Национальный исследовательский ядерный университет «МИФИ» (Московский Инженерно–Физический Институт)
Кафедра №42 «Криптология и кибербезопасность»

Лабораторная работа №5:
«Технология MPI. Введение»

Одношивкин Илья, Б21-515

Описание архитектуры

```
Architecture:            x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         48 bits physical, 48 bits virtual
  Byte Order:            Little Endian
CPU(s):                  12
  On-line CPU(s) list:   0-11
Vendor ID:               AuthenticAMD
    Model name:          AMD Ryzen 5 5500U with Radeon Graphics
    CPU family:          23
    Model:               104
    Thread(s) per core:  2
    Core(s) per socket:  6
    Socket(s):           1
    Stepping:            1
    CPU(s) scaling MHz:  52%
    CPU max MHz:         4056.0000
    CPU min MHz:         400.0000
    BogoMIPS:            4192.31
```

```
Transient hostname: DESKTOP-J2NEN3H
Icon name: computer-laptop
Chassis: laptop 
Machine ID: caf94732efe24b519ce9ca85095c24f4
Boot ID: 64bd1e3a27ad44128e6b00b59b2c533f
Operating System: Fedora Linux 38 (Workstation Edition)
CPE OS Name: cpe:/o:fedoraproject:fedora:38
OS Support End: Tue 2024-05-14
OS Support Remaining: 6month 1w 5d
Kernel: Linux 6.5.6-200.fc38.x86_64
Architecture: x86-64
Hardware Vendor: HUAWEI
Hardware Model: NBM-WXX9
Firmware Version: 2.09
Firmware Date: Wed 2022-03-23
```

```
               total        used        free      shared  buff/cache   available
Mem:         7428976     4246056      447036      117204     2735884     2759516
Swap:        7428092      858880     6569212
```

Среда разработки

Компилятор: gcc (GCC) 13.2.1 20230728 (Red Hat 13.2.1-1)
Средство сборки: Makefile
Текстовый редактор: Visual Studio Code
Версия OMP: 201511
Верся MPI: 4.1.4

Ход работы

Для начала мною был установлен MPI с помощью команды:

```
sudo dnf install openmpi openmpi-devel
```

После чего, я изучил приложенный к лабораторной работе алгоритм. Дополнительно были произведены замеры времени одного и того же алгоритма, написанного для OMP и MPI.

Описание программы

OMP: Программа, использующая OpenMP, реализует поиск максимального элемента в массиве случайных целых чисел. Исходный массив создается и инициализируется случайными значениями. Затем программа выполняет поиск максимального элемента в массиве с использованием параллельной директивы #pragma omp parallel for для распараллеливания внешнего цикла. Количество потоков задается во внешнем цикле и варьируется от 1 до максимального значения max_threads.

MPI: Программа, использующая MPI, реализует поиск максимального элемента в массиве случайных целых чисел. Исходный массив создается и инициализируется случайными значениями только в процессе с рангом 0. Затем происходит рассылка этого массива ко всем процессам с использованием функции MPI_Bcast. Каждый процесс выполняет часть работы, а именно, находит локальный максимум в своем подмассиве. Затем с помощью MPI_Reduce локальные максимумы объединяются в глобальный максимум, который выводится на экран вместе с информацией о времени выполнения и количестве процессов.

Описание директив и функций MPI

MPI_Init(&argc, &argv) - Инициализирует MPI для параллельного выполнения программы.

MPI_Comm_size(MPI_COMM_WORLD, &size) - Определяет общее количество процессов в группе MPI_COMM_WORLD и записывает это значение в переменную size.

MPI_Comm_rank(MPI_COMM_WORLD, &rank) - Определяет ранг (или индекс) текущего процесса в группе MPI_COMM_WORLD и записывает это значение в переменную rank.

MPI_Wtime() - Эта функция используется для измерения времени в MPI-приложениях. Она возвращает текущее время в секундах.

MPI_Bcast(array, count, MPI_INTEGER, 0, MPI_COMM_WORLD) - Эта функция рассылает данные от процесса с рангом 0 ко всем остальным процессам.

MPI_Reduce(&lmax, &max, 1, MPI_INTEGER, MPI_MAX, 0, MPI_COMM_WORLD) - Эта функция объединяет значения переменной lmax (lmax - локальный максимум для кажого процесса) среди всех процессов и сохраняет результат в переменной max

MPI_Finalize() - Завершает работу с MPI, освобождает ресурсы, выделенные для параллельного выполнения программы. Эта функция должна быть вызвана перед завершением программы.

График №1

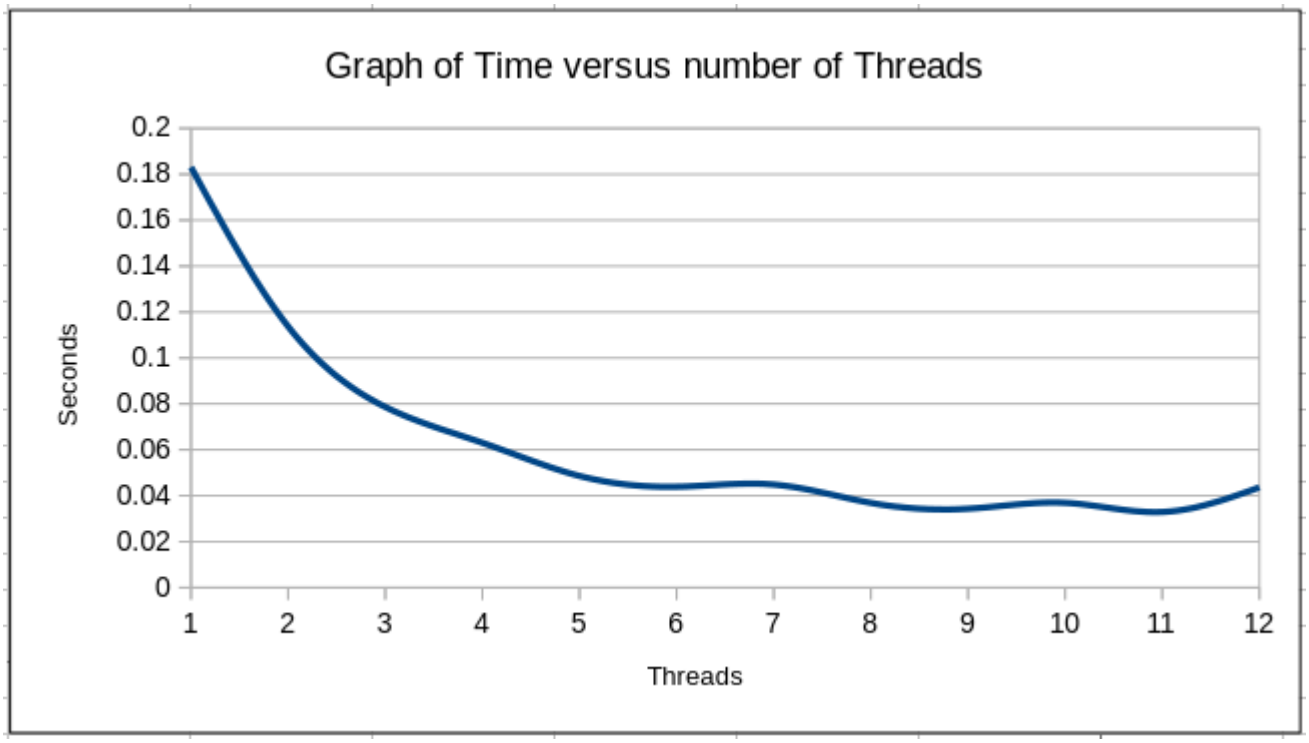

График №2

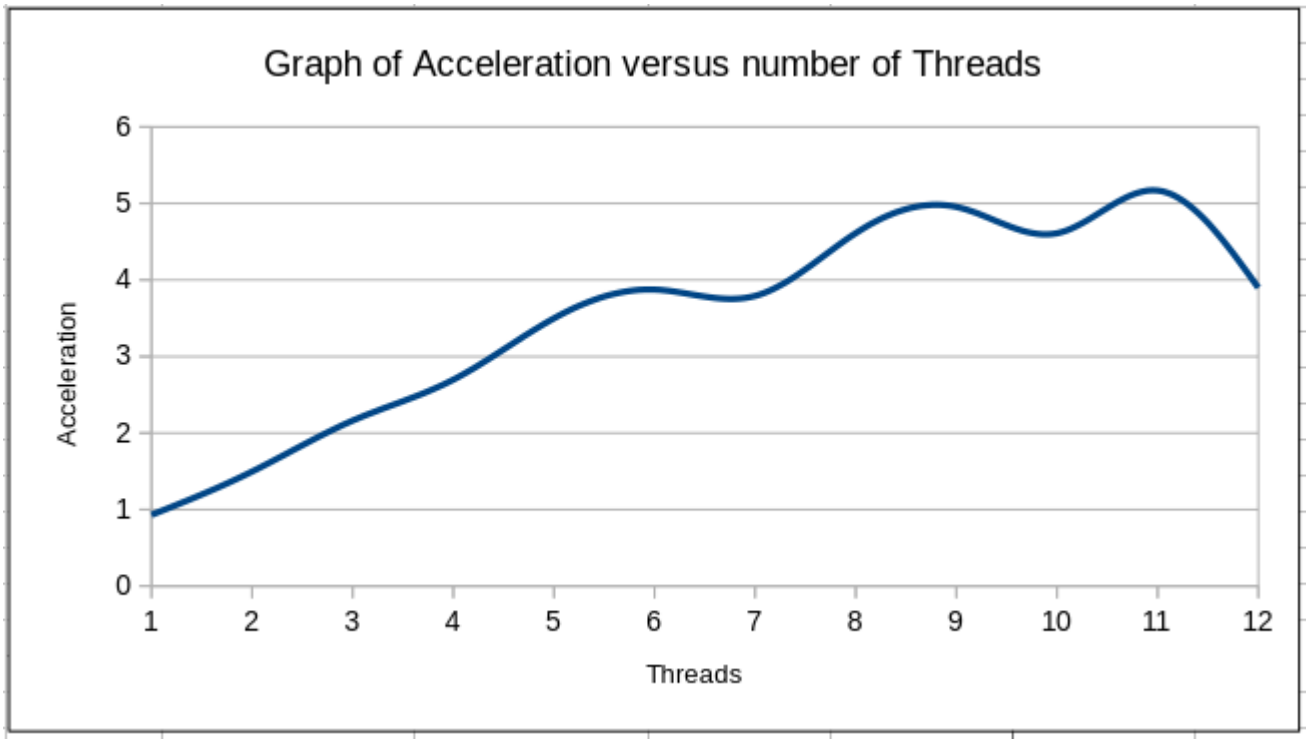

График №3

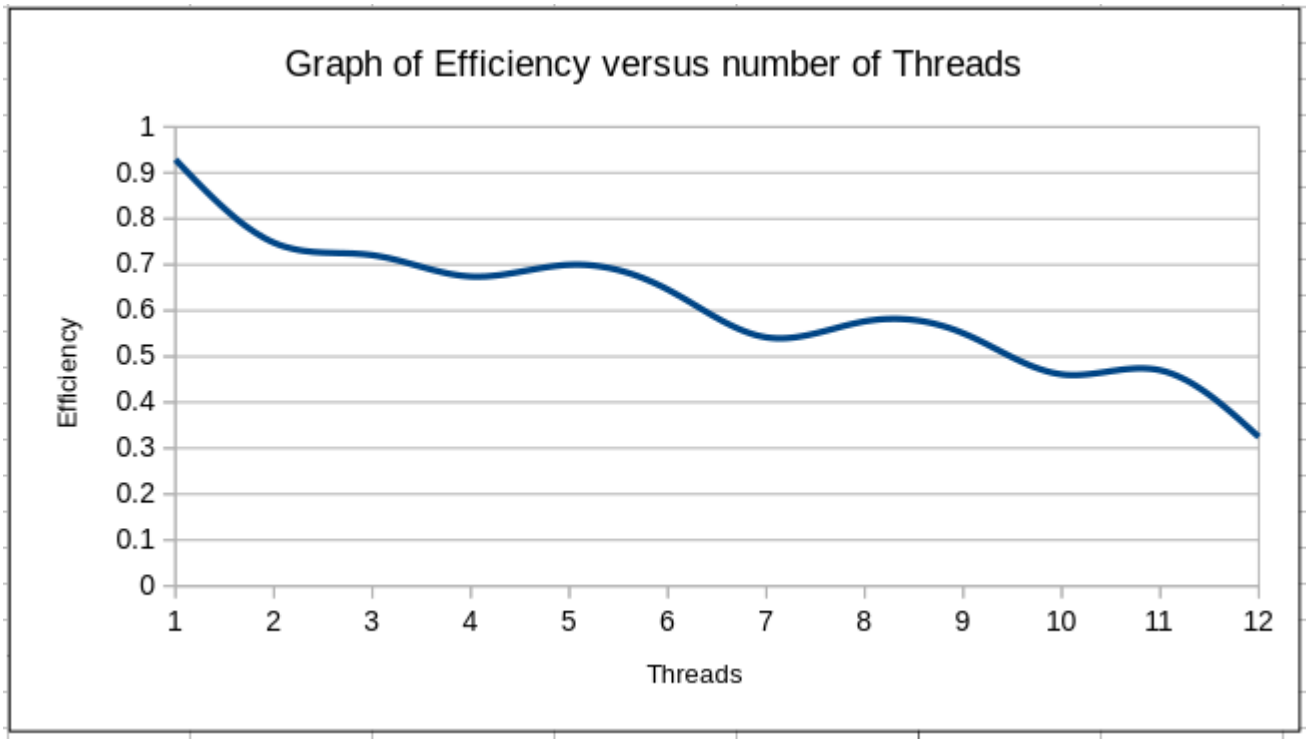

График №4

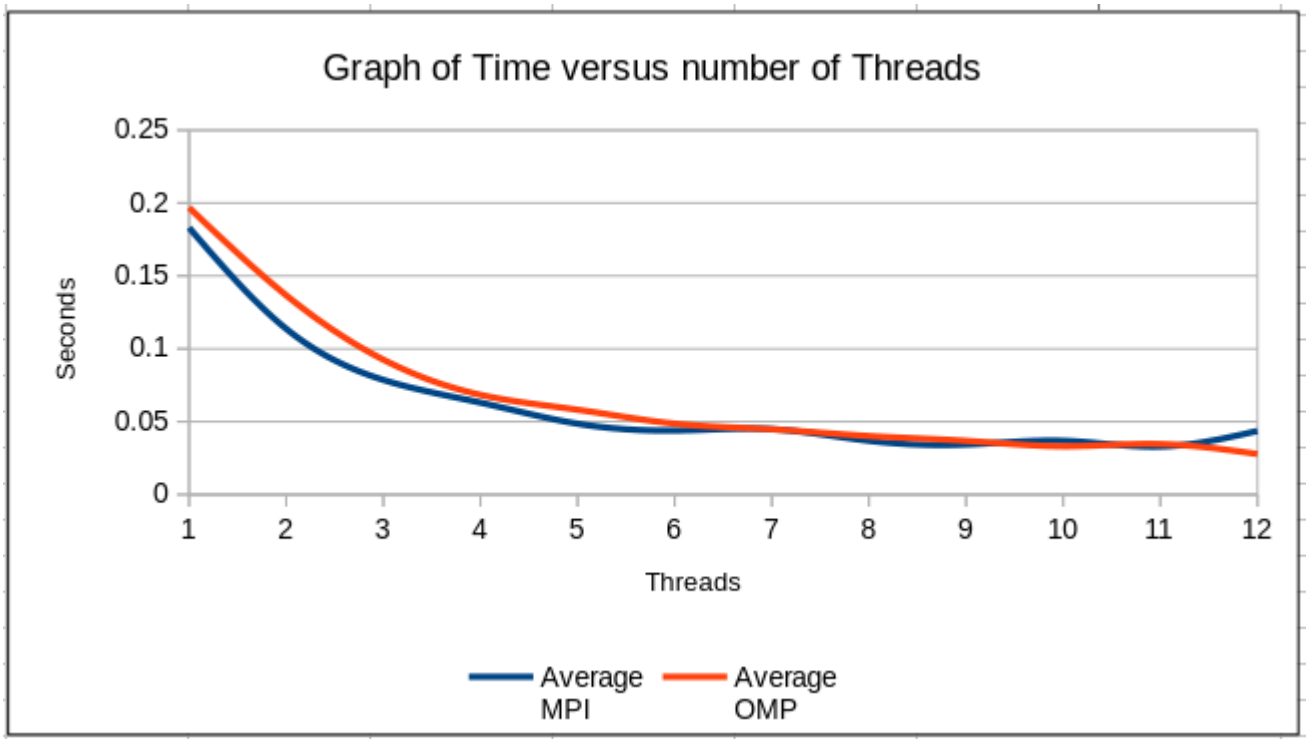

График №5

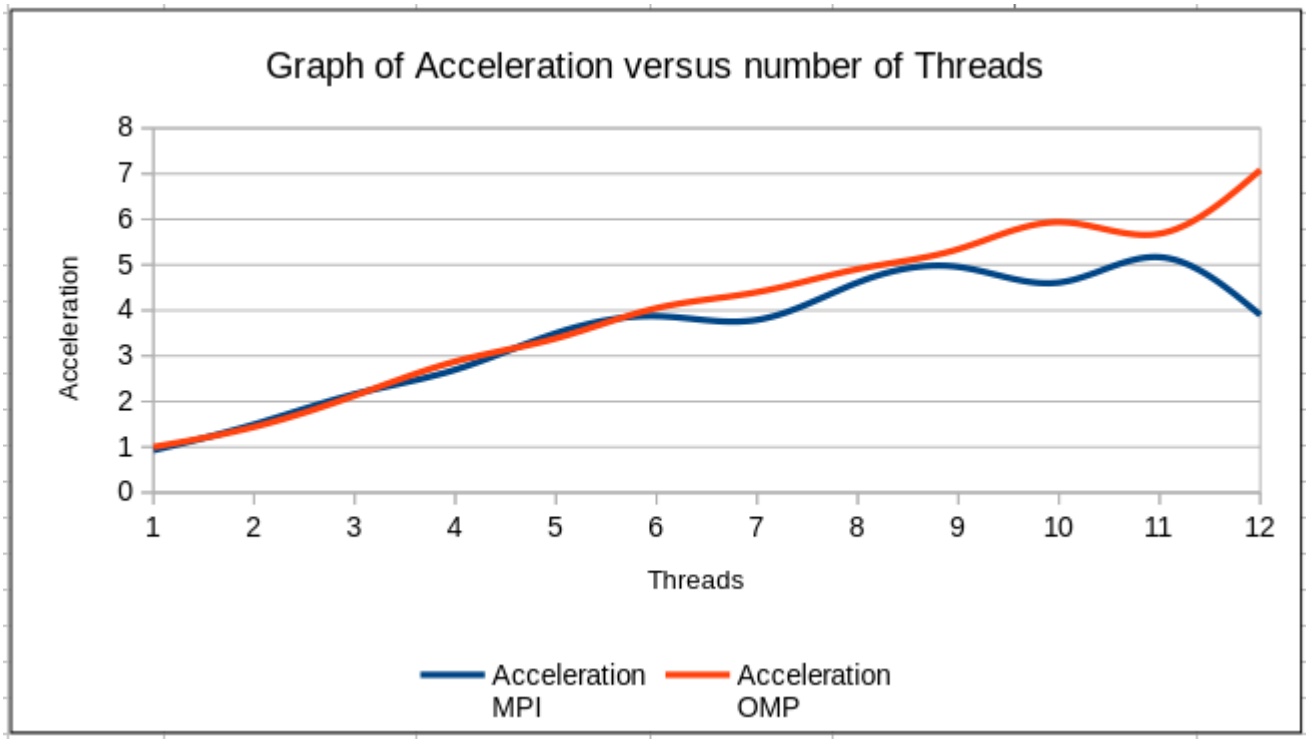

График №6

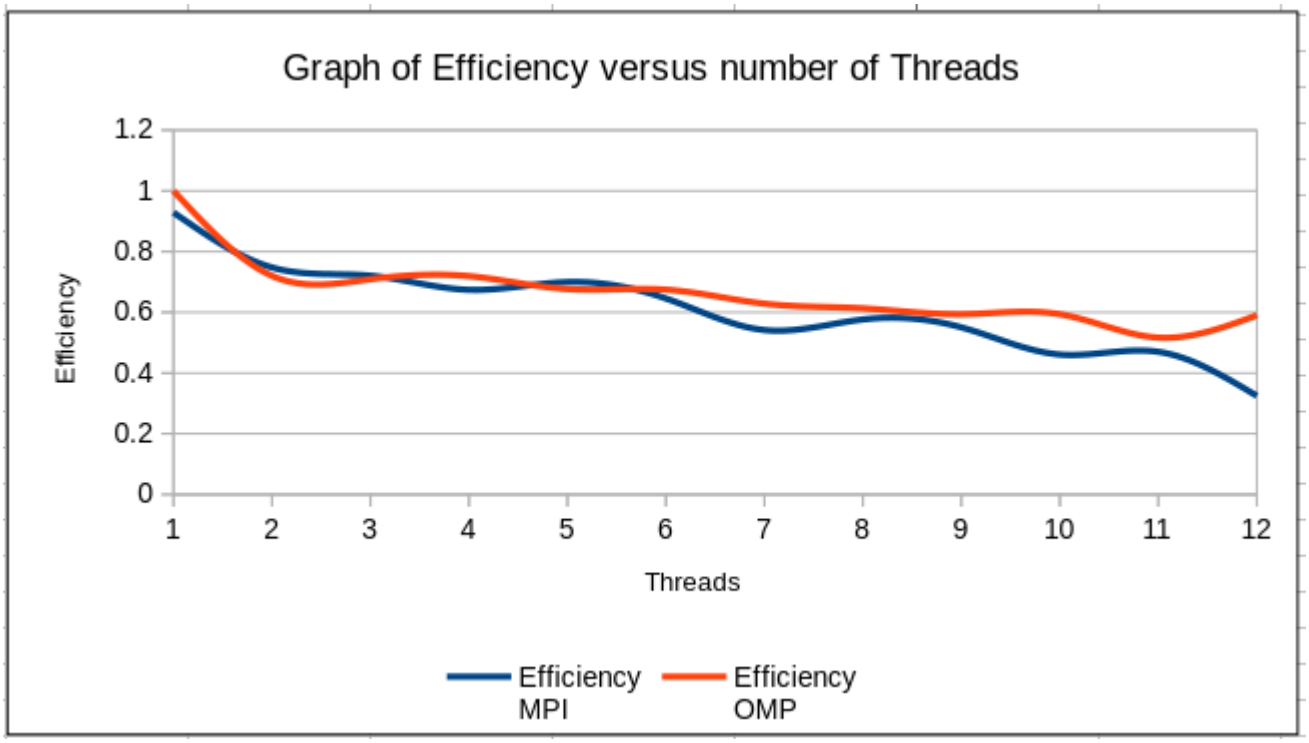

Таблица с данными

Threads	OMP:	Attempt #1	Attempt #2	Attempt #3			Average OMP		Acceleration OMP	Efficiency OMP
1		0.2763	0.156037	0.15818			0.196839		1	1
2		0.144443	0.137419	0.128165			0.1366756667		1.440190524039	0.72009526202
3		0.096956	0.091505	0.089183			0.092548		2.126885508061	0.70896183602
4		0.072454	0.066712	0.066099			0.0684216667		2.876851874406	0.7192129686
5		0.058176	0.05689	0.059458			0.0581746667		3.383586211639	0.67671724233
6		0.04842	0.047538	0.049916			0.0486246667		4.048130578444	0.67468842974
7		0.045841	0.044178	0.04421			0.044743		4.399325034084	0.62847500487
8		0.041772	0.039037	0.039579			0.0401293333		4.905115127754	0.61313939097
9		0.037701	0.034688	0.038124			0.0368376667		5.34341661162	0.59371295685
10		0.034139	0.031358	0.033974			0.033157		5.936574478994	0.5936574479
11		0.031016	0.039083	0.033816			0.0346383333		5.682692585286	0.51660841684
12		0.028562	0.026614	0.028217			0.0277976667		7.081133908122	0.59009449234
Procs	MPI:	Attempt #1	Attempt #2	Attempt #3			Average MPI		Acceleration MPI	Efficiency MPI
1		0.170847	0.169675	0.208802			0.183108		0.928709832449	0.92870983245
2		0.129871	0.084934	0.126333			0.1137126667		1.495472799864	0.74773639993
3		0.076451	0.079675	0.079867			0.0786643333		2.161770052502	0.7205900175
4		0.065006	0.061887	0.062298			0.0630636667		2.696547933041	0.67413698326
5		0.045056	0.052577	0.048195			0.0486093333		3.498385769537	0.69967715391
6		0.065006	0.028755	0.037939			0.0439		3.873671981777	0.64561199696
7		0.044157	0.044682	0.045729			0.044856		3.791113786339	0.54158768376
8		0.039581	0.035522	0.035522			0.036875		4.611639322034	0.57645491525
9		0.036229	0.03317	0.033568			0.0343223333		4.954622354735	0.55051359497
10		0.042296	0.035678	0.032699			0.036891		4.609639207395	0.46096392074
11		0.03378	0.03378	0.031142			0.0329006667		5.168715932808	0.46988326662
12		0.052296	0.039671	0.038916			0.0436276667		3.897852280281	0.32482102336

Код программы OMP

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


int main()
{
    const int count = 100000000;
    const int random_seed = 920215;
    const int max_threads = 12;
    int max = -1;

    srand(random_seed);

    int *array = 0;
    array = (int *)malloc(count * sizeof(int));
    for (int i = 0; i < count; i++) { array[i] = rand(); }


    for (int threads = 1; threads <= max_threads; threads++)
    {
        double start_time = omp_get_wtime();

        #pragma omp parallel num_threads(threads) shared(array, count) reduction(max : max) default(none)
        {
        #pragma omp for
            for (int i = 0; i < count; i++)
            {
                if (array[i] > max)
                {
                    max = array[i];
                };
            }
        }

        printf("Threads: %d, Execution time: %f seconds\n", threads, omp_get_wtime() - start_time);
    }
    free(array);

    return 0;
}
```

Код программы MPI

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

int main(int argc, char** argv)
{
    int ret  = -1;
    int size = -1;
    int rank = -1;

    const int count = 100000000;
    const int random_seed = 920215;

    int* array = NULL;
    int lmax = -1;
    int  max = -1;

    ret = MPI_Init(&argc, &argv);

    if (!rank) { printf("MPI Init returned (%d);\n", ret); }

    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    array = malloc(count * sizeof(int));

    if (!rank) {
        srand(random_seed);
        for (int i = 0; i < count; i++) { array[i] = rand(); }
    }

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

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

    MPI_Barrier(MPI_COMM_WORLD);

    double start_time = MPI_Wtime();

    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);

    double end_time = MPI_Wtime();

    if (!rank) {
        printf("Execution time: %f seconds\n", end_time - start_time);
    }

    free(array);

    MPI_Finalize();

    return(0);
}
```

Заключение

В ходе данного исследования была разработана параллельная программа на основе MPI для поиска максимального элемента. Дополнительно было произведено сравнение производительности одного и того же алгоритма для OMP и MPI.

В результате работы программы была получена зависимость среднего времени выполнения T(n) для каждой версии программы. Также была вычислена зависимость ускорения от числа потоков по формуле: A(n) = T(1)/ T(n). После этого была расчитана зависимость эффективности от числа потоков по формуле: E(n) = A(n)/n, Где T – время (в секундах) выполнения программы, n – количество потоков, A – ускорение, E – эффективность.

В OMP при увеличении числа потоков наблюдается снижение уровня эффективности, начиная с 1 и уменьшаясь до 0.5900. При добавлении второго потока происходит резкое снижение с 1 до 0.7200, но при этом при каждом добавлении нового потока снижение эффективности происходит в пределах [0; 0.005], что свидетельствует о монотонном убывании. Однако, несмотря на снижение эффективности, ускорение увеличивается при добавлении дополнительных потоков. Исходное значение ускорения составляет 1, а максимальное достигает 7.0811 График ускорения монотонно растет в интервале [0; 0.7].

При использовании MPI эффективность снижается с 1 до 0.3248. Снижение происходит в интервале [0; 0.065]. При переходе с 11 на 12 процесс эффектиновть резко уменьшается на 0.12. В то время как уровень эффективности снижается, до 11 процесса ускорение по-прежнему монотонно возрастает, начиная с 1 и достигая значения 5.1687, после чего резко убывает до 3.8975.

Исходя из полученных данных, мною были рассчитаны ускорение и эффективность алгоритма для разного числа процессов и построены соответствующие графики зависимости времени выполнения, ускорения и эффективности от числа запущенных процессов.

В ходе вычисления времени работы программы на основе MPI, я столкнулся с невозможными результатами скорости выполнения работы. Проблема была в том, что некоторые процессы не дожидались пока остальные завершат свою работу, тем самым переходя сразу к вычислению времени, таким образом появлялись некорректные данные. После установки барьеров вокруг части кода, которая отвечает за вычислительный алгоритм, время выполнения программы нормализовалось.

Сравнив данные, мною была замечено то, что по началу программа на MPI показывала лучшие результаты. Но с увеличением числа потоков OMP продемонстрировал лучшее время. Из чего мною был сделан вывод, что для данной задачи использование OMP более выгодно. При достижении 12 процессов, время выполнения программы на MPI возросло, что на мой взгляд произошло из-за увеличения накладных расходов на обмен сообщениями и конкуренции за ресурсы.

Таким образом, результаты исследования позволяют сделать вывод, что выбор стандарта параллельного программирования зависит от конфигурации системы и особенностей данных, с которыми он работает.