SF2568: Homework 2

Daniel Persson Proos dproos@kth.se
Borja Rodríguez Gàlvez borjarg@kth.se

March 2, 2018

1. (a) In order to do a on-to-all broadcast we make use of the recursive doubling idea as you can see in Algorithm 1.
There are small subtilities in the algorithm that do not appear in the standard all-to-all exange of information. Now we have a process, $SOURCE$, which sends the message to all the other processes; therefore, we have to ensure that at each step of the recusrive doubling only the processes which contain the message send it and the others receive it. In order to overcome this we first look at the case where the source is the process 0 and then we will be able to easiliy generalize to process $SOURCE$; where $SOURCE \in [0, P-1]$.
When the source process is 0 we can apply recursive doubling having only the processes where the first bit changes taking action (i.e., process 1 receiving and process 0 sending). After doing that, we can repeat this setting with the ones that change only the second bit; having the ones with null second bit sending and the active second bit receiving (i.e., 0 and 1 sending and 2 and 3 receiving). Then, we continue this setting until all processes have received the message. If the number of processes is not a power of 2 (i.e., $2^D < P < 2^{D+1}$) we only have to check if the process we have to send to exists or not, since the receiving process will have strictly a higher number than ours. An example of that is shown in Figure 1.

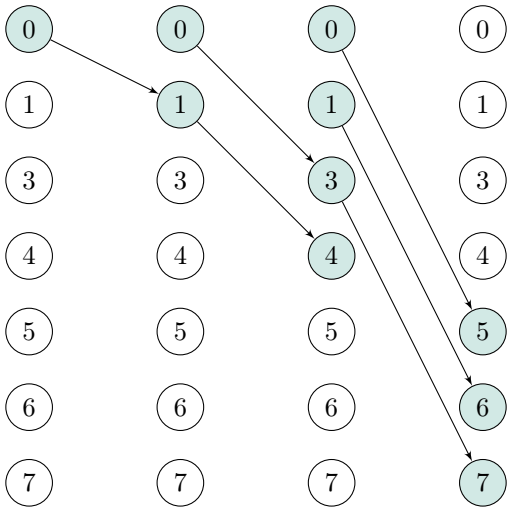

Figure 1: Example of 0-source one-to-all broadcast with 7 processes.

When the source node is not 0, we can just create a maping to all the processes to another name (in the Algorithm 1 we call it $myRank$) in which the source is 0 and all the other processes have a unique identifier in the same range of values and (i.e., 0 to P-1). This way we use the same previous technique but looking at this new identifier instead of looking at the process rank. An easy bit-wise operation that does this job is an XOR with the source rank.

(b) For the time analysis of the algorithm we define the following variables to be t_{su}: set up time; t_d: transfer data time; t_a: action time; and m: message length. With this definition the communication time is directly:

Algorithm 1 Broadcast Algorithm

```
1: procedure BROADCAST(SOURCE, MESSAGE)
2:     // D is the bit length s.t. 2^D ≤ P ≤ 2^{D+1}
3:     mask = 2^{D+1} − 1
4:     // rank is the process rank
5:     // Change the rank values so rank is equivalent to 0
6:     myRank = SOURCE XOR rank
7:     for d = 0 → D + 1 do
8:         // Updating the mask for the bits we are considering
9:         mask = bitflip(mask, d)
10:        // Checking if our rank is being considered for sending or receiving
11:        if mask AND myRank = 0 then
12:            q = bitflip(p,d)
13:            // We only continue if the process exists
14:            if q < P then
15:                // If the bit is off send else receive
16:                if myRank AND 2^d = 0 then
17:                    send(q, MESSAGE)
18:                else
19:                    MESSAGE = receive(q)
20:            else
21:                break
```

$$t_{comm} = (D+1)(t_{su} + mt_d) \approx \log P(t_{su} + mt_d) \tag{1}$$

Then, we can compute the computation time as follows:

$$t_{comp} = 2t_a + (D+1)(2t_a + 3t_a) \approx \log P(5t_a), \tag{2}$$

where the first $2t_a$ account from the generation of the mask and the XOR operation with the source; the first $2t_a$ multiplying $(D+1)$ account for the two bitflips and the last $3t_a$ multiplying $(D+1)$ come from the comparisons and $ANDs$. Therefore, the total time is approximately,

$$t_P \approx \log P(t_s u + mt_d + 5t_a) \tag{3}$$

which is indeed an $\mathcal{O}(\log P)$ algorithm.

(c) The scatter algorithm (Algorithm 2) share the same ideas than the one-to-all broadcast algorithm. For this algorithm we are asked to spread the information on one processor among all the processors (see Figure 2 for an example).

In order to develop this algorithm it is enough to apply the same skeleton that the broadcasting algorithm had but now sending only halves of the message each time. The problem that arises doing this algorithm is only that the messages would not be ordered as they would be in a linear distribution fashion. To overcome this problem we can simply rearange the message puting the first half to be only the even positions of the message (i.e., 0, 2, 4, ...) and the last half to be the odd positions (i.e., 1, 3, 5 ...). This way we are following the mask order and they will be distributed as in a linear data distribution.

Again, this distribution is only linear if the source process is process 0. To overcome this, we can simply save the new id for each process (i.e., $myRank$) and the data will be linearly distributed over the process in these new process indexing.

As in the previous exercise, this algorithm is $\mathcal{O}(\log P)$, since no additional communications are done that they were not in the one-to-all broadcasting.

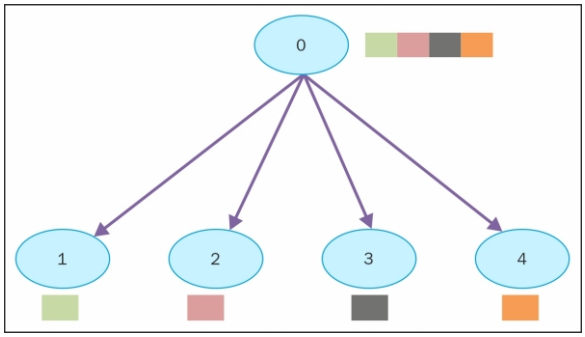

Figure 2: Scatter operation with 0-source and linear data distribution with 4 processes.

Algorithm 2 Scatter Algorithm

```
1:  procedure SCATTER(SOURCE, MESSAGE)
2:      // D is the bit length s.t. 2^D ≤ P ≤ 2^(D+1)
3:      mask = 2^(D+1) − 1
4:      // rank is the process rank
5:      // Change the rank values so rank is equivalent to 0
6:      myRank = source XOR rank
7:      for d = 0 → D do
8:          // Updating the mask for the bits we are considering
9:          mask = bitflip(mask, d)
10:         // Checking if our rank is being considered for sending or receiving
11:         if mask AND myRank = 0 then
12:             q = bitflip(p,d)
13:             // We only continue if the process exists
14:             if q < P then
15:                 // If the bit is off send else receive
16:                 if myRank AND 2^d = 0 then
17:                     MESSAGE = orderMessageEvenOdd(MESSAGE)
18:                     msgLength = length(MESSAGE)
19:                     length = floor(msgLength / 2)
20:                     send(q, MESSAGE[0:length-1])
21:                     MESSAGE = MESSAGE[length:msgLength]
22:                 else
23:                     MESSAGE = receive(q)
24:             else
25:                 break
```

2. In this exercise we want to design an algorithm for vector transposition, a very widely used operation that usually is required in parallel computations.

   (a) Basic algorithm when P = Q

   (b) Performance analysis

   (c) Extra points when P != Q

3. Now we want to solve the following 1D differential equation,

$$u'' + r(x)u = f(x), \quad 0 < x < 1, \quad u(0) = u(1) = 0 \tag{4}$$

We assume that $r(x) \leq 0 \ \forall x \in (0,1)$; in our particular case, we have set $r(x) = \cos(20x) - 1$. Also, we wanted $u(0) = u(1) = 0$, so we set the function $u(x)$ to be $\sin(5\pi x)$ in order to be able to calculate the approximation MSE. Finally, in order to perform the calculations, we have solved the 1D ODE for $f(x)$ and obtained $f(x) = \sin(5\pi x)(\cos(20x) - 1) - 25\pi^2 \sin(5\pi x)$ (see Figure 3).

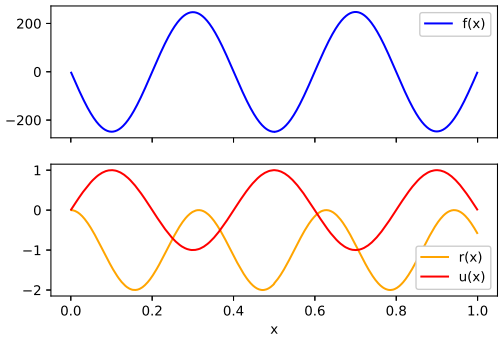

Figure 3: Computation of the exact functions taking place in the ODE.

Now, in order to solve the problem we have discretized the equation as follos: Given $N > 0$ and $h = \frac{1}{N+1}$ amb $x_n = nh$, $0 \leq n \leq N+1$. Then the system reads,

$$\frac{1}{h^2}(u_{n-1} - 2u_n + u_{n+1}) + r(x_n)u_n = f(x_n), \quad n = 1...N \tag{5}$$

where $u(x_n) \approx u_n$ and $u_0 = u_{N+1} = 0$. We solved the problem using the following Jacobi iteration,

$$u_n^{k+1} = \frac{u_{n-1}^k + u_{n+1}^k - h^2 f(x_n)}{2.0 - h^2 r(x_n)}. \tag{6}$$

We used 20 processes and converged to the right solution in around 25000 steps as you can see in Figure 4; where after that point the MSE is stable around 0. Furthermore, to see how the Jacobi iteration process gets to the right solution, we provide with a plot of the estimation at different stages (Figure 5).

The code we used for this computation is presented in Listing 1. Also, the printing functions are also shown in Listing 2.

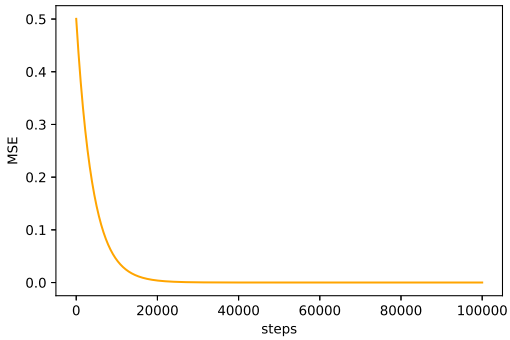

Figure 4: Decrease of the MSE at each iteration of the Jacobi iteration process.

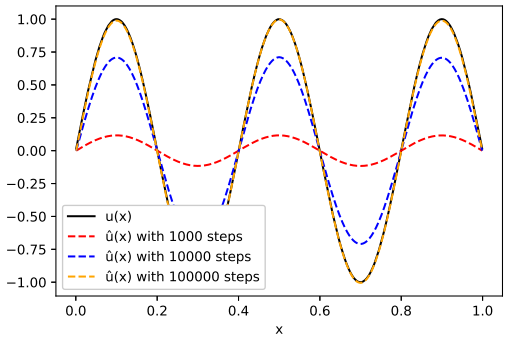

Figure 5: Estimation of the $u(x)$ function using the Jacobi iteration process with different number of iterations.

Listing 1: MPI implementation of an ODE solver using Jacobi iteration process

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>
#include <string.h>

#define N 1000 // number of inner grid points
#define SMX 1000000 // number of iterations
#define M_PI 3.14159265358979323846 // PI
#define H (1.0 / (N + 1.0)) // The first and last points x = 0, 1 must be 0
#define BLACK 0
#define RED 1

double r_func(double x)
{
  return cos(20*x) - 1.0;
}

double u_func(double x)
{
  return sin(5*M_PI*x);
}

double f_func(double x)
{
  return sin(5*M_PI*x)*(cos(20*x) - 1) - 25*M_PI*M_PI*sin(5*M_PI*x);
}

int main(int argc, char* argv[])
{
  /* Local variables */
  int size, rank, rc, tag;
  MPI_Status status;
  tag = 5;

  /* Initialize MPI */
  rc = MPI_Init(&argc, &argv);
  rc = MPI_Comm_size(MPI_COMM_WORLD, &size);
  rc = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (N < size){
    fprintf(stderr, "%s\n", "Too few discretization points...");
    exit(1);
  }

  /* Linear data distribution */
  int Ip, inVal, M_tild, L, R, a;
  a = 2;
  M_tild = N + (size - 1)*a;
  L = M_tild / size;
```

```
50      R = M_tild % size;
51      Ip = L + (rank < R ? 1: 0);
52      inVal = (L-a)*rank + (rank < R ? rank : R);
53
54      /* Create the data */
55      double u_real[Ip], u[Ip], u_new[Ip];
56      double f[Ip], r[Ip];
57      double se;
58      double mse[SMX];
59
60      /* Create the x values */
61      double x[Ip];
62      for (unsigned int i = 0; i < Ip; i++){
63        x[i] = (inVal + i + 1)*H;
64      }
65
66      /* Compute the real values of u that we are going to use for the error */
67      /* Also, compute the values of the functions so we don't need to do
            calculations
68         at each iteration */
69      for (unsigned int i = 0; i < Ip; i++){
70        u_real[i] = u_func(x[i]);
71        f[i] = f_func(x[i]);
72        r[i] = r_func(x[i]);
73        u[i] = 0.0;
74      }
75
76      /* Actual computation */
77      for (unsigned int step = 0; step < SMX; step++){
78        // Communication
79        if (rank % 2 == BLACK){
80          if (rank < size - 1){ // Avoid sending to P process
81            rc = MPI_Send(&u[Ip-2], 1, MPI_DOUBLE, rank+1, tag, MPI_COMM_WORLD);
82            rc = MPI_Recv(&u[Ip-1], 1, MPI_DOUBLE, rank+1, tag, MPI_COMM_WORLD, &
                  status);
83          }
84          if (rank > 0){ // Avoid sending to -1 process
85            rc = MPI_Send(&u[1], 1, MPI_DOUBLE, rank-1, tag, MPI_COMM_WORLD);
86            rc = MPI_Recv(&u[0], 1, MPI_DOUBLE, rank-1, tag, MPI_COMM_WORLD, &
                  status);
87          }
88        } else{ // rank % 2 == RED
89          if (rank > 0){ // Avoid sending to -1 process
90            rc = MPI_Recv(&u[0], 1, MPI_DOUBLE, rank-1, tag, MPI_COMM_WORLD, &
                  status);
91            rc = MPI_Send(&u[1], 1, MPI_DOUBLE, rank-1, tag, MPI_COMM_WORLD);
92          }
93          if (rank < size - 1){ // Avoid sending to (size) process
94            rc = MPI_Recv(&u[Ip-1], 1, MPI_DOUBLE, rank+1, tag, MPI_COMM_WORLD, &
                  status);
95            rc = MPI_Send(&u[Ip-2], 1, MPI_DOUBLE, rank+1, tag, MPI_COMM_WORLD);
96          }
97        }
98        // Computation of the values
99        for (unsigned int i = 1; i < Ip-1; i++){
100         u_new[i] = (u[i-1] + u[i+1] - H*H*f[i]) / (2.0 - H*H*r[i]);
101       }
102
103       // We update our values and compute the error
104       se = 0;
105       for (unsigned int i = 0; i < Ip; i++){
106         if (i == 0 && rank != 0) continue; // Only the first process uses the
              first element
107         else if(i == Ip-1 && rank != size-1) continue; // Only the last element
              computes the last element
108         u[i] = u_new[i];
109         se += (u[i] - u_real[i])*(u[i] - u_real[i]);
110       }
111       // We share the squared error and compute the mean
112       MPI_Allreduce(&se, &mse[step], 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
113       mse[step] /= N;
114     }
115
```

```
    // Finally we write the error in a file (only one process should do it)
    if (rank == 0){
      FILE *err_fp;
      err_fp = fopen("Files/err_matr.txt","wb"); // Here the name of your file
      for (unsigned int step = 0; step < SMX; step++){
        fprintf(err_fp, "%.10f\n", mse[step]);
      }
      fclose(err_fp);
    }

    // Also, we write (size) files for the final u approximation
    FILE *u_fp;
    char pId[5];
    char nameFile[50] = "Files/u_matr_stx_10000_p";
    sprintf(pId, "%d", rank);
    strcat(nameFile,pId);
    strcat(nameFile, ".txt");
    u_fp = fopen(nameFile,"wb"); // Here the name of your file
    for (unsigned int i = 0; i < Ip; i++){
      if (i == 0 && rank != 0) continue; // Only the first process prints the
          first element
      else if (i == Ip-1 && rank != size-1) continue; // Only the last element
          prints the last element
      fprintf(u_fp, "%.10f\n", u[i]);
    }
    fclose(u_fp);

    /* Finish the process */
    MPI_Finalize();
    exit(0);
  }
```

Listing 2: Reading the files and printing

```
import numpy as np
import matplotlib.pyplot as plt

# They were precomputed in C
f_m = np.loadtxt("Files/f_matr.txt")
u_m = np.loadtxt("Files/u_matr.txt")
r_m = np.loadtxt("Files/r_matr.txt")
x_m = np.loadtxt("Files/x_matr.txt")

# Plot of the exact functions
fig, (ax1, ax2) = plt.subplots(2,1,sharex = True)
ax1.plot(x_m, f_m, label = "f(x)", color = "blue")
ax2.plot(x_m, r_m, label = "r(x)", color = "orange")
ax2.plot(x_m, u_m, label = "u(x)", color = "red")
ax1.legend()
ax2.legend()
ax2.set_xlabel("x")
plt.savefig("Files/prob_descr.eps", format = "eps")
plt.plot()

# Get the computed error
e_m = np.loadtxt("Files/err_matr.txt")

# Plot of the error
fig, ax = plt.subplots()
ax.plot(np.arange(len(e_m)), e_m, color = "orange")
ax.set_ylabel("MSE")
ax.set_xlabel("steps")
plt.savefig("Files/MSE_dec.eps", format = "eps")

# Get the estimation of u
u_tild_m_1000 = np.array(())
for i in range(20):
    name = "Files/u_matr_stx_1000_p" + str(i) + ".txt"
    u_tmp = np.loadtxt(name)
    u_tild_m_1000 = np.hstack((u_tild_m_1000, u_tmp))
u_tild_m_10000 = np.array(())
```

```
for i in range(20):
    name = "Files/u_matr_stx_10000_p" + str(i) + ".txt"
    u_tmp = np.loadtxt(name)
    u_tild_m_10000 = np.hstack((u_tild_m_10000, u_tmp))
u_tild_m_100000 = np.array(())
for i in range(20):
    name = "Files/u_matr_stx_100000_p" + str(i) + ".txt"
    u_tmp = np.loadtxt(name)
    u_tild_m_100000 = np.hstack((u_tild_m_100000, u_tmp))

# Plot the estimation
fig, ax = plt.subplots()
ax.plot(x_m, u_m, label = "u(x)", color = "black")
ax.plot(x_m, u_tild_m_1000, label = "u^{^}(x) with 1000 steps",
        color = "red", linestyle = "dashed")
ax.plot(x_m, u_tild_m_10000, label = "u^{^}(x) with 10000 steps",
        color = "blue", linestyle = "dashed")
ax.plot(x_m, u_tild_m_100000, label = "u^{^}(x) with 100000 steps",
        color = "orange", linestyle = "dashed")
ax.legend()
ax.set_xlabel("x")
plt.savefig("Files/func_appr.eps", format = "eps")
```