Saint Petersburg National Research University of Information Technologies, Mechanics and Optics (ITMO University)

REPORT

about laboratory works

Assignment 6.
Assignment 7.
Assignment 8.

Student Pogrebnoy D.A. j4132c

Saint-Petersburg, 2021

ASSIGNMENT 6.

Task

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

Implementation

Source code and data gathered are available on https://github.com/DmitryPogrebnoy/Parallel-algorithms-of-data-analysis-and-synthesis/tree/master/OmpiTasks/Task6

The description of the code is described in the comments.

Assignment6.cpp

OmpiTasks > Task6 > Assignment6.cpp > main(int, char **)

```cpp
#include <iostream>
#include <mpi.h>
using namespace std;
int main(int argc, char **argv)
{
    int rank, size, ibuf;
    MPI_Status status;
    float rbuf;
    // Initialize the MPI environment
    MPI_Init(&argc, &argv);
    // Get the number of processes associated with the communicator
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    // Get the rank of the calling process
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    ibuf = rank;
    rbuf = 1.0 * rank;
    // Process 1 send int with tag 5 to other processes
    if (rank == 1) MPI_Send(&ibuf, 1, MPI_INT, 0, 5, MPI_COMM_WORLD);
    // Process 2 send int with tag 5 to other processes
    if (rank == 2) MPI_Send(&rbuf, 1, MPI_FLOAT, 0, 5, MPI_COMM_WORLD);
    if (rank == 0) {
        // Checks if there is new incoming message with tag 5 (blocking wait)
        MPI_Probe(MPI_ANY_SOURCE, 5, MPI_COMM_WORLD, &status);
        // If there is first new message with source process num 1
        if (status.MPI_SOURCE == 1) {
            // First recive message from process 1
            MPI_Recv(&ibuf, 1, MPI_INT, 1, 5, MPI_COMM_WORLD, &status);
            // Second recive message from process 1
            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";
        }
        // If there is first new message with source process num 2
        else if (status.MPI_SOURCE == 2) {
            // First recive message from process 2
            MPI_Recv(&rbuf, 1, MPI_FLOAT, 2, 5, MPI_COMM_WORLD, &status);
            // First recive message from process 1
            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();
}
```

Output example:

And rewritten version with tags looks like this:

Assignment6_new.cpp

OmpiTasks > Task6 > Assignment6_new.cpp > ...

```cpp
#include <iostream>
#include <mpi.h>
using namespace std;
int main(int argc, char **argv)
{
    int rank, size, ibuf;
    MPI_Status status;
    float rbuf;
    // Initialize the MPI environment
    MPI_Init(&argc, &argv);
    // Get the number of processes associated with the communicator
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    // Get the rank of the calling process
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    ibuf = rank;
    rbuf = 1.0 * rank;
    // Process 1 send int with tag 1 to other processes
    if (rank == 1) MPI_Send(&ibuf, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
    // Process 2 send int with tag 2 to other processes
    if (rank == 2) MPI_Send(&rbuf, 1, MPI_FLOAT, 0, 2, MPI_COMM_WORLD);
    if (rank == 0) {
        // Checks if there is new incoming message (blocking wait)
        MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        // If there is first new message with tag 1
        if (status.MPI_TAG == 1) {
            // First recive message from process 1 with tag 1
            MPI_Recv(&ibuf, 1, MPI_INT, 1, 1, MPI_COMM_WORLD, &status);
            // First recive message from process 2 with tag 2
            MPI_Recv(&rbuf, 1, MPI_FLOAT, 2, 2, MPI_COMM_WORLD, &status);
            cout << "Process 0 recv " << ibuf << " from process 1, " << rbuf << " from process 2\n";
        }
        // If there is first new message with tag 2
        else if (status.MPI_TAG == 2) {
            // First recive message from process 2 with tag 2
            MPI_Recv(&rbuf, 1, MPI_FLOAT, 2, 2, MPI_COMM_WORLD, &status);
            // First recive message from process 1 with tag 1
            MPI_Recv(&ibuf, 1, MPI_INT, 1, 1, MPI_COMM_WORLD, &status);
            cout << "Process 0 recv " << rbuf << " from process 2, " << ibuf << " from process 1\n";
        }
    }
    MPI_Finalize();
}
```

ASSIGMENTS 7.

Task

Write an MPI program that implements the dot product of two vectors distributed between processes. Two vectors with a size of at least 1,000,000 elements are initialized at process zero and

filled with “1”, then they are sent in equal parts to all processes. Parts of vectors are scalar multiplied on each process, the result is sent to the root process and summed up. The total is displayed.

Implementation

Source code and data gathered are available on https://github.com/DmitryPogrebnoy/Parallel-algorithms-of-data-analysis-and-synthesis/blob/master/OmpiTasks/Task7/Task7.cpp

The description of the code is described in the comments. In the main process, we initialize the value of the number of array elements and the arrays themselves. Next, we send the number of elements in arrays to all threads. After that, we send a piece of two source arrays to each thread to perform the scalar product of these pieces. After that, the values of each pieces are summed up in the main process and the answer is output. It is important that the length of arrays is entirely divided by the number of processes.

Output example:

ASSIGNMENT 8.

Task

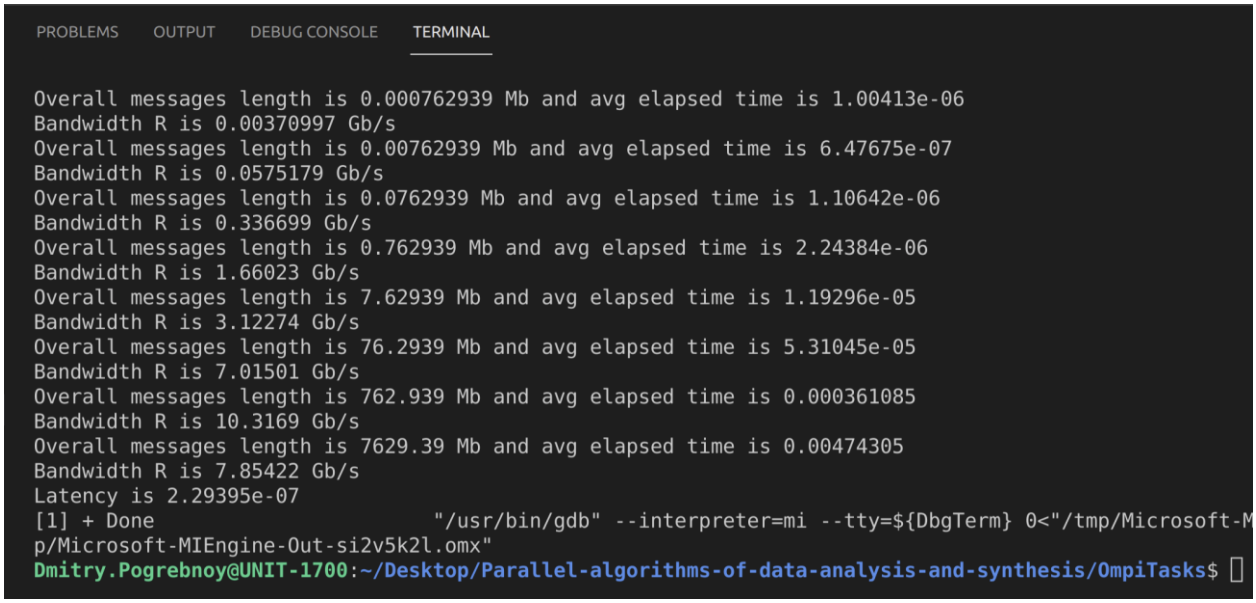
Write an MPI program in which two processes exchange messages, measure the time per exchange iteration, and determine the dependence of the exchange time on the message length. Determine the latency and maximum achievable bandwidth of the communication network. Print the message length in bytes and the throughput in MB/s to the console. Change the length of the message in a loop starting from 1 element and increase to 1,000,000 elements, increasing by 10 times at each iteration.

Implementation

Source code and data gathered are available on https://github.com/DmitryPogrebnoy/Parallel-algorithms-of-data-analysis-and-synthesis/blob/master/OmpiTasks/Task8/Task8.cpp .

The description of the code is described in the comments. The program uses two processes. In the main process, we initialize an array with the required length and forward this array 100 times to another process and back, while measuring the time. After that, we measure the delay in sending the message.

Output example:



So maximum achievable bandwidth is ~10 GB/s.