Homework 2 – SF2568 Parallel Computations for Large-Scale Problems

Cuong Duc Dao, Donggyun Park
`cuongdd@kth.se, donggyun@kth.se`

March 02, 2021

Question 1: The broadcast operation is a one-to-all collective communication operation where one of the processes sends the same message to all other processes.

(a) Assume our simple communication model for point-to-point communication. A straightforward implementation would require $P - 1$ communication steps. Design an algorithm for the broadcast operation using only point-to-point communications which requires $O(\log P)$ communication steps.

(b) Do a time performance analysis for your algorithm

(c) How can the scatter operation be implemented using $O(\log P)$ communication steps?

Solution:

(a) Using recursive doubling we divide the data distribution according to the Figure 1. It only requires O($log_2 P$). If we set the range of an arbitrary number of processors P for $2^T \leq$ P $< 2^{T+1}$, the pseudo-code for this algorithm is as below 1

Algorithm 1 Recursive doubling algorithm for broadcasting

```
if $p \leq 2^T$ then
    send(data, bitflip(p, T))
end if
if $p < P - 2^T$ then
    receive(data, bitflip(p, T))
end if
if $p < 2^T$ then
    for t = 0 to T-1 do
        loc ← bitflip(p, t)
        if $p < 2^t$ then
            send(data, loc)
        else
            if $p < 2^{t+1}$ then
                receive(data, loc)
            end if
        end if
    end for
end if
if $p < P - 2^T$ then
    send(data, bitflip(p, T))
end if
if $p \geq 2^T$ then
    receive(data, bitflip(p, T))
end if
```

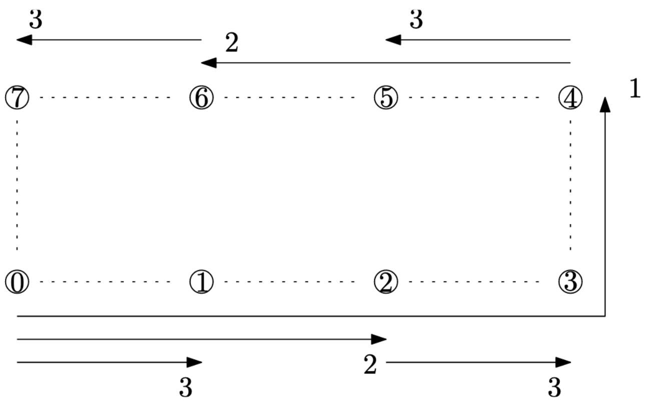


Figure 1: Illustration of Recursive Doubling for broadcast operation between 8 processes

(b)
- Local computation time: $t_{comp,1} = 1t_a$ (where t_a is the time for one basic operation)
- Recursive doubling step: $t = T(t_{startup} + w_p t_{data} + 3t_a) = \log P(t_{startup} + wt_{data} + 3t_a)$ where $3t_a$ is from the two if statement evaluations and the computation of the destination. We have $t_{startup} + wt_{data}$ from the send or receive that is triggered at most once per loop.
- Total computation time: $T_p = t_{comp,1} + t = t_a + \log P(t_{startup} + wt_{data} + 3t_a)$

(c) Assuming that we have a message of length M which can be divided into P processors, so each processor would own a chunk of data of length M/P. The scatter operation sends each chunk to its corresponding processor.
This operation can be done in $O(\log P)$ time steps using the **recursive doubling** strategy mentioned above. Processor with lower rank sends the upper half of the data to the middle node by performing the reverse bit-flip operation, i.e., from highest bit to lowest bit. A graphical illustration of this algorithm is given in Figure 2

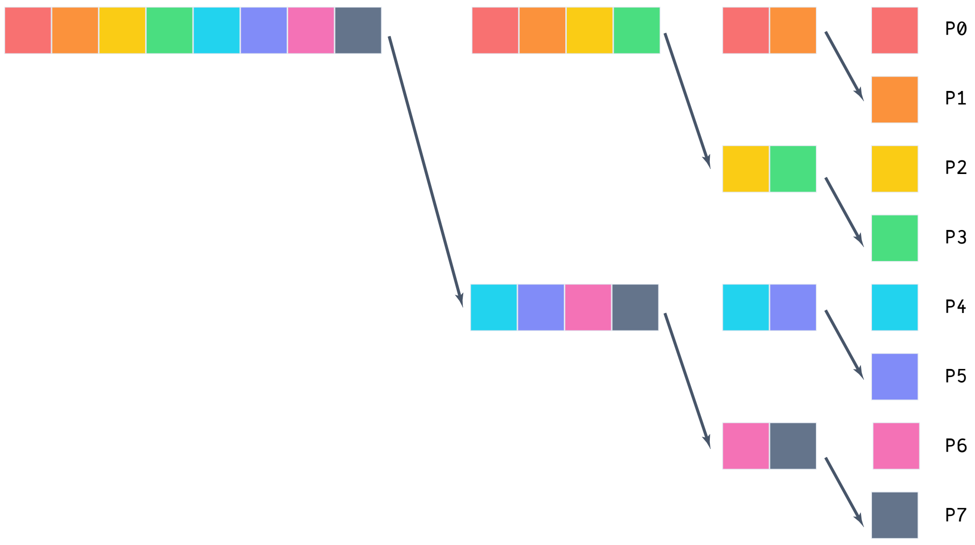


Figure 2: Illustration of scatter operation with recursive doubling with $P = 8$

Question 2: Consider a matrix A distributed on a $P \times P$ process mesh. An algorithm has been given in the lecture for evaluating the matrix-vector product $y = Ax$. While x is column distributed, y is row distributed. In order to carry out a further multiplication Ay, the vector y must be transposed (the row-distributed y must be redistributed to become column-distributed).

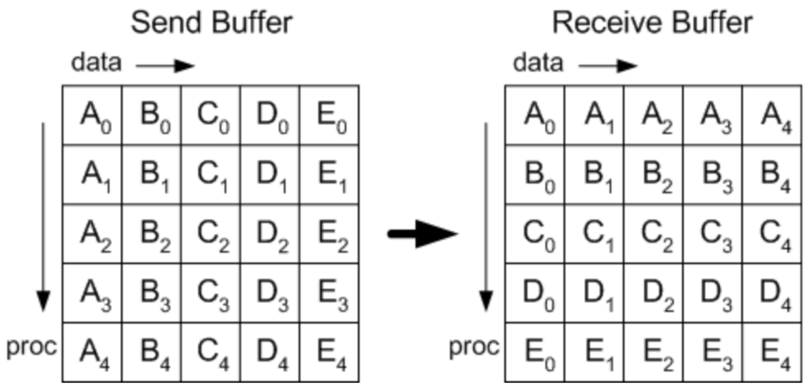
(a) Design an algorithm for this transposition. (1)

(b) Make a performance analysis of your transposition algorithm. (1)

(c) An extra credit will be given for a good (!) solution in the case that A is distributed in a PxQ process mesh with P != Q.

Solution:

(a) Let us assume that a vector y has N elements. It needs to be divided into the number of processors to be used in PxP process mesh. Then N/P elements are assigned for each processor. 'MPI_Alltoall' command distributes the data stored in a processor to every processor including itself. So when the data list and the processor aligned in the corresponding order, it automatically does the matrix transposition. Conceptually, it is the same as the scattering strategy Question 1, where each processor scatter its data to all the other processors.



A_0	B_0	C_0	D_0	E_0
A_1	B_1	C_1	D_1	E_1
A_2	B_2	C_2	D_2	E_2
A_3	B_3	C_3	D_3	E_3
A_4	B_4	C_4	D_4	E_4

A_0	A_1	A_2	A_3	A_4
B_0	B_1	B_2	B_3	B_4
C_0	C_1	C_2	C_3	C_4
D_0	D_1	D_2	D_3	D_4
E_0	E_1	E_2	E_3	E_4

Figure 3: This example shows how a vector has been divided into P chunks ($P = 5$ in this case) and each chuck has P elements and transposed by MPI_Alltoall command

After this transposition is done y is column-distributed, if necessary, we need to concatenate them back together, then we obtain the column distributed vector y.

(b) Because each processor can perform the scatter operation at the same time with other processors, the time complexity of this transposition is the same as the complexity of the scatter operation above: $T_p = t_{comp,1} + t = t_a + \log P(t_{startup} + wt_{data} + 3t_a)$

(c) Let's take an example with 20 elements.
In this case, P = 4 and Q = 3. Let's think about it with the given example below. We have this 4 by 3 matrix.

1	2	3
1	2	3
1	2	3
1	2	3

and we add 'NULL' (empty) at the end of each row. Then it becomes PxP (4x4 in this case). The data distribution would look like below.

1	2	3	NULL
1	2	3	NULL
1	2	3	NULL
1	2	3	NULL

Now we need to transpose as what we do in part (a). After finishing the transposition, it should be in the form below

1	1	1	1
2	2	2	2
3	3	3	3
NULL	NULL	NULL	NULL

Finally we have obtained the transposed version of the original.

Question 3: Approximate solution for 1D-poisson partial differential equation with Jacobi iterations.

Solution:

The data is linearly distributed over processes. The challenge is that each process would need data from some indexes (in the data vector) from its neighbour processes. This problem is tackled with red-black communication. The details implementation of the red/black algorithm can be found in the code listing (line 96 - 110).

Figure 4 and 5 present the results of our code for approximating the solution for the Poisson partial differential equation with Jacobi iterations. We can see that there is no notable difference between the approximated and the true function in the plots. Therefore, we can assume that the numerical approximation has reached convergence before 10^6 iterations.

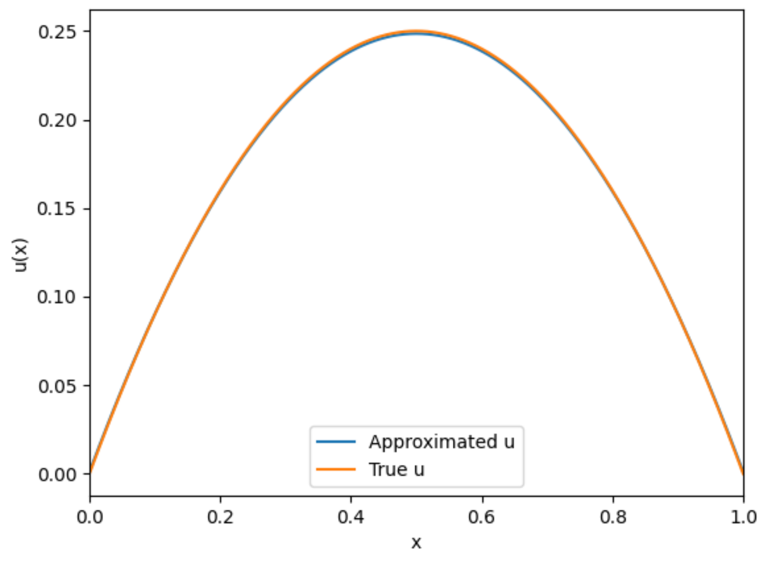


Figure 4: $u(x) = x(x-1); \quad r(x) = x - 1; \quad f(x) = 2 + x(x-1)^2$

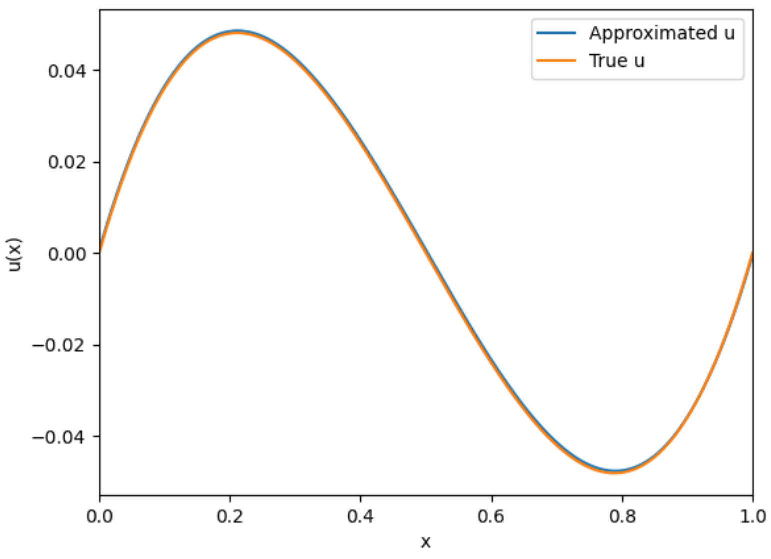


Figure 5: $u(x) = x\left(x - \frac{1}{2}\right)(x-1)$; $\quad r(x) = x - 1$; $\quad f(x) = 6x - 3 + (x-1)\,x\left(x - \frac{1}{2}\right)(x-1)$

Implementation of Jacobi iteration method for approximating the solution for Poisson partial differential equation

```c
/* Reaction-diffusion equation in 1D
 * Solution by Jacobi iteration
 * simple MPI implementation
 *
 * C Michael Hanke 2006-12-12
 */
#include <stdlib.h>
#include <stdio.h>

#define MIN(a,b) ((a) < (b) ? (a) : (b))

/* Use MPI */
#include "mpi.h"

/* define problem to be solved */
#define N 1000   /* number of inner grid points */
#define SMX 1000000 /* number of iterations */

/* implement coefficient functions */
extern double r(const double x);
extern double f(const double x);

double r(const double x) {
    // Change here before submission: -x
    // return x - 1;
    // return -x;
    return x - 1;
}

double f(const double x) {
    // return -2 + x * x * (x - 1);
    // return 2 + (x - 1) * x * (x - 1);
    return 6*x - 3 + (x-1)*x*(x-0.5)*(x-1);
}

/* We assume linear data distribution. The formulae according to the lecture
   are:
      L = N/P;
      R = N%P;
      I = (N+P-p-1)/P;    (number of local elements)
      n = p*L+MIN(p,R)+i; (global index for given (p,i)
   Attention: We use a small trick for introducing the boundary conditions:
      - The first ghost point on p = 0 holds u(0);
      - the last ghost point on p = P-1 holds u(1).
   Hence, all local vectors hold I elements while u has I+2 elements.
*/

int main(int argc, char *argv[])
{
    /* local variable */
    int P, p, tag, L, R, I, n, color, step, i;
    double h, *u, *unew, *ff, *rr, x_n;
    MPI_Status status;

    /* Initialize MPI */
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &P);
    MPI_Comm_rank(MPI_COMM_WORLD, &p);
    if (N < P) {
        fprintf(stdout, "Too few discretization points...\n");
        exit(1);
    }
    tag = 100;
    color = p % 2;  // 0: red, 1: black
    printf("Node %d, color: %d\n", p, color);
    /* Compute local indices for data distribution */
    L = N / P;
    R = N % P;
    I = (N + P - p - 1) / P;  // Number of local elements
    h = 1.0 / (N + 1);

    printf("Processor %d, L = %d; R = %d; I = %d; h = %f\n", p, L, R, I, h);
    /* arrays */
    unew = (double *) malloc(I*sizeof(double));
    /* Note: The following allocation includes additionally:
    - boundary conditins are set to zero
    - the initial guess is set to zero */
    u = (double *) calloc(I+2, sizeof(double));

    ff = (double *) malloc(I * sizeof(double));
    rr = (double *) malloc(I * sizeof(double));

    for (i = 0; i < I; i++) {
        n = p * L + MIN(p, R) + i;  // Global index for given (p, i)
        x_n = n * h;
        ff[i] = f(x_n);
        rr[i] = r(x_n);
    }

    /* Jacobi iteration */
    for (step = 0; step < SMX; step++) {
        if (step % 100000 == 0) {
            printf("Jacobi, processor %d, iter %d\n", p, step);
        }
        /* RB communication of overlap */
        if (color == 0) { // red
            MPI_Send(&u[I], 1, MPI_DOUBLE, p+1, tag, MPI_COMM_WORLD);
            MPI_Recv(&u[I+1], 1, MPI_DOUBLE, p+1, tag, MPI_COMM_WORLD, &status);
            if (p != 0) {
                MPI_Send(&u[1], 1, MPI_DOUBLE, p-1, tag, MPI_COMM_WORLD);
                MPI_Recv(&u[0], 1, MPI_DOUBLE, p-1, tag, MPI_COMM_WORLD, &status);
            }
        } else { // black
            MPI_Recv(&u[0], 1, MPI_DOUBLE, p-1, tag, MPI_COMM_WORLD, &status);
            MPI_Send(&u[1], 1, MPI_DOUBLE, p-1, tag, MPI_COMM_WORLD);
            if (p != P-1) {
                MPI_Recv(&u[I+1], 1, MPI_DOUBLE, p+1, tag, MPI_COMM_WORLD, &status);
                MPI_Send(&u[I], 1, MPI_DOUBLE, p+1, tag, MPI_COMM_WORLD);
            }
        }
        /* local iteration step */
        for (i = 0; i < I; i++) {
            unew[i] = (u[i]+u[i+2]-h*h*ff[i])/(2.0-h*h*rr[i]);
```

```c
            // unew[i] = (u[i-1] + u[i+1] - h*h*ff[i]) / (2.0 - h*h*rr[i]);
        }
        for (i = 0; i < I; i++) {
            u[i+1] = unew[i];
        }
    }

    /* output for graphical representation */
    /* Instead of using gather (which may lead to excessive memory requirements
    on the master process) each process will write its own data portion. This
    introduces a sequentialization: the hard disk can only write (efficiently)
    sequentially. Therefore, we use the following strategy: */
    FILE *f;
    double signal;
    // 1. The master process writes its portion. (file creation)
    if (p == 0) {
        f = fopen("possion.txt", "w");
        for (i = 0; i < I; i++) {
            fprintf(f, "%f ", unew[i]);
        }
        fclose(f);
        // 2. The master sends a signal to process 1 to start writing.
        signal = 1.0;
        MPI_Send(&signal, 1, MPI_DOUBLE, 1, tag, MPI_COMM_WORLD);
        printf("Master processor wrote and sent signal\n");
    } else {
        // 3. Process p waites for the signal from process p-1 to arrive.
        MPI_Recv(&signal, 1, MPI_DOUBLE, p-1, tag, MPI_COMM_WORLD, &status);
        printf("Received signal %f from processor %d\n", signal, p-1);
        if (signal == 1.) {
            // 4. Process p writes its portion to disk. (append to file)
            f = fopen("possion.txt", "a");
            for (i = 0; i < I; i++) {
                fprintf(f, "%f ", unew[i]);
            }
            fclose(f);
            printf("Processor %d wrote its portion\n", p);
            // 5. process p sends the signal to process p+1 (if it exists).
            if (p != P - 1) {
                MPI_Send(&signal, 1, MPI_DOUBLE, p+1, tag, MPI_COMM_WORLD);
                printf("Processor %d sent signal to processor %d\n", p, p+1);
            }
        }
    }

    /* That's it */
    MPI_Finalize();
    exit(0);
}
```