Лабораторная работа №6:
Коллективные операции в MPI

ИКС ИБ
Б20-505
Барабанов Андрей

2022

Содержание

1 Рабочая среда

- Модель процессора: Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz
- Объём оперативной памяти: 16,0 ГБ
- Тип оперативной памяти: DDR4
- Версия операционной системы: ManjaroLinux 22.0.0
- Разрядность операционной системы: x86_64
- Среда разработки: GCC 12.2.0
- Версия OpenMPI: 4.1.4
- Версия OpenMP: 4.5

```
[bar1k@manjaro ~]$ lscpu
Архитектура:             x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         39 bits physical, 48 bits virtual
  Порядок байт:          Little Endian
CPU(s):                  12
  On-line CPU(s) list:   0-11
ID прроизводителя:       GenuineIntel
  Имя модели:            Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz
    Семейство ЦПУ:       6
    Модель:              165
    Thread(s) per core:  2
    Ядер на сокет:       6
    Сокетов:             1
    Степпинг:            2
    CPU(s) scaling MHz:  18%
    CPU max MHz:         5000,0000
    CPU min MHz:         800,0000
    BogoMIPS:            5202,65
```

```
[bar1k@manjaro ~]$ free -h
               total        used        free      shared  buff/cache   available
Mem:            15Gi       2,2Gi        11Gi       132Mi       2,2Gi        12Gi
Swap:             0B          0B          0B
```

```
[bar1k@manjaro ~]$ cat /etc/lsb-release
DISTRIB_ID=ManjaroLinux
DISTRIB_RELEASE=22.0.0
DISTRIB_CODENAME=Sikaris
DISTRIB_DESCRIPTION="Manjaro Linux"
```

```
[bar1k@manjaro ~]$ gcc --version
gcc (GCC) 12.2.0
Copyright (C) 2022 Free Software Foundation, Inc.
Это свободно распространяемое программное обеспечение. Условия копирования
приведены в исходных текстах.

  Без гарантии каких-либо качеств, включая
коммерческую ценность и применимость для каких-либо целей.
```

```
[bar1k@manjaro lab5]$ mpirun --version
mpirun (Open MPI) 4.1.4

Report bugs to http://www.open-mpi.org/community/help/
```

```
[bar1k@manjaro ~]$ echo | cpp -fopenmp -dM | grep -i open
#define _OPENMP 201511
```

2 Описание хода работы

В данной лабораторной работе необходимо было разработать алгоритм сортировки Шелла с помощью технологии **MPI** и сравнить результаты с лабораторной работой №3, где этот алгоритм разрабатывался с помощью технологии **OpenMP**.

Пусть топология коммуникационной сети имеет вид N-мерного гиперкуба, то есть количество процессов равно $p = 2^N$.

Действия алгоритма состоят в следующем:

1. Первый этап: распредление массива на p потоках и выполнение на каждом из них сортировки Шелла. Распредление массива между p потоками реализовывается с помощью процедуры **MPI_Scatter(array, chunk_size, MPI_INTEGER, chunk, chunk_size, MPI_INTEGER, 0, MPI_COMM_WORLD)**.

2. Второй этап: выполнение операции "compare-split"(сравнить и разделить) для кажой пары процессоров в гиперкубе. Формирование пар процессоров происходит по правилу - на кажой итерации **i, 0 <= i < N)**, парными становятся процессоры, у которых различие в битовых представлении их номеров имеется только в позиции **N - i - 1**. Выполнение операции "compare-split"(сравнить и разделить) реализовывается с помощью процедуры **MPI_Sendrecv(chunk, chunk_size, MPI_INTEGER, pair, n, tmp_chunk, chunk_size, MPI_INTEGER, pair, n, MPI_COMM_WORLD, MPI_STATUS_IGNORE)**. Данная процедура позволяет избежать тупиковых ситуаций (deadlock).

3. Третий этап: объединение массива с p потоков в главный. Распредление массива между p потоками реализовывается с помощью процедуры **MPI_Gather(chunk, chunk_size, MPI_INTEGER, array, chunk_size, MPI_INTEGER, 0, MPI_COMM_WORLD)**.

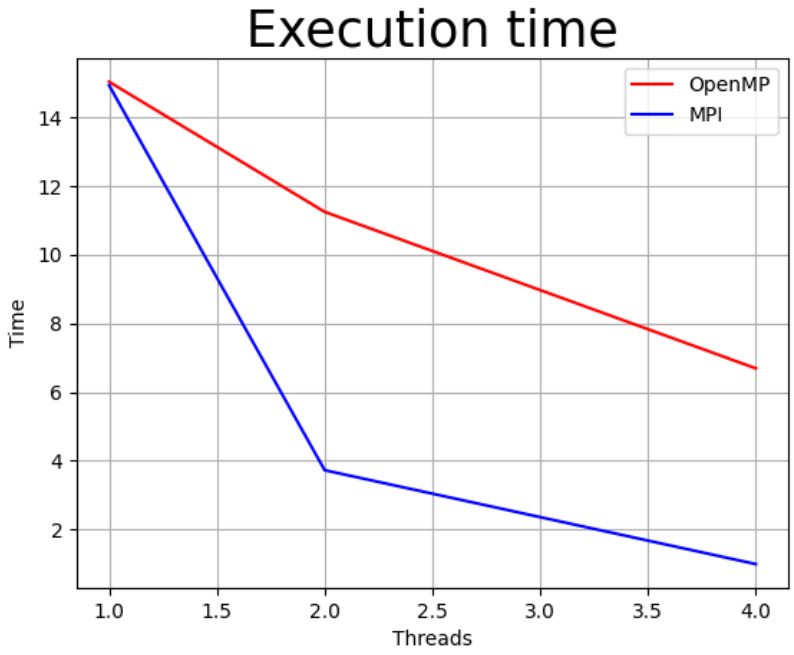
Так как число потоков, которые будут обрабатывать массив определяется на этапе выполнения команды **mpirun -np n -q ./prog5**, где **n** — количество потоков, то для получения экспериментальных результатов был написан bash-скрипт с циклами.

Для усреднения полученных результатов было расссмотрено 10 случайных массивов, которые сохрнялись файл для получения справедливых резуллътатов.

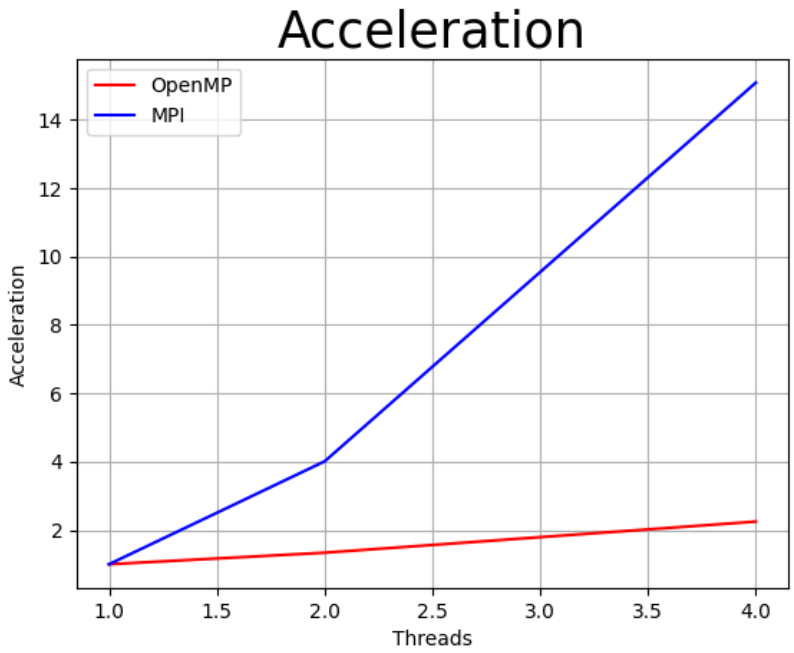
Графики сравнения времени, ускорения и эффективности двух различных технологий параллелного программирования были построены с помощью библиотеки Python **matplotlib**.

3 Графики

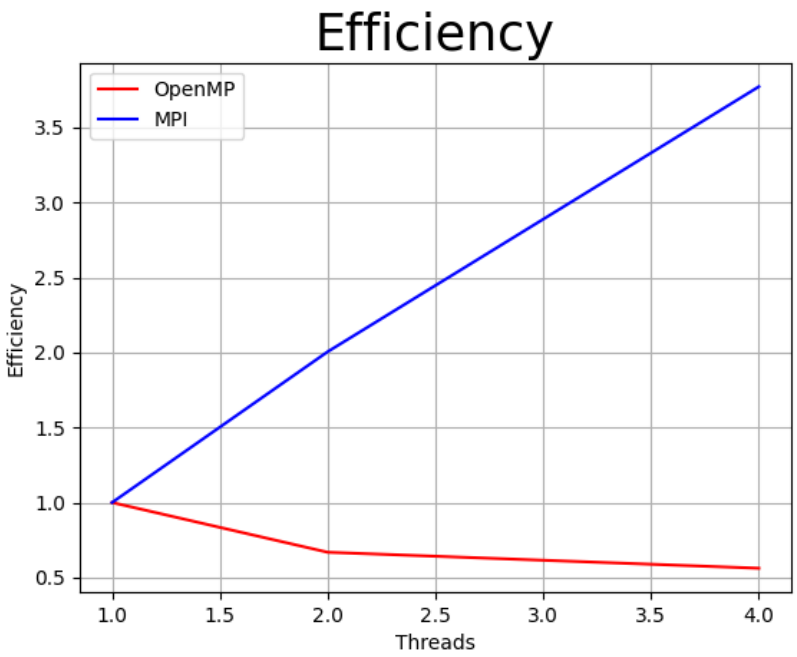
3.1 Время



3.2 Ускорение



3.3 Эффективность



4 Программные коды

4.1 build.sh

```
#!/usr/bin/zsh

ARRAY_SIZE=100000
THREADS_NUMBER=4
TESTS_NUMBER=5

echo "Compiling"
mpicc -o prog6 -lm src/lab6.c
gcc -fopenmp -o prog3 src/lab3.c

echo "Working OpenMP..."
./prog3 $TESTS_NUMBER $THREADS_NUMBER $ARRAY_SIZE

echo "Working MPI..."
for ((i = 1; i <= $TESTS_NUMBER; i++))
do
    echo "Creating array #$i"
    python3 src/array.py $ARRAY_SIZE

    for ((n = 1; n <= $THREADS_NUMBER; n *= 2))
    do
        echo "Number of theads: $n"
        mpirun -nq $n ./prog6 $ARRAY_SIZE
    done
done

python3 src/plot.py $TESTS_NUMBER $THREADS_NUMBER

# TODO: report in Latex

echo "Deleting"
rm prog3 prog6 array.txt openmp.txt mpi.txt
```

4.2 array.py

```
from sys import argv
from random import randint

if __name__ == '__main__':
    size = int(argv[1])

    file = open('array.txt', 'w')
    for _ in range(size):
        file.write(str(randint(0, size)) + ' ')
    file.write('\n')
    file.close()
```

4.3 lab3.c

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <time.h>

int main(int argc, char **argv) {
    int tests_number = atoi(argv[1]);
    int threads_number = atoi(argv[2]);

    int array_size = atoi(argv[3]);
    int *array = (int *) malloc(array_size * sizeof(int));
    int *tmp_array = (int *) malloc(array_size * sizeof(int));

    FILE *fd = fopen("openmp.txt", "w");

    for (int t = 0; t < tests_number; t++) {

        srand(time(NULL));
        for (int i = 0; i < array_size; i++)
            array[i] = rand() % array_size;

        for (int threads = 1; threads <= threads_number; threads *= 2) {

            for (int i = 0; i < array_size; i++)
                tmp_array[i] = array[i];

            double start_time = omp_get_wtime();

            for (int step = array_size / 2; step > 0; step /= 2) {
#pragma omp parallel for num_threads(threads) shared(tmp_array, array_size, step
                for (int i = step; i < array_size; i++)
                    for (int j = i - step; j >= 0; j -= step)
                        if (tmp_array[j] > tmp_array[j + step]) {
                            int tmp = tmp_array[j];
                            tmp_array[j] = tmp_array[j + step];
                            tmp_array[j + step] = tmp;
                        }
            }

            double end_time = omp_get_wtime();

            fprintf(fd, "%lf ", (end_time - start_time));
        }
    }
```

```
    fclose(fd);
    free(array);
    free(tmp_array);

    return 0;
}
```

4.4 lab6.c

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


int main(int argc, char **argv) {
    MPI_Init(&argc, &argv);

    // Initialization of process rank and number of threads.
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Checking number of threads.
    if (!rank)
        if ((size & (size - 1)) != 0) {
            fprintf(stderr, "Error! The number of threads must be a power of 2.\
            MPI_Abort(MPI_COMM_WORLD, MPI_ERR_TOPOLOGY);
        }

    // Checking size of array.
    int array_size = atoi(argv[1]);
    if (!rank)
        if (array_size % size != 0) {
            fprintf(stderr, "Error! The size of the array must be a multiple of
            MPI_Abort(MPI_COMM_WORLD, MPI_ERR_SIZE);
        }

    // Allocation of memory for array.
    int *array;
    if (!rank) {
        array = (int *) malloc(array_size * sizeof(int));
        if (!array) {
            fprintf(stderr, "Error! Failed to allocate memory.\n");
            MPI_Abort(MPI_COMM_WORLD, MPI_ERR_NO_MEM);
        }
    }

    // Initialization of values in an array.
    if (!rank) {
        FILE *fd = fopen("array.txt", "r");
        for (int i = 0; i < array_size; i++)
```

```
        fscanf(fd, "%d", &array[i]);
    fclose(fd);

}

// Allocation of memory for chunk of array.
int chunk_size = array_size / size;
int *chunk = (int *) malloc(chunk_size * sizeof(int));
int *tmp_chunk = (int *) malloc(chunk_size * sizeof(int));
int *new_chunk = (int *) malloc(chunk_size * sizeof(int));
if (!chunk || !tmp_chunk || !new_chunk) {
    fprintf(stderr, "Error! Failed to allocate memory.\n");
    MPI_Abort(MPI_COMM_WORLD, MPI_ERR_NO_MEM);
}

// Start of timing.
double start_time, end_time;
if (!rank)
    start_time = MPI_Wtime();

// Disstribution of array by threads.
MPI_Scatter(array, chunk_size, MPI_INTEGER, chunk, chunk_size, MPI_INTEGER,

// Shellsort for chunk of array.
for (int step = chunk_size / 2; step > 0; step /= 2)
    for (int i = step; i < chunk_size; i++)
        for (int j = i - step; j >= 0; j -= step)
            if (chunk[j] > chunk[j + step]) {
                int tmp = chunk[j];
                chunk[j] = chunk[j + step];
                chunk[j + step] = tmp;
            }

// Compare-split operation.
int power = log2(size);
for (int n = 0; n < power; n++) {
    // Initialization of paired process.
    int pair = rank ^ (1 << (power - n - 1));

    // Sending and receiving chunks between paired processes.
    MPI_Sendrecv(chunk, chunk_size, MPI_INTEGER, pair, n, tmp_chunk, chunk_s
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    // Merging paired chunks after comparison.
    if (rank < pair && chunk[chunk_size - 1] > tmp_chunk[0]) {
        int i = 0,
```

```
            j = 0,
            k = 0;

        while (k < chunk_size)
            if (chunk[i] < tmp_chunk[j])
                new_chunk[k++] = chunk[i++];
            else
                new_chunk[k++] = tmp_chunk[j++];

        memcpy(chunk, new_chunk, chunk_size * sizeof(int));

    } else if (rank > pair && chunk[0] < tmp_chunk[chunk_size - 1]) {
        int i = chunk_size - 1,
            j = chunk_size - 1,
            k = chunk_size - 1;

        while (k >= 0)
            if (chunk[i] < tmp_chunk[j])
                new_chunk[k--] = tmp_chunk[j--];

            else
                new_chunk[k--] = chunk[i--];

        memcpy(chunk, new_chunk, chunk_size * sizeof(int));

    }
}

//    ollecting    all chunks of array into single array.
MPI_Gather(chunk, chunk_size, MPI_INTEGER, array, chunk_size, MPI_INTEGER, 0

// End of timing.
if (!rank) {
    end_time = MPI_Wtime();

    FILE *fd = fopen("mpi.txt", "a+t");
    fprintf(fd, "%lf ", end_time - start_time);
    fclose(fd);
}

// Free memory.
free(chunk);
free(tmp_chunk);
free(new_chunk);
if (!rank)
    free(array);
```

```
    // Termination MPI.
    MPI_Finalize();

    return 0;
}
```

4.5 plot.py

```
from sys import argv
import matplotlib.pyplot as plt


def read_file(filename):
    tests_number = int(argv[1])
    threads_number = int(argv[2])

    threads = []
    x = 1
    while x <= threads_number:
        threads.append(x)
        x *= 2

    times = []
    file = open(filename, 'r')
    timing = [float(x) for x in file.readline().split()]
    for i in range(len(threads)):
        x = 0
        for j in range(tests_number):
            x += timing[len(threads) * j + i]
        times.append(x / tests_number)
    file.close()

    return threads, times


def draw_plots(data1, data2):
    plt.title("Execution time", fontsize=25)
    plt.xlabel('Threads')
    plt.ylabel('Time')
    plt.grid(True)
    plt.plot(data1[0], data1[1], 'r')
    plt.plot(data2[0], data2[1], 'b')
    plt.legend(['OpenMP', 'MPI'])
    plt.savefig('images/graphics/execution_time.png')

    acceleration1 = [data1[1][0] / x for x in data1[1]]
    acceleration2 = [data2[1][0] / x for x in data2[1]]
    plt.title("Acceleration", fontsize=25)
    plt.xlabel('Threads')
    plt.ylabel('Acceleration')
    plt.grid(True)
    plt.plot(data1[0], acceleration1, 'r')
```

```
    plt.plot(data2[0], acceleration2, 'b')
    plt.legend(['OpenMP', 'MPI'])
    plt.savefig('images/graphics/acceleration.png')

    efficiency1 = [acceleration1[x] / data1[0][x] for x in range(len(data1[0]))]
    efficiency2 = [acceleration2[x] / data2[0][x] for x in range(len(data2[0]))]
    plt.title("Efficiency", fontsize=25)
    plt.xlabel('Threads')
    plt.ylabel('Efficiency')
    plt.grid(True)
    plt.plot(data1[0], efficiency1, 'r')
    plt.plot(data2[0], efficiency2, 'b')
    plt.legend(['OpenMP', 'MPI'])
    plt.savefig('images/graphics/efficiency.png')


if __name__ == '__main__':
    openmp = read_file('openmp.txt')
    mpi = read_file('mpi.txt')

    print(openmp, mpi)
    draw_plots(openmp, mpi)
```

5 Заключение

В результате проделанной лабораторной работы мы получили, что MPI работает эффективнее чем OpenMP, несмотря на огромные затраты при пересылке частей массива с помощью сообщений между процессорами.