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

Лабораторная работа №6:
«Коллективные операции в MPI»

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

Сначала мной были изучены представленные в MPI операции коллективного обмена данными:

1) MPI_Bcast(void buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

Распространяет данные от корневого процесса (указанного параметром root) всем остальным процессам в коммуникаторе.

2) MPI_Scatter(void sendbuf, int sendcount, MPI_Datatype sendtype, void recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

Рассылает данные от корневого процесса всем процессам в коммуникаторе.

3) MPI_Gather(void sendbuf, int sendcount, MPI_Datatype sendtype, void recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

Собирает данные со всех процессов в коммуникаторе на корневой процесс.

4) MPI_Reduce(void sendbuf, void recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

Сводит данные от всех процессов в коммуникаторе на корневой процесс с использованием заданной операции op.

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_Scatter(array, n, MPI_INT, sub_array, n, MPI_INT, 0, MPI_COMM_WORLD) - Разбивает массив array на равные части и рассылает каждую часть процессам в группе MPI_COMM_WORLD. Каждый процесс получает свою часть данных в массив sub_array.

MPI_Gather(sub_array, n, MPI_INT, sorted_array, n, MPI_INT, 0, MPI_COMM_WORLD) - Собирает отсортированные части массива sub_array из каждого процесса и объединяет их в один отсортированный массив sorted_array в процессе с рангом 0.

MPI_Barrier(MPI_COMM_WORLD) - Эта функция блокирует выполнение программы до тех пор, пока все процессы в группе MPI_COMM_WORLD не достигнут этой точки.

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

Описание программы

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

MPI: Программа, использующая MPI, реализует параллельную сортировку Шелла для массива случайных целых чисел. Исходный массив создается и инициализируется случайными значениями только в процессе с рангом 0. Затем программа распределяет массив между процессами MPI, каждый из которых сортирует свою часть массива. После этого производится сбор и объединение отсортированных частей массива в процессе с рангом 0.

График №1

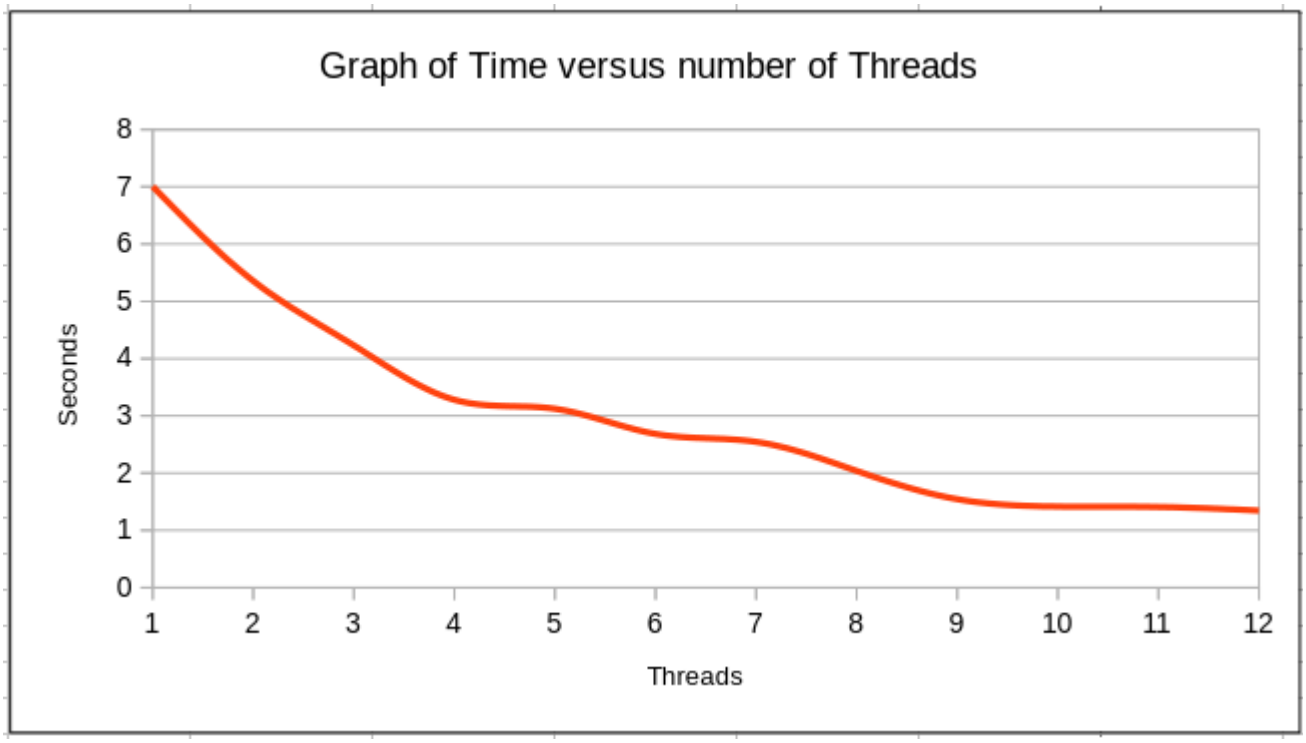

График №2

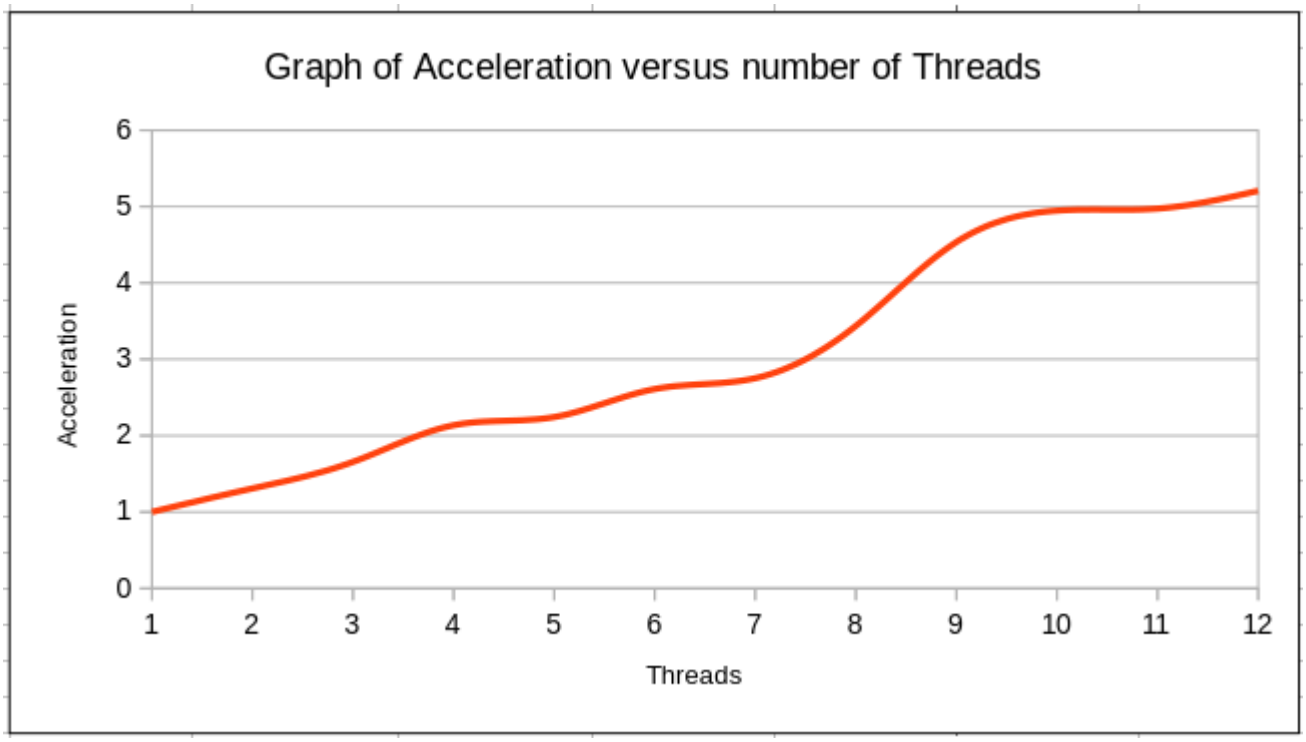

График №3

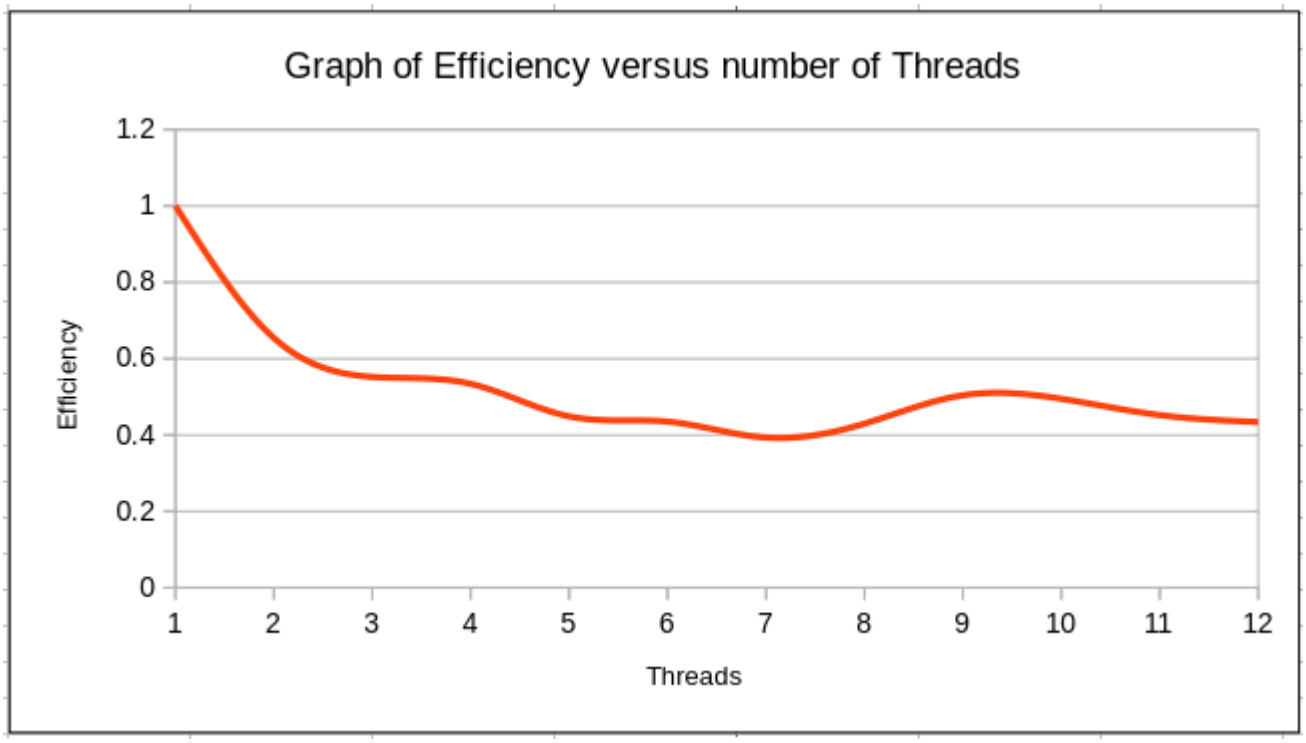

График №4

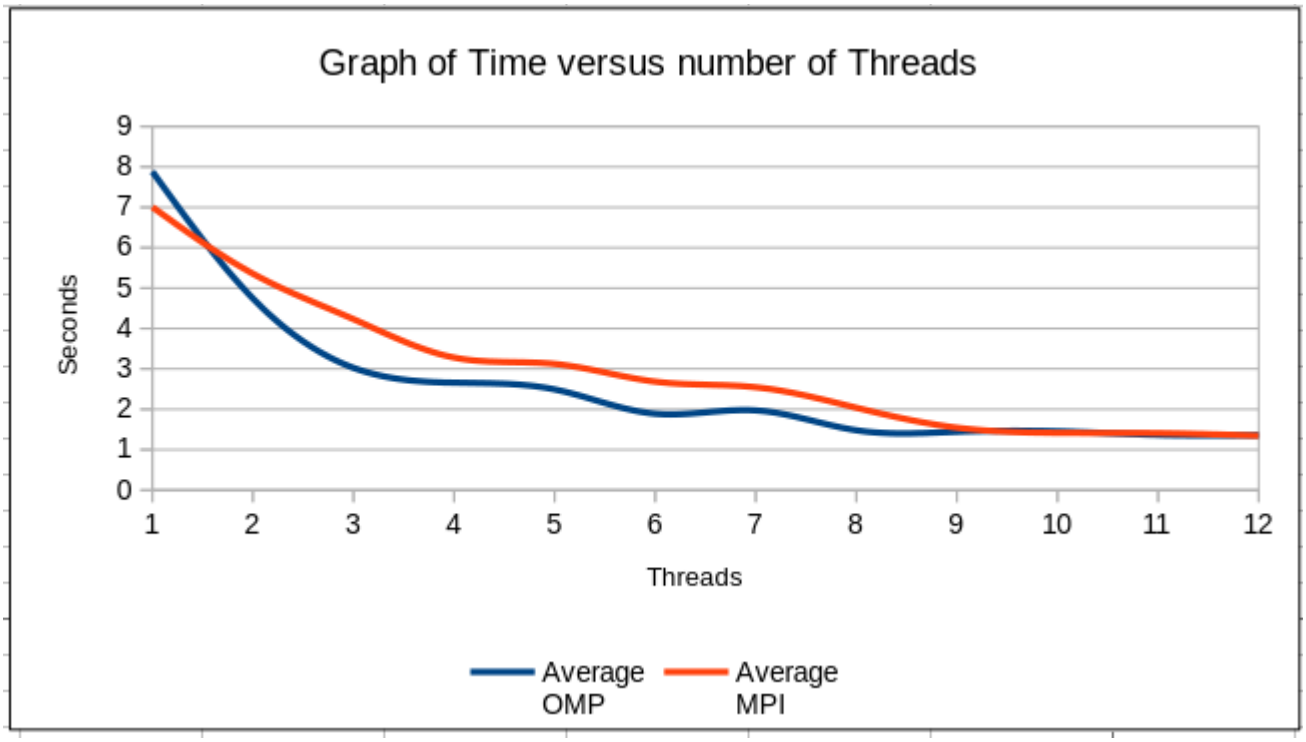

График №5

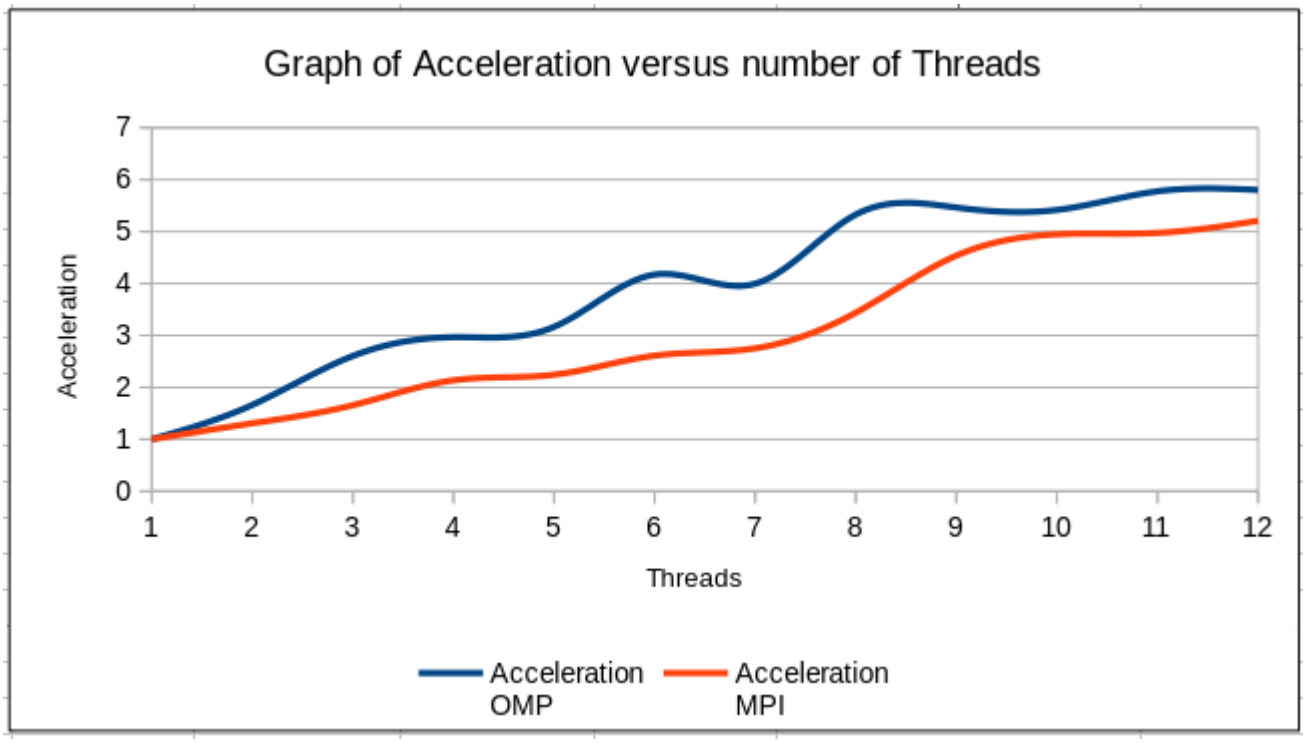

График №6

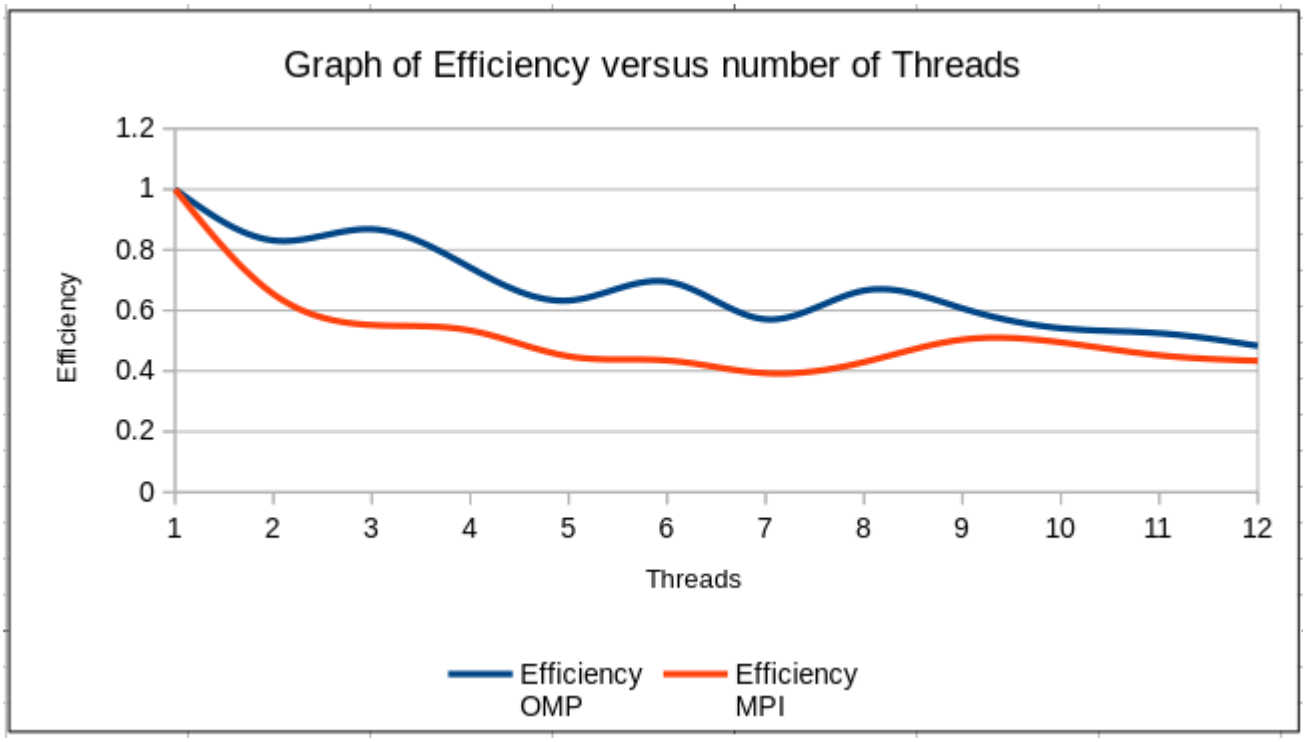

Таблица с данными

Threads	OMP:	Attempt #1	Attempt #2	Attempt #3	Attempt #4	Attempt #5		Average OMP		Acceleration OMP	Efficiency OMP
1		7.205017	8.417694	8.081944	7.535633	8.178634		7.8837844		1	1
2		4.764895	4.930547	4.865805	4.843034	4.309028		4.7426618		1.662312164	0.831156082
3		2.922535	3.059272	3.027379	3.087743	3.034027		3.0261912		2.6051838364	0.868394612
4		2.598083	2.675756	2.6378	2.723091	2.653646		2.6576752		2.9664213294	0.741605332
5		2.453155	2.655202	2.405435	2.460985	2.483572		2.4916698		3.1640566499	0.63281133
6		1.384605	1.917528	1.37378	2.454523	2.318871		1.8898614		4.1716204162	0.695270069
7		2.400698	1.975998	2.48011	1.540916	1.467487		1.9730418		3.9957513318	0.570821619
8		1.385701	1.642552	1.463793	1.511361	1.397167		1.4801148		5.3264681902	0.665808524
9		1.359266	1.862197	1.292335	1.395366	1.307614		1.4433556		5.4621220162	0.606902446
10		1.346808	1.759123	1.444006	1.365501	1.362441		1.4555758		5.4162650959	0.54162651
11		1.272392	1.407646	1.358571	1.475825	1.312111		1.365309		5.7743590645	0.524941733
12		1.223018	1.388053	1.24133	1.576777	1.36637		1.3591096		5.8006980452	0.483391504
Threads	MPI:	Attempt #1	Attempt #2	Attempt #3	Attempt #4	Attempt #5		Average MPI		Acceleration MPI	Efficiency MPI
1		7.504366	7.226164	6.728875	6.612691	6.95893		7.0062052		1	1
2		5.507453	5.414724	4.956203	5.458073	5.442664		5.3558234		1.3081471656	0.654073583
3		4.213782	4.270806	4.241431	4.168606	4.258199		4.2305648		1.6560921606	0.55203072
4		3.262048	3.287651	3.299063	3.273328	3.281019		3.2806218		2.135633312	0.533908328
5		3.10978	3.114984	3.146578	3.123034	3.12224		3.1233232		2.2431893055	0.448637861
6		2.632033	2.645564	2.638929	2.752714	2.752714		2.6843908		2.6099795902	0.434996598
7		2.643647	2.685854	2.407603	2.640251	2.353114		2.5460938		2.7517466953	0.393106671
8		1.957497	2.3771	2.12873	1.869145	1.859414		2.0383772		3.4371485317	0.429643566
9		1.68284	1.485691	1.685814	1.409581	1.459581		1.5447014		4.5356372435	0.503959694
10		1.542856	1.3099	1.411475	1.475429	1.34189		1.41631		4.9468020419	0.494680204
11		1.459248	1.475429	1.394184	1.320911	1.394184		1.4087912		4.9732034101	0.452109401
12		1.323018	1.388053	1.30149	1.376597	1.339518		1.3457352		5.2062286845	0.43385239

Код программы

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


void shell_sort(int* array, int n);

int main(int argc, char** argv) {
    const int count = 10000000;
    const int random_seed = 920215;

    int size, rank;

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

    int n = (count + size - 1) / size;

    int* array = malloc(count * sizeof(int));
    int* sub_array = malloc(n * sizeof(int));
    int* sorted_array = malloc(count * sizeof(int));

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

    MPI_Barrier(MPI_COMM_WORLD);

    double start_time = MPI_Wtime();

    MPI_Scatter(array, n, MPI_INT, sub_array, n, MPI_INT, 0, MPI_COMM_WORLD);

    shell_sort(sub_array, n);

    MPI_Gather(sub_array, n, MPI_INT, sorted_array, n, MPI_INT, 0, MPI_COMM_WORLD);

    MPI_Barrier(MPI_COMM_WORLD);

    if (!rank) {
        printf("Sorted array:\n");
        for (int i = 0; i < count; i++) {
            printf("%d\n", sorted_array[i]);
        }
        printf("Threads: %d, Execution time: %f seconds\n", size, MPI_Wtime() - start_time);
    }

    free(array);
    free(sub_array);
    free(sorted_array);

    MPI_Finalize();

    return 0;
}

void shell_sort(int* array, int n) {
    for (int gap = n / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < n; i++) {
            int temp = array[i];
            int j;
            for (j = i; j >= gap && array[j - gap] > temp; j -= gap)
                array[j] = array[j - gap];

            array[j] = temp;
        }
    }
}
```

Заключение

В ходе данного исследования была разработана параллельная программа на основе MPI для сортировки шелла. 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 добавлении дополнительных потоков уровень эффективности снижается с увеличением числа потоков. Эффективность начинается с 1 и уменьшается до 0.4833. При каждом добавлении нового потока снижение эффективности происходило в интервале [0; 0.1], что говорит о монотонной убывании. При добавлении дополнительных потоков ускорение увеличивается. Начальное значение ускорения составляет 1, а максимальное достигает 5.006. График ускорения монотонно увеличивается в интервале [0; 1].

При использовании MPI при переходе с 1 потока на 2, эффективность резко уменьшается с 1 до 0.6540, после чего убывает в интервале [0.01]. Минимальное значение эффективности: 0.4338. Ускорение монотонно возрастает с 1 до 5.2062.

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

В результате сравнения обоих алгоритмов, мною был сделан вывод, что использование OMP в данном случае лучше. Возможно, компилятор, используемый при реализации OMP-версии программы, лучше оптимизирует код для конкретного вида задачи и характеристик целевой системы, что влияет на производительность.

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