Parallel Computing

Exercise sheet 3 + Reference Solution

June 18, 2019

Disclaimer

This document contains the assignments from exercise-sheet 3 of the lecture *184.710 Parallel Computing 2019S*, the reference solution as well as my personal solution.
The reference solution is given directly in the assignments in *italics*, my personal solution is always located in the **Solution** subsection of an exercise.
I cannot guarantee the correctness of any solution provided in this document.

Exercise 1

Some MPI code is being executed by the processes belonging to the communicator `comm`, in particular, each process is looking up its rank and the size of the communicator as follows.

```
MPI_Comm_size(comm, &size);
MPI_Comm_rank(comm, &rank);

assert(size >= 3);

assert(count >= M);
assert(sendbuf != NULL);
assert(recvbuf != NULL);

assert(TAG1 != TAG2);
```

Now, consider the following three variations of the code that follows:

(a) *correct*

```
if (rank == 1){
  MPI_Sendrecv(sendbuf,count,MPI_INT,2,TAG1,
               recvbuf,count,MPI_INT,2,TAG2,
                  comm,MPI_STATUS_IGNORE);
} else if (rank == 2) {
  MPI_Send(sendbuf,count,MPI_INT,1,TAG2,comm);
  MPI_Recv(recvbuf,count,MPI_INT,1,TAG1,comm,MPI_STATUS_IGNORE);
}
```

(b) *Non-matching tags, deadlock. Reverse the tags in send/receive for rank 2.*

```
if (rank == 1){
  MPI_Sendrecv(sendbuf,count,MPI_INT,2,TAG2,
               recvbuf,count,MPI_INT,2,TAG1,
               comm,MPI_STATUS_IGNORE);
} else if (rank == 2) {
  MPI_Recv(recvbuf,count,MPI_INT,1,TAG1,comm,MPI_STATUS_IGNORE);
  MPI_Send(sendbuf,count,MPI_INT,1,TAG2,comm);
}
```

(c) *Deadlocks. Process 1 must send/receive to processes 3 and 2, respectively. Receive and send tags in processes 2 and 3 must be swapped.*

```
if (rank == 1){
  MPI_Sendrecv(sendbuf,count,MPI_INT,2,TAG2,
               recvbuf,count,MPI_INT,2,TAG1,
               comm,MPI_STATUS_IGNORE);
} else if (rank == 2) { // process 2 must send
  MPI_Send(sendbuf,count,MPI_INT,1,TAG2,comm);
} else if (rank == 3) { // process 3 must receive
  MPI_Recv(recvbuf,count,MPI_INT,1,TAG1,comm,MPI_STATUS_IGNORE);
}
```

1. Which of the three pieces (a), (b), and/or (c) are correct, which not? Give an easy repair where needed, but change only arguments to the calls (no code movements).

Solution

1. (a) correct

   (b) incorrect tags

```
if (rank == 1){
  MPI_Sendrecv(sendbuf,count,MPI_INT,2,TAG2,
               recvbuf,count,MPI_INT,2,TAG1,
               comm,MPI_STATUS_IGNORE);
} else if (rank == 2) {
  MPI_Recv(recvbuf,count,MPI_INT,1,TAG2,comm,MPI_STATUS_IGNORE);
  MPI_Send(sendbuf,count,MPI_INT,1,TAG1,comm);
}
```

(c) incorrect tags and incorrect destination

```
if (rank == 1){
  MPI_Sendrecv(sendbuf,count,MPI_INT,3,TAG2,
               recvbuf,count,MPI_INT,2,TAG1,
               comm,MPI_STATUS_IGNORE);
} else if (rank == 2) { // process 2 must send
  MPI_Send(sendbuf,count,MPI_INT,1,TAG1,comm);
} else if (rank == 3) { // process 3 must receive
  MPI_Recv(recvbuf,count,MPI_INT,1,TAG2,comm,MPI_STATUS_IGNORE);
}
```

Exercise 2

Some large array a of n elements is to be shifted from process i to process $i+k$, for $k>0$. The result received at process i is undefined (and may be anything) if $i-k<0$.

The algorithm proposed below for this problem is not optimal, and accomplishes the task by k shifts by one process. The special `MPI_PROC_NULL` process is used to make handling of the cases where $i+k \geq p$ and $i-k<0$ easier. Sending to and receiving from `MPI_PROC_NULL` always succeeds immediately and has no effect.

```
MPI_Comm_size(comm,&size);
MPI_Comm_rank(comm,&rank);

prev = (rank>0)      ? rank-1 : MPI_PROC_NULL;
next = (rank<size-1) ? rank+1 : MPI_PROC_NULL;

for (i=0; i<k; i++) {
  if (rank%2==0) {
    MPI_Send(a,n,MPI_INT,next,TAG,comm);
    MPI_Recv(b,n,MPI_INT,prev,TAG,comm,MPI_STATUS_IGNORE);
  } else {
    MPI_Recv(b,n,MPI_INT,prev,TAG,comm,MPI_STATUS_IGNORE);
    MPI_Send(a,n,MPI_INT,next,TAG,comm);
  }
  if (prev!=MPI_PROC_NULL) {
    for (j=0; j<n; j++) a[j] = b[j];
  }
}
```

The task is to step-wise improve this algorithm.

1. Instead of shifting k times, accomplish the desired result by only one shift from process i directly to process $i+k$. This code should still use only `MPI_Send()` and

`MPI_Recv()`. It must work for any $n \geq 0$ and $p > 0$ and any $k > 0$. Explain what is needed, and write out the (few) extra code lines required.

problem: even-odd distinction, what if i and $i + k$ have the same parity?

```
prev = (rank-k >= 0)  ? rank-k : MPI_PROC_NULL;
next = (rank+k < size)  ? rank+k : MPI_PROC_NULL;

if ((rank/k)%2==0)
  // solves problem if parity is the same
```

Otherwise:

- *0: send to 2, recv from -2*
- *2: send to 4, recv from 0*
- *4: send to 6, recv from 2*
- *6: send to -2, recv from 4*

2. What is a potential performance problem with using only send and receive operations for the implementation?

   Performance problem?

   - *No bidirectional communication possible (perhaps supported by the network).*
   - *Always 2 communication rounds.*

3. Give a solution to the problem using the combined `MPI_Sendrecv()` operation.

   only one `MPI_Sendrecv` operation

4. Mention a case (from the lectures) where the k-shifts operation is used.

   e.g. Hillis-Steele prefix-sums/scan algorithms

Solution

1. The loop is unnecessary as we only send/receive from one process, but the distinction of the send/receive order can not depend on even-odd distinction as this only works for odd k and needs to depend on k as every jump is k processes long each $2k$ processes have the same order. To work correctly for all k shifting-chains the offset of the chain is subtracted.

```
MPI_Comm_size(comm,&size);
MPI_Comm_rank(comm,&rank);

prev = rank - k;
if (prev < 0) prev = MPI_PROC_NULL;

next = rank + k;
if (next >= size) next = MPI_PROC_NULL;
```

```
proc_chain = rank % k;

if ((rank - proc_chain) % 2*k == 0) {
  MPI_Send(a,n,MPI_INT,next,TAG,comm);
  MPI_Recv(b,n,MPI_INT,prev,TAG,comm,MPI_STATUS_IGNORE);
} else {
  MPI_Recv(b,n,MPI_INT,prev,TAG,comm,MPI_STATUS_IGNORE);
  MPI_Send(a,n,MPI_INT,next,TAG,comm);
}
if (prev!=MPI_PROC_NULL) {
  for (j=0; j<n; j++) a[j] = b[j];
}
```

2. The solution does not utilize bidirectional communication networks as the send/receive operations occur sequential (they block).

3. `MPI_Sendrecv()` makes it unnecessary to distinct every second process in the shift-chain.

```
MPI_Comm_size(comm,&size);
MPI_Comm_rank(comm,&rank);

prev = rank - k;
if (prev < 0) prev = MPI_PROC_NULL;

next = rank + k;
if (next >= size) next = MPI_PROC_NULL;

MPI_Sendrecv(a,n,MPI_INT,next,TAG,
             b,n,MPI_INT,prev,TAG,
             comm,MPI_STATUS_IGNORE);

if (prev!=MPI_PROC_NULL) {
  for (j=0; j<n; j++) a[j] = b[j];
}
```

4. The k-shift can be used by the d-dimensional stencil operations to shift the result to the horizontal neighbor ($k = 1$) and the vertical neighbor ($k =$ `width`).

Exercise 3

The following piece of code is part of a recursive algorithm where the set of processes is recursively split until only a single process remains in the communicator.

```
MPI_Comm_size(comm,&size);
MPI_Comm_rank(comm,&rank);
```

```
MPI_Comm_split(comm,((rank%3==0) ? 1 : 0),size-rank,&newcomm);
MPI_Comm_rank(newcomm,&newrank);

int x = rank;
MPI_Bcast(&x,1,MPI_INT,0,newcomm);
printf("x␣is␣%d␣for␣process␣%d,␣now␣%d\n",x,rank,newrank);
```

1. Assume the program is executed with $p = 11$ MPI processes. What is the outcome (in particular, the value of `x`) from each process?

   Something like the following (Note: Only the first recursion-step was required):

```
x is 10 for process 1, now 6
x is 10 for process 2, now 5
x is 10 for process 4, now 4
x is 10 for process 5, now 3
x is 10 for process 7, now 2
x is 10 for process 8, now 1
x is 10 for process 10, now 0
x is 9 for process 0, now 3
x is 9 for process 3, now 2
x is 9 for process 6, now 1
x is 9 for process 9, now 0
```

Solution

1. Assuming, that after the given code-block `comm` and `newcomm` are switched and the code starts again the output would be similar (order may change) to this:

```
x is 9 for process 0, now 3
x is 10 for process 1, now 6
x is 10 for process 2, now 5
x is 9 for process 3, now 2
x is 10 for process 4, now 4
x is 10 for process 5, now 3
x is 9 for process 6, now 1
x is 10 for process 7, now 2
x is 10 for process 8, now 1
x is 9 for process 9, now 0
x is 10 for process 10, now 0
x is 3 for process 0, now 1
x is 2 for process 0, now 1
x is 2 for process 2, now 0
x is 3 for process 3, now 0
x is 0 for process 0, now 0
x is 0 for process 1, now 0
x is 0 for process 0, now 0
x is 0 for process 1, now 0
```

```
x is 6 for process 0, now 2
x is 5 for process 1, now 3
x is 5 for process 2, now 2
x is 6 for process 3, now 1
x is 5 for process 4, now 1
x is 5 for process 5, now 0
x is 6 for process 6, now 0
x is 0 for process 0, now 0
x is 2 for process 1, now 1
x is 2 for process 2, now 0
x is 0 for process 0, now 0
x is 0 for process 1, now 0
x is 3 for process 0, now 1
x is 2 for process 1, now 1
x is 2 for process 2, now 0
x is 3 for process 3, now 0
x is 0 for process 0, now 0
x is 0 for process 1, now 0
x is 0 for process 0, now 0
x is 0 for process 1, now 0
```

Exercise 4

The following example uses one-sided communication to transfer data from all processes except process 0 to process 0 (here: data is just the rank). The code does not do what is intended, and there are at least two reasons for that.

```
if (rank == 0) {
  int *data;
  MPI_Comm_size(comm, &size);
  data = (int*)malloc(size*sizeof(int));

  MPI_Win_create(data, size*sizeof(int), sizeof(int), MPI_INFO_NULL,
                 comm,&win);

  for (i=1; i<size; i++) {
    printf("From rank %d this %d\n", i, data[i]);
  }

  MPI_Win_free(&win);
  free(data);
} else {
  MPI_Win_create(NULL,0,sizeof(int),MPI_INFO_NULL,comm,&win);
  MPI_Comm_rank(comm,&rank);
  MPI_Win_lock(MPI_LOCK_SHARED,0,0,win);
  MPI_Put(&rank,1,MPI_INT,0,0,1,MPI_INT,win);
  MPI_Win_unlock(0,win);
  MPI_Win_free(&win);
}
```

1. Present two fixes to the code such that process 0 can correctly print out the data (rank) transmitted from each of the other processes. You can use either collective operations or additional point-to-point communication.
   - *Fix 1:*
     - *Option add `MPI_Barrier`*
       * *after unlock operations for `rank != 0`*
       * *after `MPI_Win_create` for `rank 0`*
     - *Another option: use `MPI_Win_fence()`*
   - *Fix 2: All non-roots put to the same offset → put offset to i.*

Solution

1. Fixes are included in the following code-listing and indicated by comments.

```
// Get rank before as it is used in the condition
MPI_Comm_rank(comm,&rank);

if (rank == 0) {
  int *data;
  MPI_Comm_size(comm, &size);
  data = (int*)malloc(size*sizeof(int));

  MPI_Win_create(data, size*sizeof(int), sizeof(int), MPI_INFO_NULL,
               comm,&win);

  // Wait for other processes to finish their put-operation
  MPI_Win_fence(0,win);

  for (i=1; i<size; i++) {
    printf("From␣rank␣%d␣this␣%d\n", i, data[i]);
  }

  MPI_Win_free(&win);
  free(data);
} else {
  MPI_Win_create(NULL,0,sizeof(int),MPI_INFO_NULL,comm,&win);
  MPI_Win_lock(MPI_LOCK_SHARED,0,0,win);

  // Add displacement in the target data-sturcture
  // so that the processes don't override the data
  MPI_Put(&rank,1,MPI_INT,
          0,rank*sizeof(int),1,MPI_INT,win);
  MPI_Win_unlock(0,win);

  //Signal finish of write
  MPI_Win_fence(0,win);

  MPI_Win_free(&win);
}
```

Exercise 5

Let us assume that before starting the real communication, process 0 in the communicator `comm` will first read the data needed into an array a of size n (of, say, integers), and then have to distribute these data evenly to the other processes, such that each process has its part of the array a in a (smaller) array b. We want to achieve this using MPI collective operations (but usually do not time this part since it takes $\Omega(n)$ time steps).

1. First, assume that the number of processes p in `comm` divides n, and that all processes already know n, the number of elements in the array. Accomplish the distribution with a single collective operation, and write out the call that all processes in `comm` have to perform. Process 0 is the root process.

   MPI_Scatter(a,n/size,MPI_DOUBLE, b,n/size,MPI_DOUBLE ,0,comm);

2. Now assume that n is not known, and that each process i needs to get instead n_i elements from the a array where n_i's are local values not known to the root. Write a call to a collective operation that allows the root to compute $n = \Sigma_{i=0}^{p-1} n_i$, followed by another collective call that distributes the a array to the smaller b arrays of the processes. Hint: Some (local) computation is necessary here before the second collective call.

   What to do?

   a) *Root needs to get the number of elements (n_i) from all processes.*

   b) *Root sends the required elements to each process.*

```
int ns[size], disp[size];
MPI_Gather(&n,1,MPI_INT,ns,1,MPI_INT,0,comm);
disp[0] = 0;
for (i=1; i<size; i++) disp[i] = disp[i-1]+ns[i-1];
MPI_Scatterv(a,ns,disp,MPI_INT,b,n,MPI_INT,0,comm);
```

Solution

1. Make use of the scatter-routine of MPI:

```
MPI_Comm_rank(comm,&rank);
MPI_Comm_size(comm,&p);

MPI_Scatter(a,n/p,MPI_INT,
            b,n/p,MPI_INT,
            0,comm);
```

2. Make use of gather and vectored scatter routines:

```
MPI_Comm_rank(comm,&rank);
MPI_Comm_size(comm,&p);

if (rank == 0){ //Root

int total_n = 0;
int[] local_n = new int[p];
int[] displs = new int[p];

  MPI_Gather(ni,1,MPI_INT,
             local_n,1,MPI_INT,
             0,comm);

  for(i=0; i<p; i++) {
    displs[i] = total_n;
    total_n += local_n[i];
  }

  MPI_Scatterv(a,local_n,displs,MPI_INT,
               b,ni,MPI_INT,
               0,comm);
} else { //Others
  MPI_Gather(ni,1,MPI_INT,
             NULL,1,MPI_INT,
             0,comm);

  MPI_Scatterv(NULL,NULL,NULL,MPI_INT,
               b,ni,MPI_INT,
               0,comm);
}
```

Exercise 6

Each MPI process in a communicator `comm` has an n-element vector v_i for ranks $0 \leq i < p$ (p is the number of processes in `comm`). The task is, using MPI collective functionality (almost) exclusively, to compute all *inclusive prefix-sums* for the sequence of vectors v_i that is $w_i = \Sigma_{j=0}^{i} v_j$ for process i (note that Σ means vector addition here), and the total element-wise sums over all vectors, that is $\Sigma_{j=0}^{p-1} v_j$. You may assume that the elements are `double`s and that the MPI operator is `MPI_SUM`.

1. Which collective operations are you using? Write out a code snippet solving the problem.

   Example code given:

```
double v[n]; // initialized somewhere
double w[n];

MPI_Comm_rank(comm,&rank);

MPI_Scan(v,w,n,MPI_DOUBLE,MPI_SUM,comm);

// intended solution
if (rank == size-1) {
  // copy from w to sum
  int i;
  for (i=0; i<n; i++) sum[i] = w[i];
}
MPI_Bcast(sum,n,MPI_DOUBLE,size-1,comm);

// second best solution
MPI_Allreduce(v,sum,n,MPI_DOUBLE,MPI_SUM,comm);
```

2. Using the best-case assumptions as explained in the lecture (collective operations in fully connected network), what may the asymptotic running time of your solution be (recall that MPI does not define a performance model and does not give any guarantees for the collective operations)?

   running time: $O(n + \log p)$

Solution

1. The MPI routine for *inclusive prefix-sums* is `MPI_Scan`

```
MPI_Comm_size(comm,&p);

MPI_Scan(vi,wi,n,MPI_DOUBLE,MPI_SUM,comm);

//Broadcast total element-wise sum
//which is obtained in the last (p-1) process
double[] total_sum = new double[n];
memcpy(total_sum,wi,n*sizeof(double));
MPI_Bcast(total_sum,n,MPI_DOUBLE,p-1,comm)
```

2. The asymptotic runtime is $O(n + \log p)$ ($+O(n + \log p)$ for the broadcast) as the operation can be distributed tree-like. The process $p - 1$ would serve as root (and contain the total sum).

Exercise 7

An array `a` is distributed across a set of MPI processes with each part stored in a local array `a[]` of n elements (such that the total number of elements is pn). The task is to compute the *exclusive prefix-sums* for each element in the distributed array, that is for element `a[i]` at process r the sum of all elements at process before r (rank $r' < r$) plus the sum of the elements local to process r with index $j < i$. The idea is explained in the lecture on prefix-sums. Use MPI collective functions as much as possible. You may assume that n is the same for all processes. The number of processes p can be very large. Your solution should be scalable/efficient for large p and large n. A code template is given below:

```
MPI_Comm_rank(comm,&rank);

// local reduction
sum = a[0];
for (i=1; i<n; i++) sum += a[i];

// MPI code/function to compute the partial sum required for rank
...
// sum is a partial sum for rank

if (rank == 0) {
  sum = a[0];
} else {
  e = a[0];
  a[0] = sum;
  sum += e;
}

// complete the exclusive prefix-sums for rank,
// use sum as running sum
...
```

1. Complete the code template with the necessary MPI function(s) to communicate.

   Example code given:

```
MPI_Comm_rank(comm,&rank);

// local reduction
sum = a[0];
for (i=1; i<n; i++) sum += a[i];
```

```
MPI_Exscan(MPI_IN_PLACE,&sum,1,MPI_INT,MPI_SUM,comm);

if (rank == 0) {
  sum = a[0];
} else {
  e = a[0];
  a[0] = sum;
  sum += e;
}

for (i=1; i<n; i++) {
  e = a[i];
  a[i] = sum;
  sum += e;
}
```

2. What may an estimated asymptotic running time be, using optimistic assumptions as discussed in the lecture?
   - *MPI_Exscan with one element:* $O(\log p)$
   - *total time is* $O(n + \log p)$

Solution

1. An implementation using `MPI_Exscan`:

```
MPI_Comm_rank(comm,&rank);

// local reduction
sum = a[0];
for (i=1; i<n; i++) sum += a[i];

// Compute the partial sum required for rank
MPI_Exscan(MPI_IN_PLACE,&sum,1,MPI_INT,comm)

if (rank == 0) {
  sum = a[0];
} else {
  e = a[0];
  a[0] = sum;
  sum += e;
}

// Complete local (exclusive) prefix sums
for (i=1; i<n; i++){
  e = a[i];
  a[i] = sum;
  sum += e;
}
```

2. Local work (per process) is $O(2n)$ and the communication for the *exclusive prefix-sums* should be in $O(n \log p)$ when using optimistic assumptions (tree-like communication structure), but uses just 1 value per process (the sum) and therefore is in $O(\log p)$ Asymptotic running time is therefore $O(n + \log p)$.