Национальный исследовательский ядерный университет “МИФИ”

Лабораторная работа №5:Технология MPI.

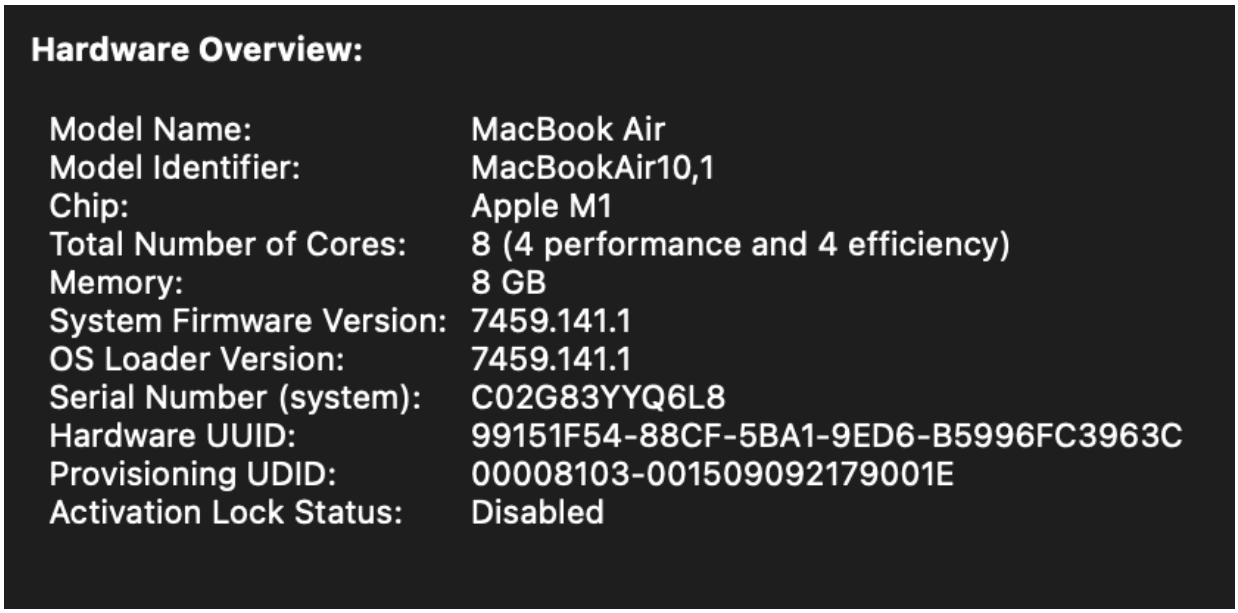
Введение

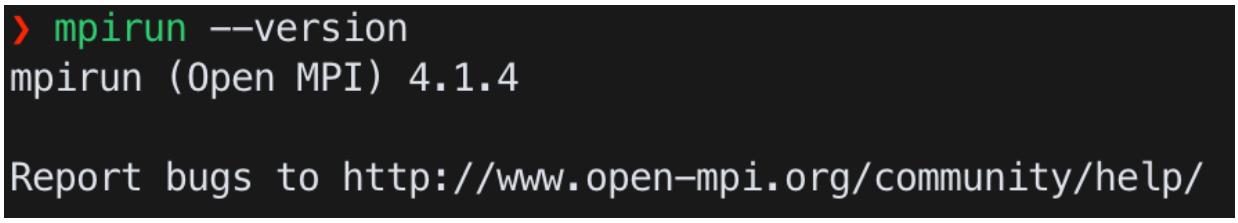
Данилишин Ярослав

Б20-505

2022 год

Описание используемой рабочей среды





Описание хода работы

В этой лабораторной работы необходимо было сравнить алгоритм поиска максимального элемента в массиве, реализованный с помощью **MPI**, а также сравнить полученные результаты с результатами, приведенными в лабораторной работе №1, где алгоритм был реализован с помощью **OpenMP**

Описание алгоритма

Алгоритм поиска максимума в одномерном массиве с помощью технологии **MPI** заключается в том, что массив длины N разбивается на равные промежутки, количество которых определяется заданным числом работающих потоков.

```
MPI_Bcast(array, count, MPI_INTEGER, 0, MPI_COMM_WORLD)
```

Функция широковещательной передачи **MPI_BCAST** посылает сообщение из корневого процесса всем процессам группы. В момент возврата управления содержимое корневого буфера обмена будет уже скопировано во все процессы.

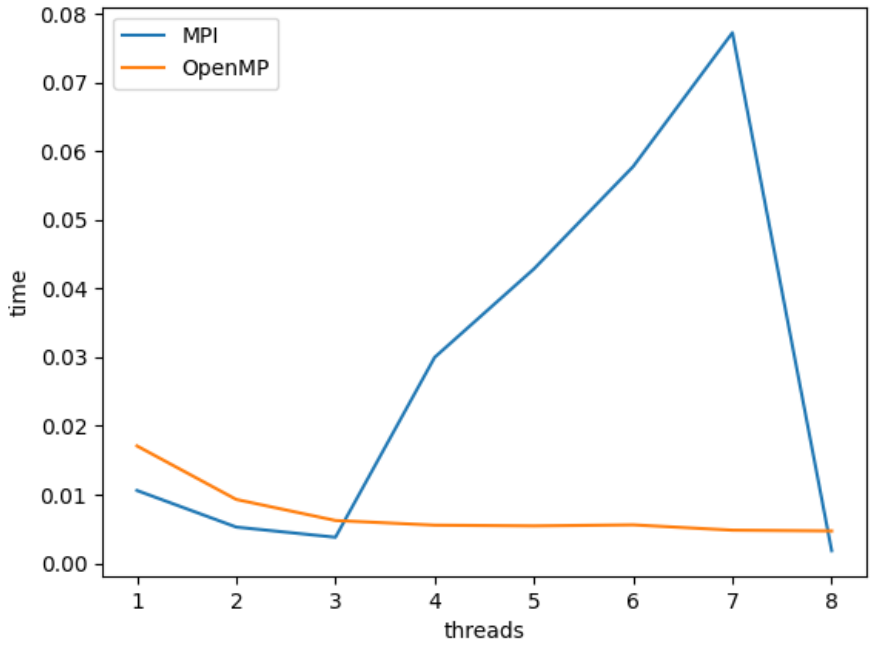
В каждой подпоследовательности массива находится локальный максимум и с помощью опции

```
MPI_Reduce(&local_max, &max, 1, MPI_INTEGER, MPI_MAX, 0,
MPI_COMM_WORLD)
```

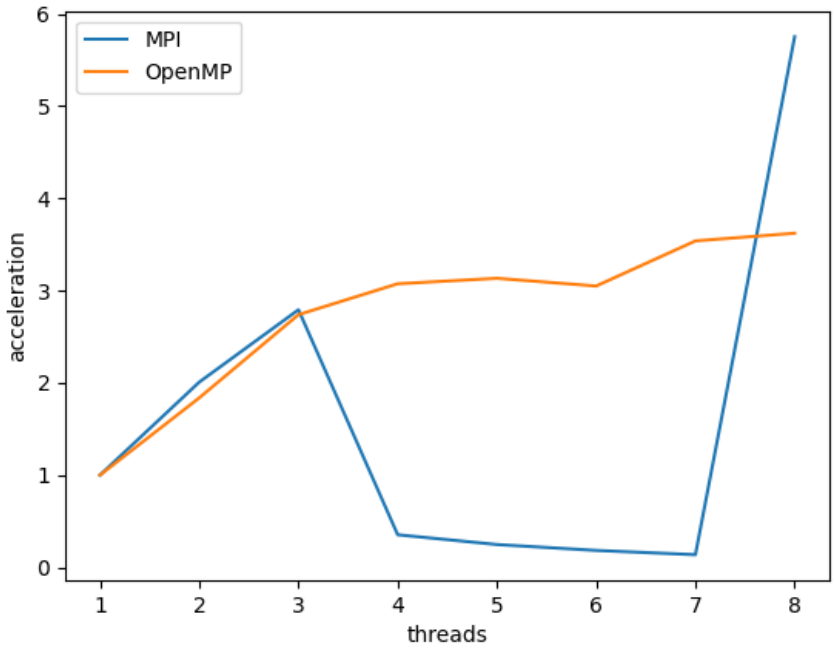
выбирается главный максимум. Функция **MPI_REDUCE** объединяет элементы входного буфера каждого процесса в группе, используя операцию op(**MPI_MAX**) , и возвращает объединенное значение в выходной буфер процесса с номером root. Буфер ввода определен аргументами sendbuf(**local_max**), count(**1**) и datatype(**MPI_INREGER**); буфер вывода определен параметрами recvbuf(**max**), count(**1**) и datatype(**MPI_INREGER**); оба буфера имеют одинаковое число элементов одинакового типа.

Результаты

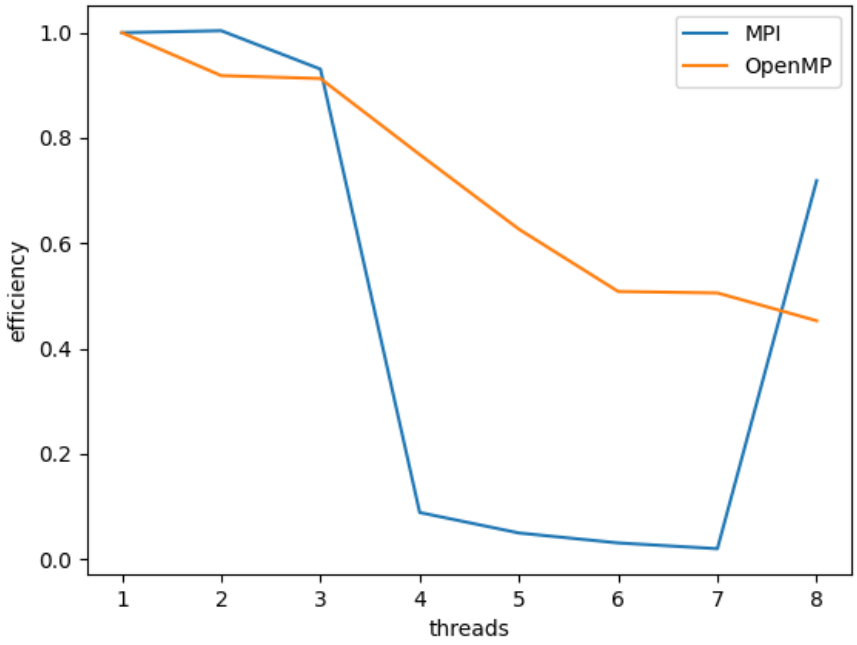
Как и в первой лабораторной было рассмотрено 10 разных массивов на разном количестве потоков (1-8), результат был усреднен. ***Время от числа потоков***



Ускорение от числа потоков



Эффективность от числа потоков



Приложения

- `lab5.c`

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

```
int main(int argc, char** argv)
{
    int ret  = -1;  ///< For return values
    int size = -1;  ///< Total number of processors
    int rank = -1;  ///< This processor's number
    FILE *fd = fopen("data", "ab");
    const int count = 10000000; ///< Number of array elements
    const int random_seed = atoi(argv[1]); ///< RNG seed
    int* array = 0; ///< The array we need to find the max in
    int lmax = -1;  ///< Local maximums
    int  max = -1;  ///< The maximal element
    /* Initialize the MPI */
    ret = MPI_Init(&argc, &argv);
    if (!rank) { printf("MPI Init returned (%d);\n", ret); }

    /* Determine our rankand processor count */
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("MPI Comm Size: %d;\n", size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf("MPI Comm Rank: %d;\n", rank);

    /* Allocate the array */
    array = (int*)malloc(count * sizeof(int));

    /* Master generates the array */
    if (!rank) {
        /* Initialize the RNG */
        srand(random_seed);
        /* Generate the random array */
        for (int i = 0; i < count; i++) { array[i] = rand(); }
    }

    //printf("Processor #%d has array: ", rank);
    //for (int i = 0; i < count; i++) { printf("%d ", array[i]); }
    //printf("\n");

    /* Send the array to all other processors */
    MPI_Bcast(array, count, MPI_INTEGER, 0, MPI_COMM_WORLD);

    //printf("Processor #%d has array: ", rank);
    //for (int i = 0; i < count; i++) { printf("%d ", array[i]); }
    //printf("\n");

    const int wstart = (rank    ) * count / size;
    const int wend   = (rank + 1) * count / size;
```

```
    //printf("Processor #%d checks items %d .. %d;\n", rank, wstart, wend
    clock_t start_time = clock();
    for (int i = wstart; i < wend; i++)
    {
        if (array[i] > lmax) { lmax = array[i]; }
    }

    //printf("Processor #%d reports local max = %d;\n", rank, lmax);

    MPI_Reduce(&lmax, &max, 1, MPI_INTEGER, MPI_MAX, 0, MPI_COMM_WORLD);
    clock_t end_time = clock();
    ret = MPI_Finalize();
    if (!rank) {
        printf("\n*** Global Maximum is %d;\n", max);
    }
    printf("MPI Finalize returned (%d);\n", ret);
    double time = (double)(end_time - start_time) / CLOCKS_PER_SEC;
    if(!rank){
        fwrite(&time, sizeof(double), 1, fd);
    }
    fclose(fd);
    return(0);
}
```

- `process_data.py`

```
import struct as st
import matplotlib.pyplot as plt
from tabulate import tabulate


def read_file_mpi():
    f = open("data", "rb")
    max_num_threads = 8
    iteration_count = 10
    times = [[] for i in range(max_num_threads)]
    for i in range(max_num_threads * iteration_count):
        times[i % max_num_threads].append(float(st.unpack("d", f.read(8)
    threads = [i+1 for i in range(max_num_threads)]
    f.close()
    return times, threads


def read_file_openmp():
    f = open("/Users/danilishinyar/UCHEBA/Parallel_prog/lab1/data", "rb"
    max_num_threads = int(st.unpack("i", f.read(4))[0])
    iteration_count = int(st.unpack("i", f.read(4))[0])
```

```
    times = [[] for i in range(max_num_threads)]
    for i in range(max_num_threads * iteration_count):
        times[i % max_num_threads].append(float(st.unpack("d", f.read(8)
    threads = [i+1 for i in range(max_num_threads)]
    f.close()
    return times, threads


def plots(times_mpi, threads_mpi, times_openmp, threads_openmp):
    times_avg_mpi = [sum(x)/len(x) for x in times_mpi]
    times_avg_openmp = [sum(x)/len(x) for x in times_openmp]
    plt.plot(threads_mpi, times_avg_mpi)
    plt.plot(threads_openmp, times_avg_openmp)
    plt.xlabel("threads")
    plt.ylabel("time")
    plt.legend(["MPI", "OpenMP"])
    plt.savefig("plots/plot_times.png")
    plt.close()

    s_mpi = [(sum(times_mpi[0])/len(times_mpi[0]))/(sum(x)/len(x)) for x
    s_openmp = [(sum(times_openmp[0])/len(times_openmp[0]))/(sum(x)/len(x
    plt.plot(threads_mpi, s_mpi)
    plt.plot(threads_openmp, s_openmp)
    plt.xlabel("threads")
    plt.ylabel("acceleration")
    plt.legend(["MPI", "OpenMP"])
    plt.savefig("plots/plot_acc.png")
    plt.close()

    e_mpi = [s_mpi[i]/(i + 1) for i in range(len(s_mpi))]
    e_openmp = [s_openmp[i]/(i + 1) for i in range(len(s_openmp))]
    plt.plot(threads_mpi, e_mpi)
    plt.plot(threads_openmp, e_openmp)
    plt.xlabel("threads")
    plt.ylabel("efficiency")
    plt.legend(["MPI", "OpenMP"])
    plt.savefig("plots/plot_eff.png")
    plt.close()


data = read_file_mpi()
data1 = read_file_openmp()
plots(data[0], data[1], data1[0], data1[1])
```

- `scrpt.sh`

```
#!/bin/bash

mpicc -o lab5 lab5.cpp

for ((i = 1; i <= 10; i++))
    do
    for ((j=1; j <= 8; j++))
        do
        mpirun -np $j -q lab5 $i
    done
done
rm lab5
python3 process_data.py
```

Заключение

В результате проделанной лабораторной работы мы получили, что MPI работает эффективнее чем OpenMP, при числе потоков меньше 4-ех. Однако, чтоит отметить, что сравнивать технологии OpenMP и MPI не является разумной идеей, потому что при работе с MPI затрачивается огромное количество времни на пересылку массива, к тому же некоторые потоки в MPI могут быть заняты выполнением других приложений и программ, из-за этого невозможно эффективно расчитать время при работе на одной машине.