FEDERAL STATE AUTONOMOUS EDUCATIONAL INSTITUTION OF HIGHER EDUCATION

ITMO UNIVERSITY

Report

MPI. Assignments $6 - 7$

Parallel algorithms for the analysis and synthesis of data

Performed by
Aleksandr Shirokov
J4133c
Accepted by
Petr Andriushchenko
Deadline: 20.12.21

St. Petersburg
2021

Contents

1 Assignments

1.1 Assignment 6. MPI. Retrieving information about the message attributes.

1.1.1 Formulation of the problem

1. Compile the example Assignment6.c in detail, run it and explain it.
2. Transform the program using the MPI_TAG field of the status structure in the condition.

1.1.2 Example of launch parameters and output. Detailed description of solution

Code for **assignment 6** is here.
Compilation example: mpic++ -o ./cpf/6.o Assignment6.c
Launch example: mpirun –oversubscribe -np 4 ./cpf/6.o

```
(base) aptmess@improfeo:~/ITMO/parallel_algorithms/HT/hw_mpi$ mpirun --oversubscribe -np 4 ./cpf/6.o
Process 0 recv 1 from process 1, 2from process 2
(base) aptmess@improfeo:~/ITMO/parallel_algorithms/HT/hw_mpi$ mpirun --oversubscribe -np 4 ./cpf/6.o
Process 0 recv 2 from process 2, 1from process 1
```

There could be only two results of program output

Let's move to the the code and explain how it works.

```cpp
#include <iostream>
#include <mpi.h>
using namespace std;
int main(int argc, char **argv)
{
    int rank, size, ibuf;
    MPI_Status status;
    float rbuf;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    ibuf = rank;
    rbuf = 1.0 * rank;
    if (rank == 1) MPI_Send(&ibuf, 1, MPI_INT, 0, 5, MPI_COMM_WORLD);
    if (rank == 2) MPI_Send(&rbuf, 1, MPI_FLOAT, 0, 5, MPI_COMM_WORLD);
    if (rank == 0) {
        MPI_Probe(MPI_ANY_SOURCE, 5, MPI_COMM_WORLD, &status);
        if (status.MPI_SOURCE == 1) {
            MPI_Recv(&ibuf, 1, MPI_INT, 1, 5, MPI_COMM_WORLD, &status);
            MPI_Recv(&rbuf, 1, MPI_FLOAT, 2, 5, MPI_COMM_WORLD, &status);
            cout << "Process 0 recv " << ibuf << " from process 1, " << rbuf << "from process 2\n";
        }
        else if (status.MPI_SOURCE == 2) {
            MPI_Recv(&rbuf, 1, MPI_FLOAT, 2, 5, MPI_COMM_WORLD, &status);
            MPI_Recv(&ibuf, 1, MPI_INT, 1, 5, MPI_COMM_WORLD, &status);
            cout << "Process 0 recv " << rbuf << " from process 2, " << ibuf << "from process 1\n";
        }
    }
    MPI_Finalize();
}
```

Assignment6 code

Firstly there is an initialization of parallel part using MPI_Init, after if rank of process is 1 then the int 1 will be send as a message and if rank of process is 2, then the float value 2.0 will be send as message. After we are going to main process 0 logic:

- MPI_Probe this function is waiting for message from any process with msgtag = 5 and wouldn't go next if the message doesn't come to process 0. Let's make it clear - function only understand that message come to process, but doesn't get it.
- After that if status.MPI_SOURCE == 1 so if first was message from process 1 then there is a print message that 1st process's message was quicklier, else - that the second was quicklier and the value from second process will be displayed first.

After I have transformed the problem using MPI_TAG field. Here are results:

```
(base) aptmess@improfeo:~/ITMO/parallel_algorithms/HT/hw_mpi$ mpic++ -o ./cpf/6.1.o Assignment6.1.c
(base) aptmess@improfeo:~/ITMO/parallel_algorithms/HT/hw_mpi$ mpirun --oversubscribe -np 4 ./cpf/6.1.o
Process 0 recv 1 from process 1, 2from process 2
(base) aptmess@improfeo:~/ITMO/parallel_algorithms/HT/hw_mpi$ mpirun --oversubscribe -np 4 ./cpf/6.1.o
Process 0 recv 2 from process 2, 1from process 1
(base) aptmess@improfeo:~/ITMO/parallel_algorithms/HT/hw_mpi$
```

Results are the same. Take a look at code

Code for **assignment 6.1** is here.
Compilation example: mpic++ -o ./cpf/6.1.o Assignment6.1.c
Launch example: mpirun –oversubscribe -np 4 ./cpf/6.1.o

```cpp
#include <iostream>
#include <mpi.h>
using namespace std;
int main(int argc, char **argv)
{
    int rank, size, ibuf, first_process_tag, second_process_tag;
    first_process_tag = 5;
    second_process_tag = 4;
    MPI_Status status;
    float rbuf;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    ibuf = rank;
    rbuf = 1.0 * rank;
    if (rank == 1) MPI_Send(&ibuf, 1, MPI_INT, 0, first_process_tag, MPI_COMM_WORLD);
    if (rank == 2) MPI_Send(&rbuf, 1, MPI_FLOAT, 0, second_process_tag, MPI_COMM_WORLD);
    if (rank == 0) {
        MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        if (status.MPI_TAG == first_process_tag) {
            MPI_Recv(&ibuf, 1, MPI_INT, 1, first_process_tag, MPI_COMM_WORLD, &status);
            MPI_Recv(&rbuf, 1, MPI_FLOAT, 2, second_process_tag, MPI_COMM_WORLD, &status);
            cout << "Process 0 recv " << ibuf << " from process 1, " << rbuf << "from process 2\n";
        }
        else if (status.MPI_TAG == second_process_tag) {
            MPI_Recv(&rbuf, 1, MPI_FLOAT, 2, second_process_tag, MPI_COMM_WORLD, &status);
            MPI_Recv(&ibuf, 1, MPI_INT, 1, first_process_tag, MPI_COMM_WORLD, &status);
            cout << "Process 0 recv " << rbuf << " from process 2, " << ibuf << "from process 1\n";
        }
    }
    MPI_Finalize();
}
```

Assignment6 part II code

Everything is more or less the same, but now we are expecting any tag in MPI_Probe function and processes 1 and 2 has different tags (5 and 4) and condition is also have changed (status.MPI_TAG). Program works correctly.

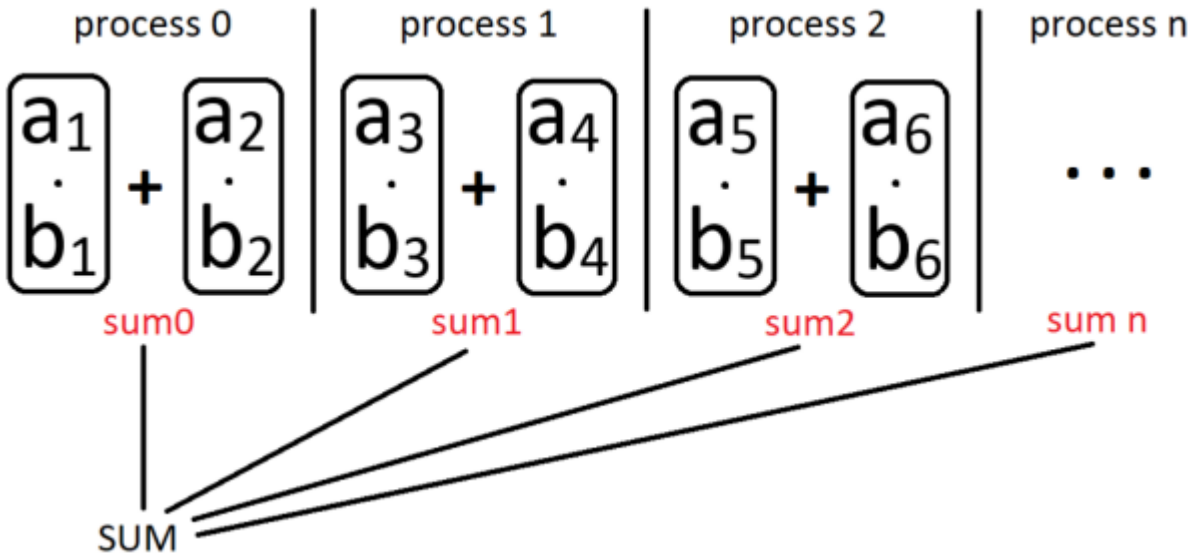
1.2 Assignment 7. MPI. Dot product of vectors.

1.2.1 Formulation of the problem

1. Write an MPI program that implements the dot product of two vectors distributed between processes.
2. Two vectors with a size of at least 1,000,000 elements are initialized at process 0 and filled with "1", then they are sent in equal parts to all processes.
3. Parts of vectors are scalar multiplied on each process, the result is sent to the root process and summed up.
4. The total is displayed.

Scalar product for two vectors $a = [a_1, \ldots, a_n]$ and $b = [b_1, \ldots, b_n]$ in n-dimensional space defined as:

$$a \cdot b = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + \ldots + a_n b_n$$



The algorithm of parallel dot product

1.2.2 Example of launch parameters and output. Detailed description of solution

Code for **assignment 7** is here.
Compilation example: mpic++ -o ./cpf/7.o Assignment7.c
Launch example: mpirun –oversubscribe -np 4 ./cpf/7.o 1000000

```
aptmess@improfeo: ~/ITMO/parallel_algorithms/HT/hw_mpi
(base) aptmess@improfeo:~/ITMO/parallel_algorithms/HT/hw_mpi$ mpic++ -o ./cpf/7.o Assignment7.c
(base) aptmess@improfeo:~/ITMO/parallel_algorithms/HT/hw_mpi$ mpirun --oversubscribe -np 10 ./cpf/7.o 10000000
sum=10000000
equal with array size: 1
(base) aptmess@improfeo:~/ITMO/parallel_algorithms/HT/hw_mpi$
```

```
(base) aptmess@improfeo:~/ITMO/parallel_algorithms/HT/hw_mpi$ mpic++ -o ./cpf/7.o Assignment7.c
(base) aptmess@improfeo:~/ITMO/parallel_algorithms/HT/hw_mpi$ mpirun --oversubscribe -np 3 ./cpf/7.o 9
4 8 4 7 10 3 1 4 1
7 6 6 3 2 8 10 7 7
sum=210
equal with array size: 0
(base) aptmess@improfeo:~/ITMO/parallel_algorithms/HT/hw_mpi$
```

Some testing of function not on only ones

Let's move to the the code and explain how it works.

```cpp
#include <ctime>
#include <stdio.h>
#include <cstdlib>

using namespace std;

int serial_dot(int x[], int y[], int n)
{
    /*
    serial product of two arrays

    a . b = Sum[a[i] * b[i], {i, 1, n}] = a1 * b1 + ... + an * bn;

    */
    int sum = 0;
    for (int i = 0; i < n; i++)
        sum = sum + x[i] * y[i];

    return sum;
}

int parallel_dot(int x[], int y[], int batch_size, int root)
{
    /*
    parallel dot with MPI_SUM as a reduce operation

    MPI_Reduce(
        sbuf - adress of start buffer for arguments
        OUT rbuf - adress of start buffer for results
        count - the amount of arguments for each process
        datatype - type of argument
        op - identifier of reduce operation
        root - main process which will take the result
        comm - communicator
    )
    */
    int local_dot, full_dot = 0;
    int serial_dot(int x[], int y[], int m);

    local_dot = serial_dot(x, y, batch_size);

    MPI_Reduce(&local_dot, &full_dot, 1, MPI_INT, MPI_SUM, root, MPI_COMM_WORLD);

    return full_dot;
}


int split_data_by_processes(int arr[], int batch[], int batch_size, int root)
{
    /*
    splitting arr between each process by batches with length=batch_size

    MPI_Scatter(
        sbuf - adress of start buffer for arguments in sending message
        scount - amount of elements in sending message
        stype - type of sending message
        OUT rbuf - adress of start buffer for requesting message
        rcount - amount of elements in requesting message
        rtype - type of elements in requesting message
        source - number of process, where data is collecting
        comm - communicator
    )
    */
    MPI_Scatter(arr, batch_size, MPI_INT, batch, batch_size, MPI_INT, root, MPI_COMM_WORLD);
    return 0;
}
```

```cpp
int main(int argc, char* argv[])
{
    int length_array = atoi(argv[1]);
    const int root = 0;

    MPI_Init(&argc, &argv);

    int rank, n, batch_size, dot;
    int *first_array, *second_array;

    MPI_Comm_size(MPI_COMM_WORLD, &n);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    batch_size = length_array / n;

    if (rank == root)
    {
        first_array = new int[length_array]();
        second_array = new int[length_array]();
        for (int i = 0; i < length_array; i++)
        {
            first_array[i] = 1;
            second_array[i] = 1;
        }
        if (length_array < 20)
        {
            for (int i = 0; i < length_array; i++)
            {
                cout << first_array[i] << ' ';
            }
            cout << endl;

            for (int i = 0; i < length_array; i++)
            {
                cout << second_array[i] << ' ';
            }
            cout << endl;
            }

    }

    int *first_batch = new int[batch_size]();
    int *second_batch = new int[batch_size]();
    split_data_by_processes(first_array, first_batch, batch_size, root);
    split_data_by_processes(second_array, second_batch, batch_size, root);
    dot = parallel_dot(first_batch, second_batch, batch_size, root);

    if (rank == root)
    {
        cout << "sum=" << dot << endl;
        cout << "equal with array size: " << (length_array == dot) << endl;
    }

    MPI_Finalize();
    return 0;
}
```

Assignment7 code

On picture in section Formulation of the problem there are a structure of algorithm - we should split our arrays into different processes, map some function inside processes such as SERIAL_DOT and collect (reduce) result in the main process 0 - this algorithm our program is doing. On the left picture there are three functions:

- SERIAL_DOT which calculate the dot product of two vectors;
- PARALLEL_DOT which run SERIAL_DOT function and after that with MPI_REDUCE function reduce results from local variable to SBUF variable - FULL_DOT.

- SPLIT_DATA_BY_PROCESSES which splitts the input array between each process by *batches* with the length of *batch_size*

Have to mentioned that for correct work of algorithm the amount of processes should be a divider of length of array. Batch size is $\frac{\text{length\_of\_array}}{\text{amount\_of\_processes}}$.

The main function is using this functions - firstly in root process 0 we initialize the arrya of length LENGTH_ARRAY and fill them 1, after that we initializing batch, split data for each process by batch size for every process, in each process serial dot is running and sending to main process, which collect the dot variable - sum of each serial dot in processes. Then root process show result and check that sum is equal to size of array (because each array is filled by 1). Program works correctly.

1.3 Appendix

The link to the sourse code which is placed on my github.