

FEDERAL STATE AUTONOMOUS EDUCATIONAL
INSTITUTION OF HIGHER EDUCATION

ITMO UNIVERSITY

Report

MPI. Assignments 10 — 11

Parallel algorithms for the analysis and synthesis of data

Performed by
Aleksandr Shirokov

J4133c

Accepted by
Petr Andriushchenko

Deadline: 21.12.21

St. Petersburg

2021

Contents

1 Assignments

1.1 Assignment 10. MPI. Sending and receiving messages without blocking. Ring exchange using non-blocking operations.

1.1.1 Formulation of the problem

Complete the program ASSIGNMENT10.C. Compile and run it.
Study the code carefully and explain how it works.

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

Code for assignment 10 is [here](#).

Compilation example: `MPIC++ -O ./CPF/10.O ASSIGNMENT10.C`

Launch example: `MPIRUN -OVERSUBSCRIBE -NP 10 ./CPF/10.O`

```
aptmess@improfeo: ~/ITMO/parallel_algorithms/HT/hw_mpi
(base) aptmess@improfeo:~/ITMO/parallel_algorithms/HT/hw_mpi$ mpirun --oversubscribe -np 10 ./cpf/10.o
9 (previous) -> 0 (current) -> 1 (next)
0 (previous) -> 1 (current) -> 2 (next)
1 (previous) -> 2 (current) -> 3 (next)
2 (previous) -> 3 (current) -> 4 (next)
3 (previous) -> 4 (current) -> 5 (next)
4 (previous) -> 5 (current) -> 6 (next)
5 (previous) -> 6 (current) -> 7 (next)
6 (previous) -> 7 (current) -> 8 (next)
7 (previous) -> 8 (current) -> 9 (next)
8 (previous) -> 9 (current) -> 0 (next)
(base) aptmess@improfeo:~/ITMO/parallel_algorithms/HT/hw_mpi$ _
```

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

```
1  #include <iostream>
2  #include "mpi.h"
3
4  using namespace std;
5  int main(int argc, char **argv)
6  {
7      int rank, size, prev, next;
8      int buf[2];
9      MPI_Init(&argc, &argv);
10     MPI_Request reqs[4];
11     MPI_Status stats[4];
12     MPI_Comm_size(MPI_COMM_WORLD, &size);
13     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
14     prev = rank - 1;
15     next = rank + 1;
16     if (rank == 0) prev = size - 1;
17     if (rank == size - 1) next = 0;
18     MPI_Irecv(&buf[0], 1, MPI_INT, prev, 5, MPI_COMM_WORLD, &reqs[0]);
19     MPI_Irecv(&buf[1], 1, MPI_INT, next, 6, MPI_COMM_WORLD, &reqs[1]);
20     MPI_Isend(&rank, 1, MPI_INT, prev, 6, MPI_COMM_WORLD, &reqs[2]);
21     MPI_Isend(&rank, 1, MPI_INT, next, 5, MPI_COMM_WORLD, &reqs[3]);
22     MPI_Waitall(4, reqs, stats);
23
24     //Your code here.
25     //Here you need to display the number of the current process, and what it receives from the previous and next processes.
26     cout << buf[0] << " (previous)" << " -> " << rank << " (current)" << " -> " << buf[1] << " (next)" << '\n' << endl;
27     MPI_Finalize();
28 }
```

Assignment 10

The overall goal of the program is that all processes exchange messages with their nearest neighbors (on the left - previous, on the right - next) in accordance with the topology of the ring. With `MPI_WAITALL` the execution of the process is blocked until all exchange operations on the specified `REQS` identifiers (lines 18-21) are completed and if the error exists in this operations, then the error field in the `STATS` array elements will be set to the appropriate value. In lines 18–21 there are operations `MPI_Irecv` and `MPI_Isend` which are equal to previous functions `MPI_RECV` and `MPI_SEND` but in this functions the return from the function occurs immediately after the initialization of the receiving/transmitting process without waiting for the receipt/ processing of the entire message, so we can solve the problem with blocking operations in `MPI_SEND` and `MPI_RECV`. In this lines the process waiting for their nearest neighbours and save information in int array `BUF` and send information about yourself's rank to previous and next. The result is displayed on screens - ring topology works.

1.2 Assignment 11.MPI. Combined reception and transmission of messages.

1.2.1 Formulation of the problem

Based on ASSIGNMENT 10, write a program for ring topology exchange using the **MPI_Sendrecv()** function.

In situations where you need to exchange data between processes, it is safer to use the overlaid **MPI_Sendrecv** operation. The **MPI_Sendrecv** function combines the execution of the send and receive operations. Both operations use the same communicator, but message IDs may differ. The location of the received and transmitted data in the address space of the process should not overlap. The data sent can be of different types and lengths.

In cases when it is necessary to exchange data of the same type with replacement of the sent data with the received ones, it is more convenient to use the **MPI_Sendrecv_replace** function. In this operation, the data sent from the buf array is replaced with the received data.

The special address **MPI_PROC_NULL** can be used for source and dest in data transfer operations. Communication operations with such an address do nothing. The use of this address is convenient instead of using logical constructs to analyze the conditions to send / read a message or not.

```
int MPI_Sendrecv (
    • void *sendbuf - the address of the data to be sent
    • int sendcount - the number of sent variables
    • MPI_Datatype sendtype - the type of data being sent
    • int dest - destination rank
    • int sendtag - the tag of the sent message
    • void *recvbuf
    • int recvcount - is the number of received data
    • MPI_Datatype recvtype - the type of data being received
    • int source - from whom the message is received
    • int recvtag - received message tag
    • MPI_Comm comm - MPI_Comm comm
    • MPI_Status *status - status
)
```

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

Code for **assignment 11** is [here](#).

Compilation example: `MPIC++ -O ./CPF/11.O ASSIGNMENT11.C`

Launch example: `MPIRUN -OVERSUBSCRIBE -NP 11 ./CPF/11.O`

```
(base) aptmess@improfeo:~/ITM0/parallel_algorithms/HT/hw_mpi$ mpirun --oversubscribe -np 11 ./cpf/11.o
6 (previous) -> 7 (current) -> 8 (next)
1 (previous) -> 2 (current) -> 3 (next)
2 (previous) -> 3 (current) -> 4 (next)
3 (previous) -> 4 (current) -> 5 (next)
4 (previous) -> 5 (current) -> 6 (next)
5 (previous) -> 6 (current) -> 7 (next)
0 (previous) -> 1 (current) -> 2 (next)
7 (previous) -> 8 (current) -> 9 (next)
10 (previous) -> 0 (current) -> 1 (next)
8 (previous) -> 9 (current) -> 10 (next)
9 (previous) -> 10 (current) -> 0 (next)
(base) aptmess@improfeo:~/ITM0/parallel_algorithms/HT/hw_mpi$ _
```

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

```
1  #include <iostream>
2  #include "mpi.h"
3
4  using namespace std;
5  int main(int argc, char **argv)
6  {
7      int rank, size, prev, next;
8      int buf[2];
9      MPI_Init(&argc, &argv);
10     MPI_Status stats[2];
11     MPI_Comm_size(MPI_COMM_WORLD, &size);
12     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
13     prev = rank - 1;
14     next = rank + 1;
15     if (rank == 0) prev = size - 1;
16     if (rank == size - 1) next = 0;
17     MPI_Sendrecv(&rank, 1, MPI_INT, next, 5, &buf[0], 1, MPI_INT, prev, 5, MPI_COMM_WORLD, &stats[1]);
18     MPI_Sendrecv(&rank, 1, MPI_INT, prev, 6, &buf[1], 1, MPI_INT, next, 6, MPI_COMM_WORLD, &stats[0]);
19
20     //Your code here.
21     //Here you need to display the number of the current process, and what it receives from the previous and next processes.
22     cout << buf[0] << " (previous)" << " -> " << rank << " (current)" << " -> " << buf[1] << " (next)" << '\n' << endl;
23     MPI_Finalize();
24 }
```

Assignment 11

The main goal of program is the same as in Assignment 10 and the results is equally the same, but in assignment 11 we are using function `MPI_Sendrecv()` due to syntax in previous subsection which is sending and try to receive message in the same function. On line 17 this function process sends a message with current process's rank to next nearest process and receive rank of the previous nearest process. On the line 18 on the contrary - sends a message with rank to previous nearest process and try to receive rank from nearest next process. The program works correctly.

1.3 Appendix

The link to the source code which is placed on my [github](#).