

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

Лабораторная работа №5:
«Технология MPI. Введение»

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

Версия 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 каждого 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

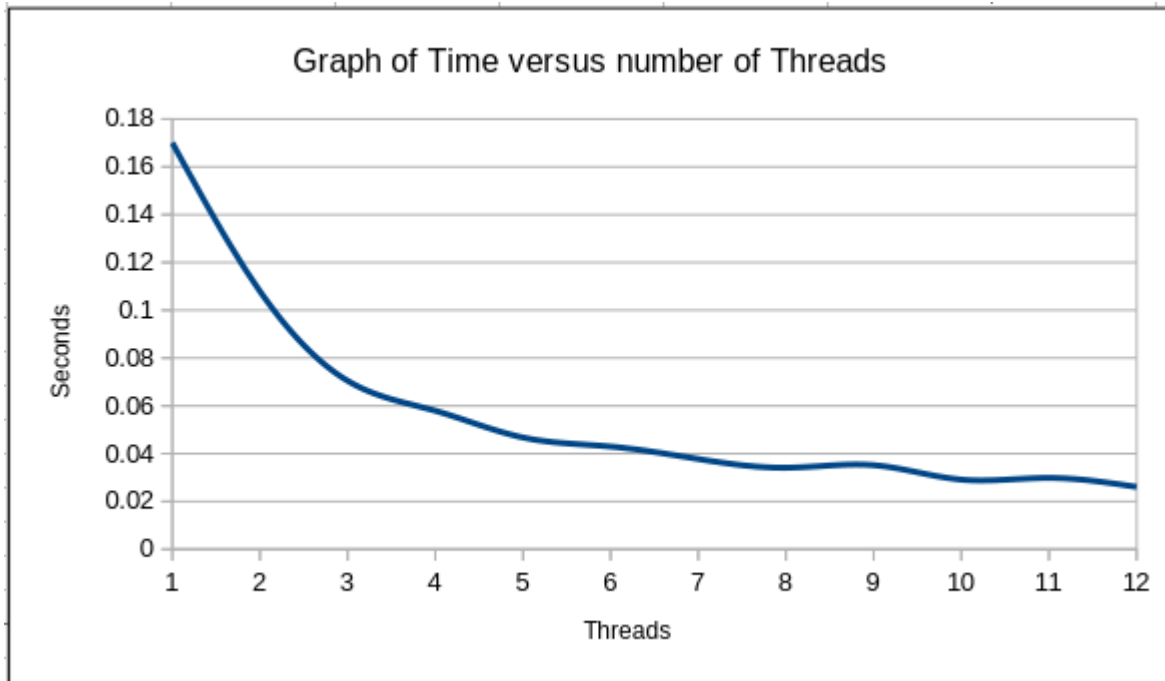


График №2

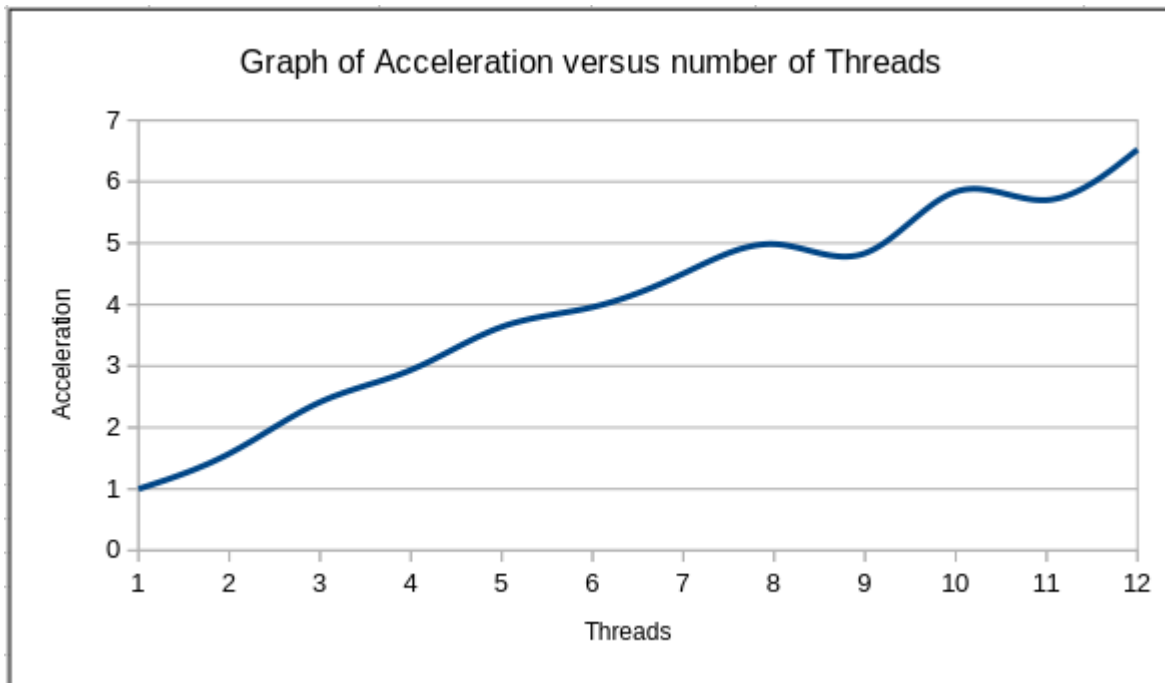


График №3

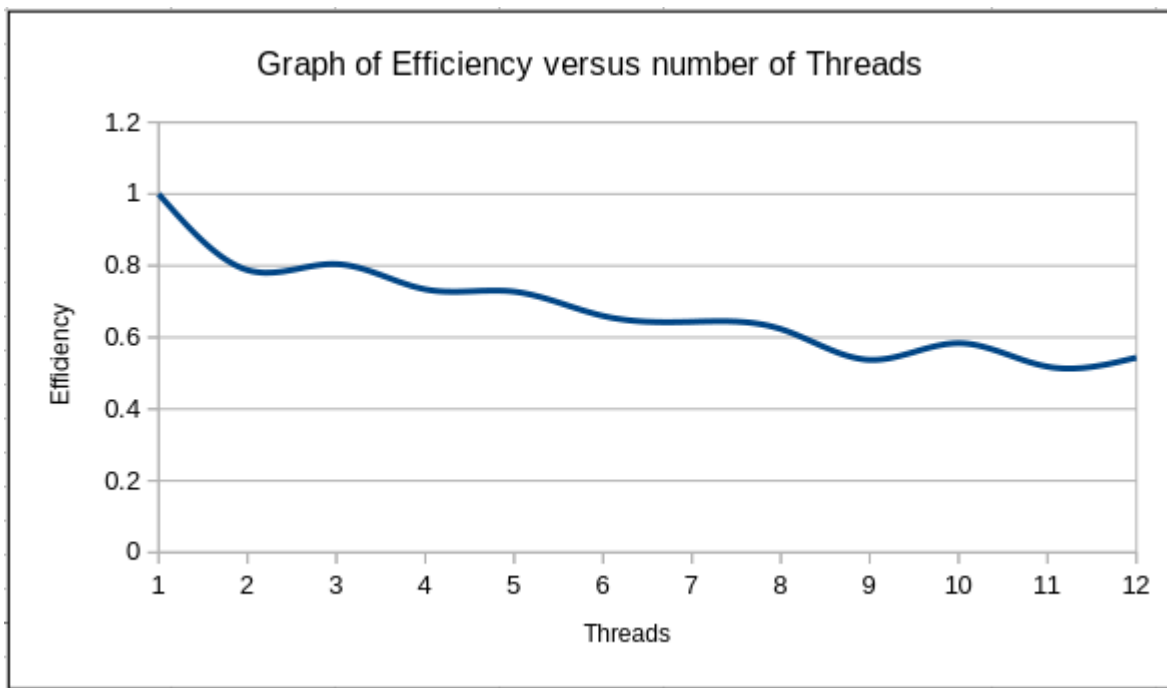


График №4

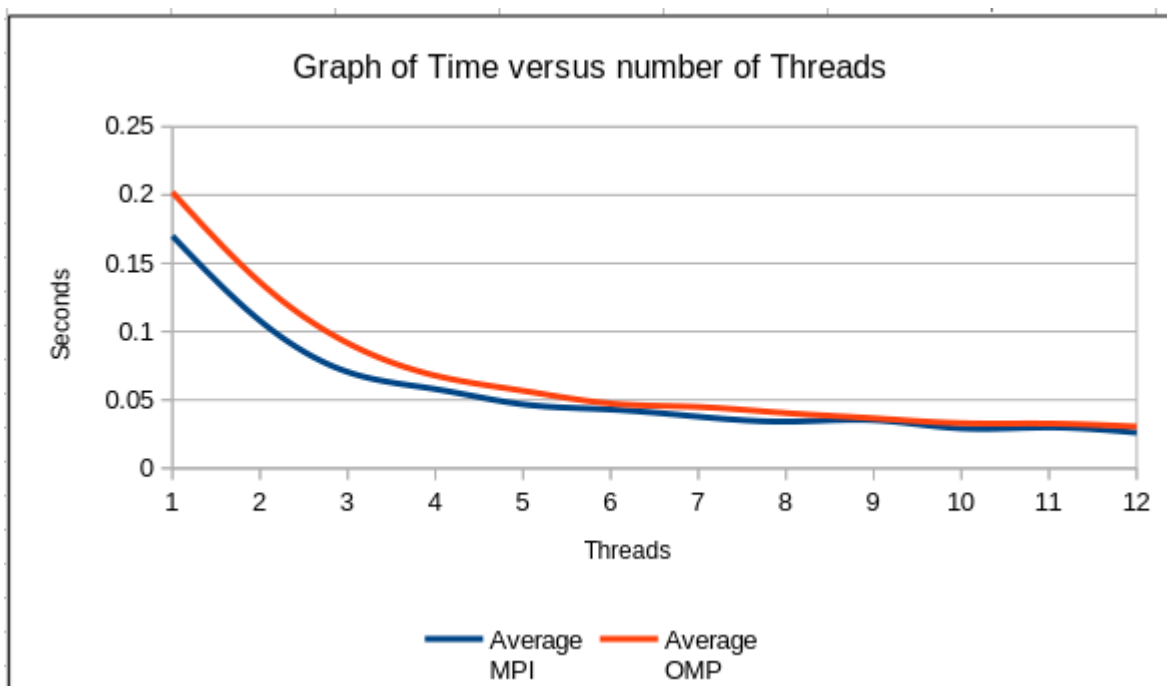


График №5

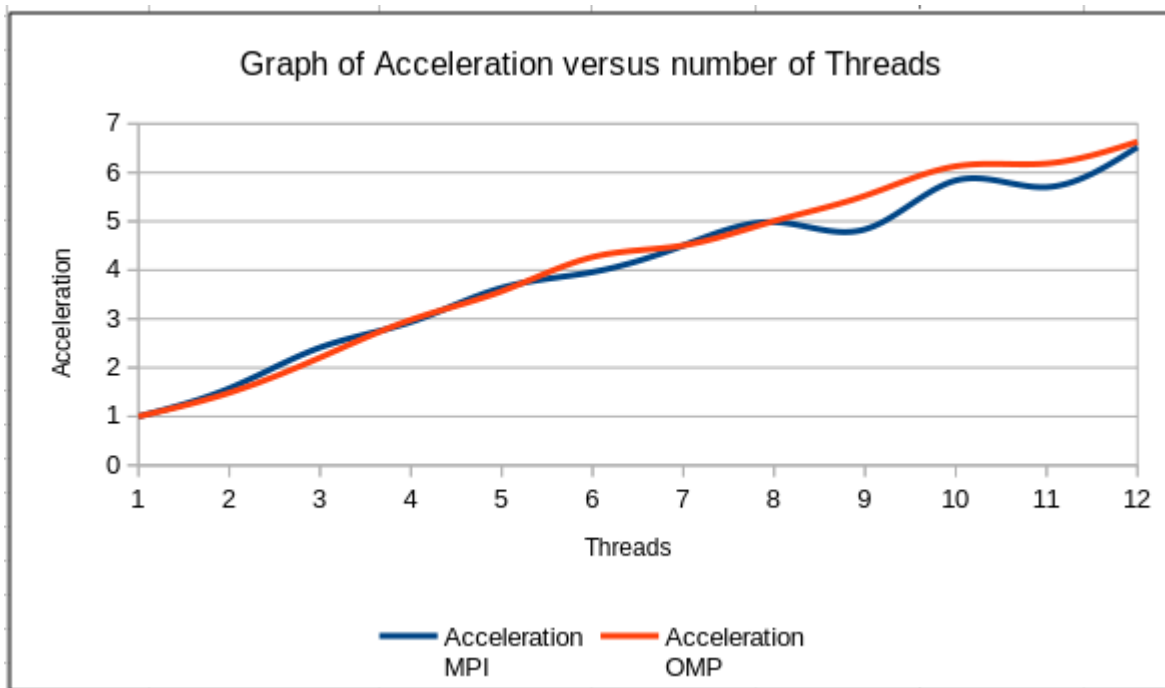


График №6

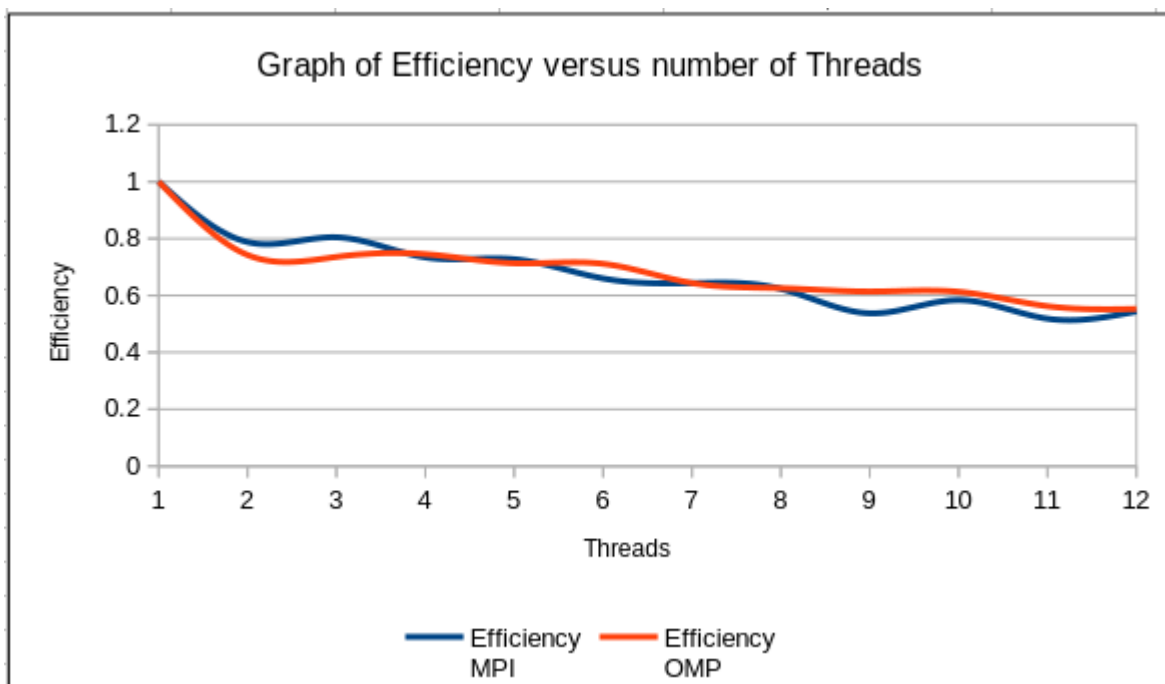


Таблица с данными

Threads	<u>OMP:</u>	Attempt #1	Attempt #2	Attempt #3	Attempt #4	Attempt #5		<u>Average OMP</u>		<u>Acceleration OMP</u>	<u>Efficiency OMP</u>
1		0.2768	0.156039	0.15816	0.265841	0.153659		0.2020998		1	1
2		0.144443	0.137419	0.128165	0.134375	0.136255		0.1361314		1.484593561809	0.7422967809
3		0.096956	0.091505	0.089183	0.08977	0.090543		0.0915914		2.206536858264	0.73551228609
4		0.072454	0.066712	0.066099	0.067423	0.066299		0.0677974		2.98093732208	0.74523433052
5		0.058176	0.05689	0.059458	0.056242	0.052607		0.0566746		3.565967823328	0.71319356467
6		0.04842	0.047538	0.049916	0.046938	0.043991		0.0473606		4.267255904697	0.71120931745
7		0.045841	0.044178	0.04421	0.04407	0.045944		0.0448486		4.506267754177	0.64375253631
8		0.041772	0.039037	0.039579	0.03911	0.042374		0.0403744		5.00564218911	0.62570527364
9		0.037701	0.034688	0.038124	0.03471	0.037687		0.036582		5.52456946039	0.61384105115
10		0.034139	0.031358	0.033974	0.031335	0.034074		0.032976		6.128693595342	0.61286935953
11		0.031016	0.039083	0.033816	0.028494	0.030961		0.032674		6.185340025709	0.5623036387
12		0.028562	0.026614	0.028217	0.030569	0.03843		0.0304784		6.630918945876	0.55257657882
	<u>MPI:</u>	Attempt #1	Attempt #2	Attempt #3	Attempt #4	Attempt #5		<u>Average MPI</u>		<u>Acceleration MPI</u>	<u>Efficiency MPI</u>
		0.173674	0.169045	0.169585	0.169459	0.168508		0.1700542		1	1
		0.145507	0.100939	0.084773	0.085233	0.123239		0.1079382		1.575477449133	0.78773872457
		0.056879	0.057043	0.079464	0.079464	0.079464		0.0704628		2.413389760271	0.80446325342
		0.052412	0.057043	0.058401	0.058825	0.062937		0.0579236		2.935836170404	0.7339590426
		0.049065	0.049034	0.044035	0.050035	0.04157		0.0467478		3.637694180261	0.72753883605
		0.046113	0.03834	0.04157	0.048376	0.040376		0.042955		3.958891863578	0.6598153106
		0.040931	0.033414	0.038803	0.042803	0.032803		0.0377508		4.504651557053	0.64352165101
		0.039289	0.023075	0.038771	0.034702	0.034702		0.0341078		4.985786242443	0.62322328031
		0.034494	0.037609	0.039233	0.030233	0.03428		0.0351698		4.835233637951	0.53724818199
		0.021638	0.033258	0.033002	0.027002	0.030752		0.0291304		5.837688462912	0.58376884629
		0.029264	0.029909	0.030752	0.030752	0.02847		0.0298294		5.700892408161	0.5182629462
		0.027479	0.023542	0.024523	0.024781	0.030021		0.0260692		6.523184447547	0.54359870396

Код программы OMP

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <omp.h>
4
5
6  int main()
7  {
8      const int count = 100000000;
9      const int random_seed = 920215;
10     const int max_threads = 12;
11     int max = -1;
12
13     srand(random_seed);
14
15     int *array = 0;
16     array = (int *)malloc(count * sizeof(int));
17     for (int i = 0; i < count; i++) { array[i] = rand(); }
18
19
20     for (int threads = 1; threads <= max_threads; threads++)
21     {
22         double start_time = omp_get_wtime();
23
24         #pragma omp parallel num_threads(threads) shared(array, count) reduction(max : max) default(none)
25         {
26             #pragma omp for
27             for (int i = 0; i < count; i++)
28             {
29                 if (array[i] > max)
30                 {
31                     max = array[i];
32                 };
33             }
34         }
35
36         printf("Threads: %d, Execution time: %f seconds\n", threads, omp_get_wtime() - start_time);
37     }
38     free(array);
39
40     return 0;
41 }
```

Код программы MPI

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <mpi.h>
4
5  int main(int argc, char** argv)
6  {
7      int ret = -1;
8      int size = -1;
9      int rank = -1;
10
11     const int count = 100000000;
12     const int random_seed = 920215;
13
14     int* array = NULL;
15     int lmax = -1;
16     int max = -1;
17
18     ret = MPI_Init(&argc, &argv);
19     if (!rank) { printf("MPI Init returned (%d);\n", ret); }
20
21     MPI_Comm_size(MPI_COMM_WORLD, &size);
22     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
23
24     array = malloc(count * sizeof(int));
25
26     if (!rank) {
27         srand(random_seed);
28         for (int i = 0; i < count; i++) { array[i] = rand(); }
29     }
30
31     MPI_Bcast(array, count, MPI_INTEGER, 0, MPI_COMM_WORLD);
32
33     const int wstart = (rank) * count / size;
34     const int wend = (rank + 1) * count / size;
35
36     MPI_Barrier(MPI_COMM_WORLD);
37
38     double start_time = MPI_Wtime();
39
40     for (int i = wstart; i < wend; i++)
41     {
42         if (array[i] > lmax) { lmax = array[i]; }
43     }
44
45     double end_time = MPI_Wtime();
46
47     MPI_Barrier(MPI_COMM_WORLD);
48
49     MPI_Reduce(&lmax, &max, 1, MPI_INTEGER, MPI_MAX, 0, MPI_COMM_WORLD);
50
51     ret = MPI_Finalize();
52
53     if (!rank) {
54         printf("Threads: %d, Execution time: %f seconds\n", size, end_time - start_time);
55         printf("Max element: %d\n", max);
56     }
57
58     free(array);
59
60     return(0);
61 }
```

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 рассчитана 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 0.5525. При добавлении второго потока происходит резкое снижение с 1 до 0.7422, 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 6.6309. График ускорения монотонно растет в интервале $[0; 0.7]$.

При переходе от одного потока к двум в использовании MPI, эффективность вначале резко снижается с 1 до 0.07877, после чего продолжает уменьшаться в интервале $[0; 0.065]$. Минимальное значение эффективности достигает 0.05182. В то время как уровень эффективности снижается, ускорение по-прежнему монотонно возрастает, начиная с 1 и достигая максимального значения 6.5231.

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 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 происходило вычисление времени выполнения работы программы до того, как все процессы завершили свою работу. После установки барьеров вокруг части кода, которая отвечает за поиск максимального элемента, время выполнения программы нормализовалось.

Несмотря на то, что программа написанная для OMP достигла большего ускорения и большей эффективности, время выполнения на MPI для каждого потока меньше. Из чего можно сделать вывод, что в рамках поставленной задачи использование MPI будет более выгодно. Хотя разница во времени незначительная, но в любом случае алгоритм с MPI справился с этой задачей быстрее.

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