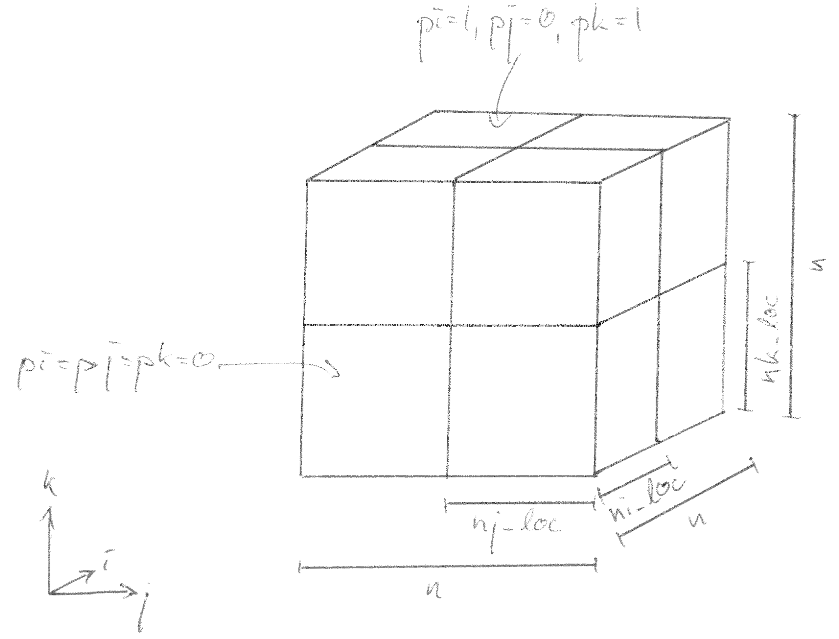
```
mpirun -n 8 ./jacobi_3d_slicing 32 2 2 2 100
```

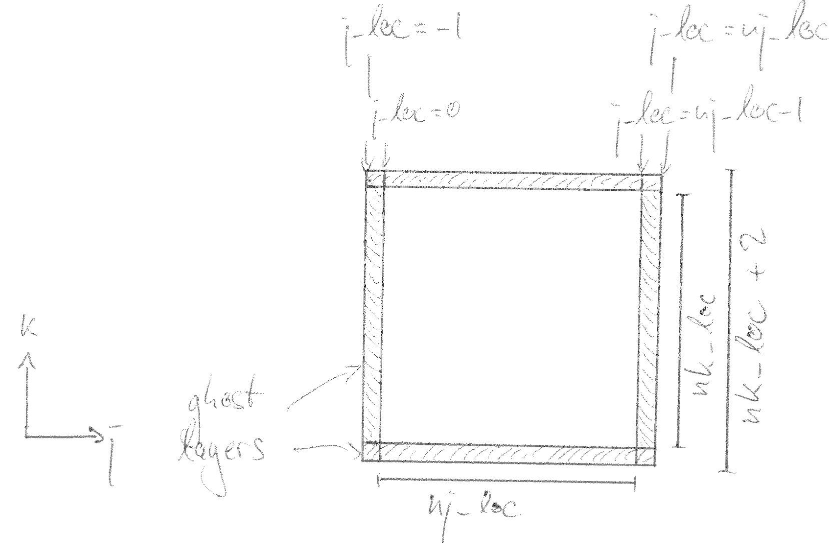
You should obtain a relative error of about $6.35 \cdot 10^{-1}$. Make sure that your code produces the same result for different partitionings, e.g., when running

```
mpirun -n 16 ./jacobi_3d_slicing 32 2 4 2 100
```

etc. *(25 pts.)*

Note: Do not change the code in the `main()` function. Furthermore, do not change the signature (number, types, and names of arguments) of any functions that are part of the code template.


pi=1, pj=0, pk=1
pi=pj=pk=0
n
nk_loc
ni_loc
n
nj_loc
n
k
i
j


j_loc=-1
j_loc=0
j_loc=nj_loc
j_loc=nj_loc-1
nk_loc
nk_loc+2
ghost layers
nj_loc
nj_loc+2
k
j