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

REPORT

about laboratory work № 1

« Point-to-Point Communications in MPI»

Student
Nerinovsky Arseny **_M41331_**
(Surname, initials) Group

Saint-Petersburg, 2018

Report

June 29, 2019

1 GOAL OF LABORATORY WORK

The goal of this task was to show the working of MPI_SEND, MPI_Ssend, MPI_Bsend, MPI_Rsend) and conclude the performance analysis of each of them.

2 TASK DEFINITION

We develop a simple application which sends few bytes of data from one process to another one.

3 BRIEF THEORY

The Open MPI Project is an open source Message Passing Interface implementation that is developed and maintained by a consortium of academic, research, and industry partners. Open MPI is therefore able to combine the expertise, technologies, and resources from all across the High Performance Computing community in order to build the best MPI library available. Open MPI offers advantages for system and software vendors, application developers and computer science researchers.

MPI has a number of different "send modes." These represent different choices of buffering (where is the data kept until it is received) and synchronization (when does a send complete). In the following, I use "send buffer" for the user-provided buffer to send.

- MPI_Send MPI_Send will not return until you can use the send buffer. It may or may not block (it is allowed to buffer, either on the sender or receiver side, or to wait for the matching receive).
- MPI_Bsend May buffer; returns immediately and you can use the send buffer. A late add-on to the MPI specification. Should be used only when absolutely necessary.
- MPI_Ssend will not return until matching receive posted
- MPI_Rsend May be used ONLY if matching receive already posted. User responsible for writing a correct program.
- MPI_Isend Nonblocking send. But not necessarily asynchronous. You can NOT reuse the send buffer until either a successful, wait/test or you KNOW that the message has been received (see MPI_Request_free). Note also that while the I refers to immediate, there is no performance requirement on MPI_Isend. An immediate send must return to the user without requiring a matching receive at the destination.

An implementation is free to send the data to the destination before returning, as long as the send call does not block waiting for a matching receive. Different strategies of when to send the data offer different performance advantages and disadvantages that will depend on the application - MPI_Ibsend buffered nonblocking - MPI_Issend Synchronous nonblocking. Note that a Wait/Test will complete only when the matching receive is posted. - MPI_Irsend As with MPI_Rsend, but nonblocking.

Note that "nonblocking" refers ONLY to whether the data buffer is available for reuse after the call. No part of the MPI specification, for example, mandates concurent operation of data transfers and computation.

4 ALGORITHM (METHOD) of IMPLEMENTATION

[1]:
```
%cat hello.c
```

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

#include <unistd.h>
#include <stdbool.h>

#ifndef MSG_LEN
# define MSG_LEN 32
#endif


#ifndef SEND_FN
# define SEND_FN MPI_Send
#endif

#if !defined(SYNC) && !defined(SEND_RECV) && !defined(ASYNC)
# define SYNC
#endif




void rand_str(char *str, size_t len)
{
        for(size_t i = 0; i < len - 1; ++i) {
                str[i] = rand() % 26 + 64;
        }
        str[len] = 0;
}

int main(int argc,char **argv)
{
        int rank, size;
```

```
        MPI_Init(&argc,&argv);
        MPI_Comm_rank(MPI_COMM_WORLD,&rank);
        MPI_Comm_size(MPI_COMM_WORLD,&size);

        MPI_Request req;
        MPI_Status status;
        bool wait = false;
        srand(rank+10);

        char buf[MSG_LEN], rbuf[MSG_LEN];

#ifdef SYNC
        printf("SYNC\n");
#endif

#ifdef SEND_RECV
        printf("SEND_RECV\n");
#endif

#ifdef ASYNC
        printf("ASYNC\n");
#endif

        for(size_t i = 0; i < 10; ++i) {

#ifdef SYNC
                if( (i + rank) % 2 == 0 ) {
                        MPI_Recv(buf, MSG_LEN, MPI_CHAR, !rank, 0,
MPI_COMM_WORLD,MPI_STATUS_IGNORE);
                        printf("%*sRECV(%d) : %s\n",rank*44, " ", rank, buf);
                } else {
                        rand_str(buf, MSG_LEN);
                        printf("%*sSEND(%d) : %s\n", rank*44, " ", rank, buf);
                        SEND_FN(buf, MSG_LEN, MPI_CHAR, !rank, 0,
MPI_COMM_WORLD);
                }
#endif

#ifdef SEND_RECV
                rand_str(buf, MSG_LEN);
                printf("%*sSEND(%d) : %s\n", rank*44, " ", rank, buf);
                MPI_Sendrecv(buf, MSG_LEN, MPI_CHAR, !rank, 0,
                                                rbuf, MSG_LEN, MPI_CHAR, !rank,
0,
                                                        MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
                printf("%*sRECV(%d) : %s\n",rank*44, " ", rank, rbuf);
#endif
```

```
#ifdef ASYNC

                if( (i + rank) % 2 == 0 ) {
                        if(wait) {
                                MPI_Wait(&req, &status);
                                MPI_Irecv(buf, MSG_LEN, MPI_CHAR, !rank, 0,
MPI_COMM_WORLD, &req);
                                wait = true;
                                printf("%*sRECV(%d) : %s\n",rank*44, " ", rank,
buf);
                        }
                } else {
                        rand_str(buf, MSG_LEN);
                        printf("%*sSEND(%d) : %s\n", rank*44, " ", rank, buf);
                        if(wait) MPI_Wait(&req, &status);
                        MPI_Isend(buf, MSG_LEN, MPI_CHAR, !rank, 0,
MPI_COMM_WORLD, &req);
                        wait = true;
                }
#endif
                // sleep(rand() % 5);
        }

        MPI_Finalize();
}
```

5 RESULT AND EXPERIMENTS

[2]:

```python
import subprocess
import os


def compile(*defs, **defskw):
    args = [f"-D{k}" for k in defs] + [f"-D{k}={v}" for k, v in defskw.items()]
    _cmd = 'mpicc -o hello hello.c'.split() + args
    # print(' '.join(_cmd))
    cmd = subprocess.run(_cmd, stdout=subprocess.PIPE, stderr=subprocess.PIPE)

    if(cmd.stdout): print('cmd.stdout', cmd.stdout)
    if(cmd.stderr): print('cmd.stderr', cmd.stderr)

def run(env=None):
    cmd = subprocess.run('mpiexec -np 2 ./hello'.split(), stdout=subprocess.
↪PIPE, stderr=subprocess.PIPE, env=env)
    if(cmd.stderr): print('cmd.stderr', cmd.stderr)
```

```

```

5.1 Message length

Lets analyze the effect the message lenght has on the perofmance of the applocaion.

[3]:
```
for i in range(8):
    compile(MSG_LEN=10**i)
    print(f"Using message length {10**i}", end="\n\t")
    %timeit run()
    print()
```

```
Using message length 1
        11.1 ms ś 218 ţs per loop (mean ś std. dev. of 7 runs, 100 loops each)

Using message length 10
        11.1 ms ś 222 ţs per loop (mean ś std. dev. of 7 runs, 100 loops each)

Using message length 100
        11.3 ms ś 249 ţs per loop (mean ś std. dev. of 7 runs, 100 loops each)

Using message length 1000
        11.1 ms ś 343 ţs per loop (mean ś std. dev. of 7 runs, 100 loops each)

Using message length 10000
        13.3 ms ś 481 ţs per loop (mean ś std. dev. of 7 runs, 100 loops each)

Using message length 100000
        27.6 ms ś 1.6 ms per loop (mean ś std. dev. of 7 runs, 10 loops each)

Using message length 1000000
        166 ms ś 6.05 ms per loop (mean ś std. dev. of 7 runs, 10 loops each)

Using message length 10000000
        5.46 ms ś 260 ţs per loop (mean ś std. dev. of 7 runs, 100 loops each)
```

5.2 MPI send method

Lets analyze the effect the send method has on the perofmance of the application.

[4]:
```
for snd in 'MPI_Rsend MPI_Ssend MPI_Send'.split():

    compile(MSG_LEN=10**8, SEND_FN=snd)

    print(f"Using {snd} as send function", end="\n\t")
    %timeit run()
    print()
```

```
Using MPI_Rsend as send function
        5.37 ms ś 232 ţs per loop (mean ś std. dev. of 7 runs, 100 loops each)

Using MPI_Ssend as send function
        5.72 ms ś 378 ţs per loop (mean ś std. dev. of 7 runs, 100 loops each)

Using MPI_Send as send function
        5.66 ms ś 204 ţs per loop (mean ś std. dev. of 7 runs, 100 loops each)
```

5.3 SYNC vs ASYNC vs SENDRECV

Lets analyze the performanc impact of aync and sendrecv message passing.

[5]:
```
for snd in 'SYNC ASYNC SEND_RECV'.split():

    compile(snd, MSG_LEN=10**8)

    print(f"Using {snd}", end="\n\t")
    %timeit run()
    print()
```

```
Using SYNC
        5.94 ms ś 154 ţs per loop (mean ś std. dev. of 7 runs, 100 loops each)

Using ASYNC
        5.77 ms ś 90.2 ţs per loop (mean ś std. dev. of 7 runs, 100 loops each)

Using SEND_RECV
        5.91 ms ś 165 ţs per loop (mean ś std. dev. of 7 runs, 100 loops each)
```

6 CONCLUSION

The best performance is likely if you can write your program so that you could use just MPI_Ssend for larger data while for smaller MPI)Send performs better because for larger data MPI_Ssend can completely avoid buffering data. While MPI_Send allows the MPI implementation the maximum flexibility in choosing how to deliver your data. Use MPI_Bsend only when it is too inconvienent to use MPI_Isend as MPI_Bsend returns the buffer immediately. The remaining routines, MPI_Rsend, MPI_Issend, etc., are rarely used but may be of value in writing system-dependent message-passing code entirely within MPI.