CAAM 520, Spring 2020 - Problem Set 1

Eappen Nelluvelil

February 21, 2020

1. *Message passing*

   The following are the values of `x[0]`, `x[1]`, and `x[2]` on ranks `0`, `1`, and `2` before and after the completion of each MPI message or broadcast:

   (a) Rank `0`

   i. `MPI_Bcast(&x[0], 1, MPI_INT, 0, MPI_COMM_WORLD);`
   Before this MPI broadcast,
   A. `x[0] = a`,
   B. `x[1] = b`, and
   C. `x[2] = c`.
   After this broadcast,
   A. `x[0] = a`,
   B. `x[1] = b`, and
   C. `x[2] = c`.

   ii. `MPI_Bcast(&x[1], 1, MPI_INT, 1, MPI_COMM_WORLD);`
   Before this MPI broadcast,
   A. `x[0] = a`,
   B. `x[1] = b`, and
   C. `x[2] = c`.
   After this MPI broadcast,
   A. `x[0] = a`,
   B. `x[1] = a`, and
   C. `x[2] = c`.

   (b) Rank `1`

   i. `MPI_Bcast(&x[1], 1, MPI_INT, 0, MPI_COMM_WORLD);`
   Before this MPI broadcast,
   A. `x[0] = b`,
   B. `x[1] = c`, and
   C. `x[2] = a`.
   After this broadcast,
   A. `x[0] = a`,
   B. `x[1] = a`, and
   C. `x[2] = c`.

   ii. `MPI_Recv(&x[1], 1, MPI_INT, 2, 999, MPI_COMM_WORLD, MPI_STATUS_IGNORE);`
   Before this MPI message is received,

A. `x[0] = a`,
B. `x[1] = a`, and
C. `x[2] = c`.

After this MPI message is received,

A. `x[0] = a`,
B. `x[1] = b`, and
C. `x[2] = c`.

iii. `MPI_Bcast(&x[0], 1, MPI_INT, 1, MPI_COMM_WORLD);`

Before this MPI broadcast,

A. `x[0] = a`,
B. `x[1] = b`, and
C. `x[2] = c`.

After this MPI broadcast,

A. `x[0] = a`,
B. `x[1] = b`, and
C. `x[2] = c`.

(c) Rank `2`

i. `MPI_Bcast(&x[0], 1, MPI_INT, 0, MPI_COMM_WORLD);`

Before this MPI broadcast,

A. `x[0] = c`,
B. `x[1] = a`, and
C. `x[2] = b`.

After this broadcast,

A. `x[0] = a`,
B. `x[1] = a`, and
C. `x[2] = b`.

ii. `MPI_Send(&x[2], 1, MPI_INT, 1, 999, MPI_COMM_WORLD);`

Before this MPI message is sent,

A. `x[0] = a`,
B. `x[1] = a`, and
C. `x[2] = b`.

After this MPI message is sent,

A. `x[0] = a`,
B. `x[1] = a`, and
C. `x[2] = b`.

iii. `MPI_Bcast(&x[0], 1, MPI_INT, 1, MPI_COMM_WORLD);`

Before this MPI broadcast,

A. `x[0] = a`,
B. `x[1] = a`, and
C. `x[2] = b`.

After this MPI broadcast,

A. `x[0] = a`,
B. `x[1] = a`, and
C. `x[2] = b`.

These are the outputs returned by `foo` on each of the ranks:

(a) Rank 0: `result = a + b + b`.
(b) Rank 1: `result = 0` because `MPI_Reduce` is called on rank `0`.
(c) Rank 2: `result = 0` because `MPI_Reduce` is called on rank `0`.

2. *MPI collectives*

   (a) See code.
   (b) See code.

3. *Deadlocks and MPI*

   (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()`.
   In MPI, a call to `MPI_Send` blocks until it is safe to use the corresponding send buffer again, and a call to `MPI_Recv` blocks until the corresponding message has been received. If it is the case that there our program does not have an equal number of calls to `MPI_Send()` as there are `MPI_Recv()`, then our program will deadlock, i.e., ranks will wait for events that will not happen. If it is the case that
      i. the call to `MPI_Send()` does not buffer immediately and our program waits on the send to finish,
      ii. there is no corresponding `MPI_Send` for a `MPI_Recv`, or
      iii. not all ranks call a collective such as `MPI_Bcast`,

   the program will deadlock.
   (b) Yes, the given code can result in a deadlock. If the data buffer is too big to fit into the MPI's internals send buffer and there are an odd number of ranks, then the `MPI_Send` corresponding to the last rank will deadlock because there is not a corresponding `MPI_Recv` and MPI will wait until the send buffer is usable again.
   (c) Yes, the given code can result in a deadlock because `MPI_Reduce` is a collective, but it is called only on the zeroth rank. To avoid the deadlock, we can add an `else` clause that calls the reduce collective on every other rank, but the rank does not print the final result.

4. *The Jacobi method with MPI*

   (a)
   (b)
   (c) See code.
   (d) To perform the halo exchange, we need information about the given part's neighbors. We are given the given part's coordinates (`pi`, `pj`, and `pk`). We can then perform the following checks to see if we ought to perform exchanges in the `i-`, `j-`, and `k-` directions:
      i. If `pi != (npi - 1)`, then it follows there is a part in front of us (whose rank is computed via `part_to_rank(pi + 1, pj, pk, npi, npj, npk)`) that we can exchange information with.
      In this case, we receive information about the front part's rearmost layer into the given part's frontmost ghost layer (corresponding to `i_loc = ni_loc`) and send information about the given part's frontmost, non-ghost layer (corresponding to `i_loc = ni_loc - 1`) to the front part's rearmost ghost layer. This is achieved via two nested `for` loops (iterating over `j_loc = 0, ..., nj_loc - 1`, `k_loc = 0, ..., nk_loc - 1`), and due to the memory access pattern used in `ijk_to_index`), we use `MPI_Sendrecv` to send and receive points.
      If `pi != 0`, it follows that there is a part behind us
      (whose rank is computed via `part_to_rank(pi - 1, pj, pk, npi, npj, npk)`) that we can exchange information with.
      The point-by-point exchange is similar to the one in the earlier case, but we receive information into the rearmost ghost layer (corresponding to `i_loc = -1`) and send information about the rearmost, non-ghost layer (corresponding to `i_loc = 0`).

ii. If `pj != (npj - 1)`, then it follows there is a part to the right of us (whose rank is computed via `part_to_rank(pi, pj + 1, pk, npi, npj, npk)`) that we can exchange information with.
In this case, we receive information about the right part's leftmost, non-ghost layer into the given part's rightmost ghost layer (corresponding to `j_loc = nj_loc`) and send information about the given part's rightmost, non-ghost layer (corresponding to `j_loc = nj_loc - 1`) to the right part's leftmost ghost layer. This is achieved via one `for` loop (iterating over `k_loc = 0, ..., nk_loc - 1`), and due to the memory access pattern used in `ijk_to_index`) we use `MPI_Sendrecv` to send and receive strips (starting from `i_loc = -1` and ending at `i_loc = ni_loc`) of length `ni_loc + 2`.
If `pj != 0`, it follows that there is a part to the left of us
(whose rank is computed via `part_to_rank(pi, pj - 1, pk, npi, npj, npk)`) that we can exchange information with.
The strip-by-strip exchange is similar to the one in the earlier case, but we receive information into the leftmost ghost layer (corresponding to `j_loc = -1`) and send information about the leftmost, non-ghost layer (corresponding to `j_loc = 0`).

iii. If `pk != (npk - 1)`, then it follows there is a part on top of us (whose rank is computed via `part_to_rank(pi, pj, pk + 1, npi, npj, npk)`) that we can exchange information with.
In this case, we receive information about the top part's bottommost, non-ghost layer into the given part's topmost ghost layer (corresponding to `k_loc = nk_loc`) and send information about the given part's topmost, non-ghost layer (corresponding to `k_loc = nk_loc - 1`) to the top part's bottommost ghost layer. This is achieved via one `MPI_Sendrecv` (due to the memory access pattern used in `ijk_to_index`), and we send the layer starting from `i_loc = -1`, `j_loc = -1`, and `nk_loc - 1` and ending at `i_loc = ni_loc`, `j_loc = nj_loc`, and `nk_loc - 1` (of size `ni_loc + 2` times `nj_loc + 2`). Likewise, we receive into the layer starting from `i_loc = -1`, `j_loc = -1`, and `nk_loc` and ending at `i_loc = ni_loc`, `j_loc = nj_loc`, and `nk_loc` (of size `ni_loc + 2` times `nj_loc + 2`).
If `pk != 0`, it follows that there is a part to the bottom of us
(whose rank is computed via `part_to_rank(pi, pj, pk - 1, npi, npj, npk)`) that we can exchange information with.
The layer exchange is similar to the one in the earlier case, but we receive information into the bottom ghost layer (corresponding to `k_loc = -1`) and send information about the bottomost, non-ghost layer (corresponding to `nk_loc = 0`).

(e) See code.