Национальный исследовательский ядерный университет «МИФИ» (Московский Инженерно–Физический Институт)

Кафедра №42 «Криптология и кибербезопасность»

**Лабораторная работа №5:**

**«Технология MPI. Введение»**

Антон Гатченко Б22-525

2024 г.

*Используемая рабочая среда:*

* Процессор - AMD Ryzen 5 5600H (laptop), 6c/12t
* Оперативная память – DDR4 16 ГБ
* ОС - Windows 10 Pro 22H2 19045.4780, 64 bit
* IDE – GCC/G++ 13.1, OpenMP 201511, Microsoft MPI Version 10.1.12498.16

*Ход работы:*

В ходе работы проводились замеры времени выполнения программы, написанной с использованием технологии MSMPI по поиску максимального числа в массиве целочисленных чисел (int 4 byte) размером элементов. Эти результаты сравнивались с результатами программы с OpenMP.

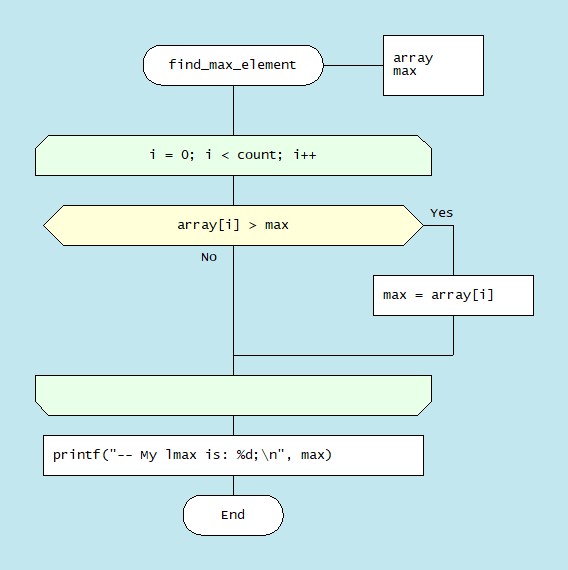
*Алгоритм:*

Теоретическая временная сложность:

Теоретическое ускорение параллельных вычислений:

Теоретическая эффективность параллельных вычислений:

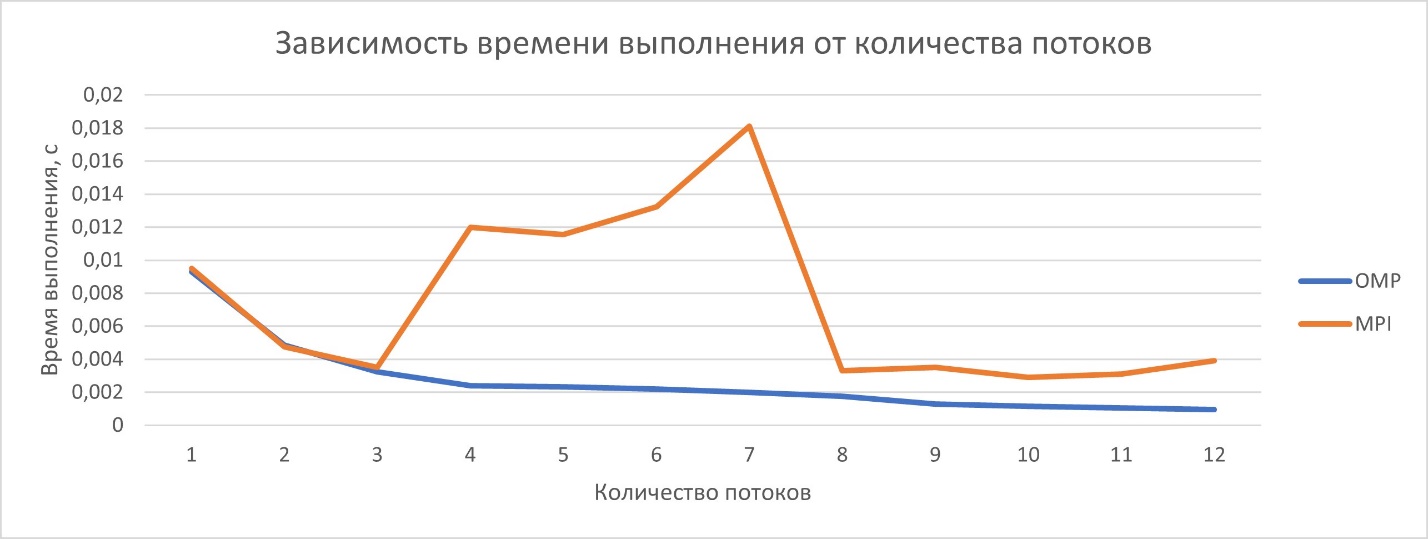
Блок-схема алгоритма:

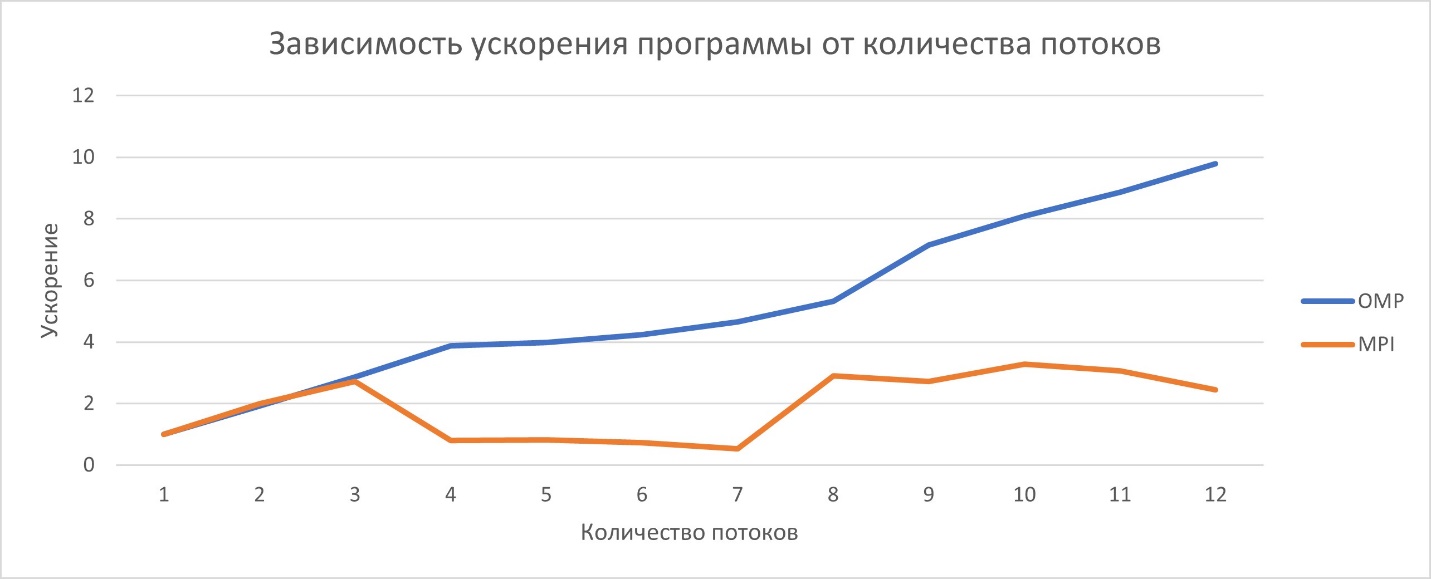


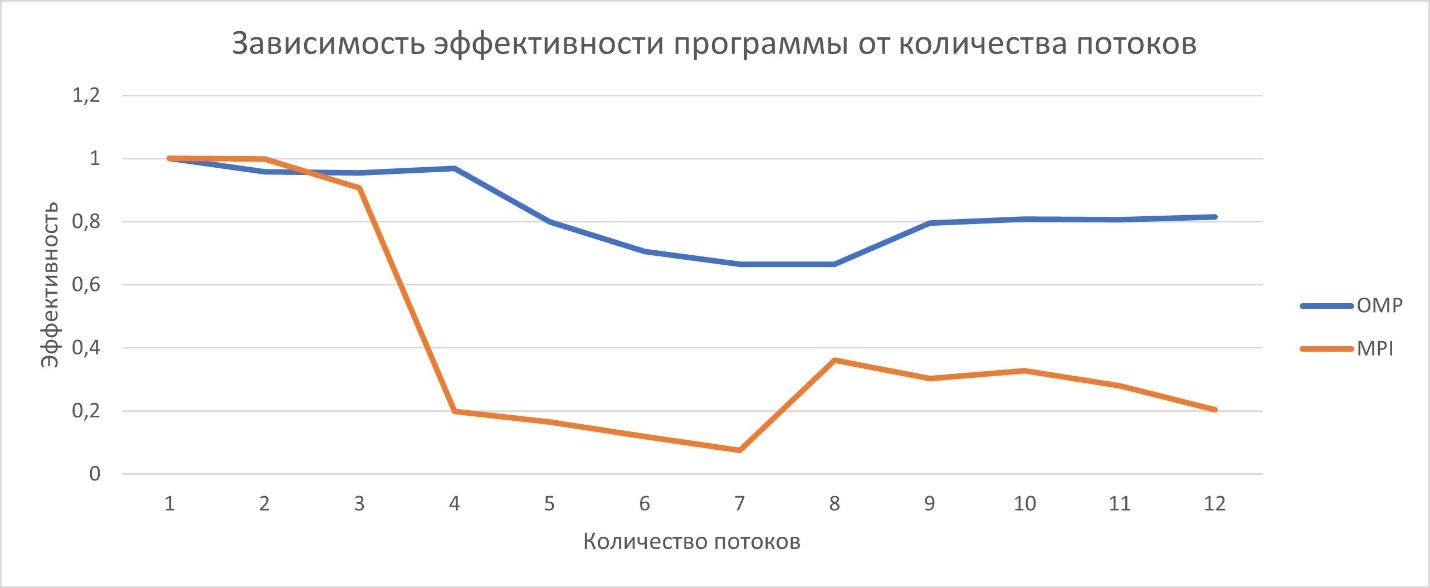
*Использованные функции MSMPI:*

* MPI\_Init() – инициализирует среду MPI и должна быть первой функцией, вызываемой в любом MPI-приложении.
* MPI\_Comm\_size() – возвращает общее количество процессов в данном коммуникаторе (группе созданных MPI\_Init процессов), который чаще всего является глобальным коммуникатором MPI\_COMM\_WORLD.
* MPI\_Comm\_rank() – возвращает уникальный идентификатор процесса из заданного коммуникатора.
* MPI\_Bcast() – передает данные от одного процесса всем остальным в коммуникаторе.
* MPI\_Reduce() – собирает данные от всех процессов и применяет к ним указанную операцию (максимум в данной программе), после чего результат передается указанному процессу.
* MPI\_Barrier() – каждый процесс останавливается на этой линии, пока все остальные процессы не дойдут до неё.
* MPI\_Wtime() – возвращает текущее время в секундах, используется для измерения времени выполнения программы.
* MPI\_Finalize() – завершает работу MPI и освобождает ресурсы, связанные с MPI. Должна быть последней вызванной функцией в MPI-программе.

*Графики выполнения программы при различном количестве потоков и сравнение с OpenMP-версией:*







*Заключение:*

В ходе данной лабораторной работы был изучен процесс использования технологии MPI для организации параллельных вычислений на примере функции поиска максимального элемента в массиве. Программа была запущена при разном количестве потоков, что позволило оценить влияние распараллеливания на производительность, а также время выполнения и иные показатели сравнивались с OpenMP-версией программы.

Результаты тестов показали, что теоретические предположения об ускорении и эффективности выполнения программы при увеличении числа потоков расходятся с практическими (особенно при количестве потоков). OpenMP оказалась более эффективной при среднем и большом количестве потоков. Программа с использованием MPI на 4-7 потоках показала результат хуже, чем при последовательном выполнении, что возможно объясняется выделением логических ядер для выполнения алгоритма. Поскольку производимые процессорами операции однотипны, разделение ядра на два потока не даст значительного прироста либо ухудшит производительность в таком сценарии.

Снижение производительности может также быть вызвано сторонними процессами системы и влиянием планировщика ОС на выделяемые программе ресурсы.

OpenMP использует совместную память, что позволяет потокам быстрее обмениваться данными без необходимости сложных операций передачи и синхронизации, как в MPI. Это особенно важно при большом количестве потоков.

Таким образом, несмотря на то что MPI является мощным инструментом для распределенных вычислений, для задач, требующих частого обмена данными и синхронизации, OpenMP оказывается более эффективной в сценариях с общим доступом к памяти и малым числом операций передачи данных, то есть при выполнении программы на одном устройстве.

*Приложение:*

1. Исходный код программы с измерением времени работы программы:

#include <cstdlib>  
#include <cstdio>  
#include <mpi.h>  
#include <algorithm>  
#include <iostream>  
#include <random>  
#include <vector>  
  
#define SEED 920215  
#define RUNS\_NUM 15  
  
using std::cout, std::wcout, std::cin, std::endl, std::vector, std::swap, std::min, std::max;  
  
void generate\_random\_array(int \*array, int size, const int seed){  
 std::mt19937\_64 gen(seed);  
 std::uniform\_int\_distribution dist(INT\_MIN, INT\_MAX);  
  
 for (int i = 0; i < size; ++i){  
 array[i] = dist(gen);  
 }  
}  
  
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  
 constexpr int count = 1e7; ///< Number of array elements  
 int \*array = nullptr; ///< 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 = static\_cast<int \*>(malloc(count \* sizeof(int)));  
 int seed = SEED;  
 double timeSpent = 0;  
  
 for (int i = 0; i < RUNS\_NUM; ++i){  
 /\* Master generates the array \*/  
 if (!rank){  
 generate\_random\_array(array, count, seed);  
 seed += 1234;  
 }  
  
 /\* Send the array to all other processors \*/  
 MPI\_Bcast(array, count, MPI\_INTEGER, 0, MPI\_COMM\_WORLD);  
  
 const double start = MPI\_Wtime();  
  
 const int wstart = rank \* count / size;  
 const int wend = (rank + 1) \* count / size;  
  
 for (int i = wstart; i < wend; i++){  
 if (array[i] > lmax){ lmax = array[i]; }  
 }  
  
 MPI\_Reduce(&lmax, &max, 1, MPI\_INTEGER, MPI\_MAX, 0, MPI\_COMM\_WORLD);  
  
 MPI\_Barrier(MPI\_COMM\_WORLD); // same as reduce "end" with max?  
 const double end = MPI\_Wtime();  
 timeSpent += end - start;  
 }  
  
 if (!rank){  
 printf("Time spent: %lf\n", timeSpent / RUNS\_NUM);  
 }  
  
 ret = MPI\_Finalize();  
 return 0;  
}

1. Таблица времени работы программы

|  |  |  |
| --- | --- | --- |
| Время работы программы | | |
| **Количество потоков** | **OMP** | **MPI** |
| 1 | 0,0093 | 0,009515 |
| 2 | 0,0048501 | 0,004762 |
| 3 | 0,0032501 | 0,003497 |
| 4 | 0,0024001 | 0,012 |
| 5 | 0,0023301 | 0,011558 |
| 6 | 0,0022002 | 0,013245 |
| 7 | 0,0020005 | 0,018103 |
| 8 | 0,0017501 | 0,00329 |
| 9 | 0,0013001 | 0,003501 |
| 10 | 0,0011503 | 0,002899 |
| 11 | 0,00105 | 0,003102 |
| 12 | 0,0009504 | 0,003901 |

1. Таблица ускорения программы

|  |  |  |
| --- | --- | --- |
| Ускорение программы | | |
| **Количество потоков** | **OMP** | **MPI** |
| 1 | 1 | 1 |
| 2 | 1,917486237 | 1,998110038 |
| 3 | 2,861450417 | 2,720903632 |
| 4 | 3,874838548 | 0,792916667 |
| 5 | 3,991245011 | 0,823239315 |
| 6 | 4,226888465 | 0,718384296 |
| 7 | 4,648837791 | 0,525603491 |
| 8 | 5,313982058 | 2,892097264 |
| 9 | 7,1532959 | 2,717794916 |
| 10 | 8,084847431 | 3,282166264 |
| 11 | 8,857142857 | 3,067375887 |
| 12 | 9,785353535 | 2,439118175 |

1. Таблица эффективности программы

|  |  |  |
| --- | --- | --- |
| Эффективность программы | | |
| **Количество потоков** | **OMP** | **MPI** |
| 1 | 1 | 1 |
| 2 | 0,958743119 | 0,999055019 |
| 3 | 0,953816806 | 0,906967877 |
| 4 | 0,968709637 | 0,198229167 |
| 5 | 0,798249002 | 0,164647863 |
| 6 | 0,704481411 | 0,119730716 |
| 7 | 0,664119684 | 0,075086213 |
| 8 | 0,664247757 | 0,361512158 |
| 9 | 0,794810656 | 0,301977213 |
| 10 | 0,808484743 | 0,328216626 |
| 11 | 0,805194805 | 0,278852353 |
| 12 | 0,815446128 | 0,203259848 |