МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение

высшего образования

« Национальный исследовательский ядерный университет «МИФИ»»

ЛАБОРАТОРНАЯ РАБОТА №5:

«Технология MPI. Введение»

Аверин Владислав

Группа Б19-505

Ноябрь, 2021

Содержание

Характеристики лабораторного оборудования

Процессор: 11th Gen Intel Core i7-1185G7 3.00Ghz (8 CPUs)

RAM: 16Гб DDR4

Операционная система: OS Linux Manjaro KDE Plasma 5.22.5; версия ядра: 5.10.68-1-MANJARO (64-бита)

Используемая технология MPI: OpenMPI for Linux 4.1.1 (диспетчер mpirun)

Компилятор (основной): GCC 11.1.0

Редактор кода: Visual Studio Code 1.60.1

Параметры командной строки: -O3 -fopenmp (ради чистоты эксперимента для использования функции omp_get_wtime(), которая измеряла время в 1 лабораторной)

==========

Что касается проверяемых массивов: они такие же, какие использовались в 1 лабораторной:

Начальное значение для инициализации ГПСЧ (srand() из <stdlib.h>): 920215

Шаг для нового значения для инициализации ГПСЧ: 1000

Кол-во различных массивов: 10

Размер массивов: 1e8

Различные подходы к рассмотрению задачи

Одной из главных особенностей технологии MPI являются способы коммуникации процессоров друг с другом (т.к. в отличие от OpenMP здесь у нас нет общей памяти). Так что при анализе любого кода, написанного на MPI, способы таймирования можно абстрактно разделить на три различных подхода:

- вариант, ***учитывающий*** коммуникационные затраты между процессорами (что немаловажно для MPI, но не так критично для OpenMP).
- вариант, который их ***игнорирует***. Однако, это не правильно: ведь тогда мы опустим главную часть реализации технологии MPI.
- и третий, при котором в добавок ко второму варианту в реализацию для обеих технологий мы включаем еще и генерацию массива. Тогда код на MPI можно заметно ускорить, избавившись от пересылки массива главным процессором и вместо этого генерируя его в каждом из потоков заново. Но как по мне, это немного не соответствует назначению проверяемого алгоритма - очень странно реализовывать алгоритм отдельной программой (подразумевается, что это часть какой-то другой программы), а значит какой-то из процессоров (мастер) все равно должен получить на вход данные (в нашем случае, массив). А при отдельной генерации в каждом из процессов мы немного «читерим», облегчая себе задачу, но из-за этого программа лишается «встраиваемости» в другой код.

Все эти варианты (программные коды, анализ результатов) будут рассмотрены далее по порядку.

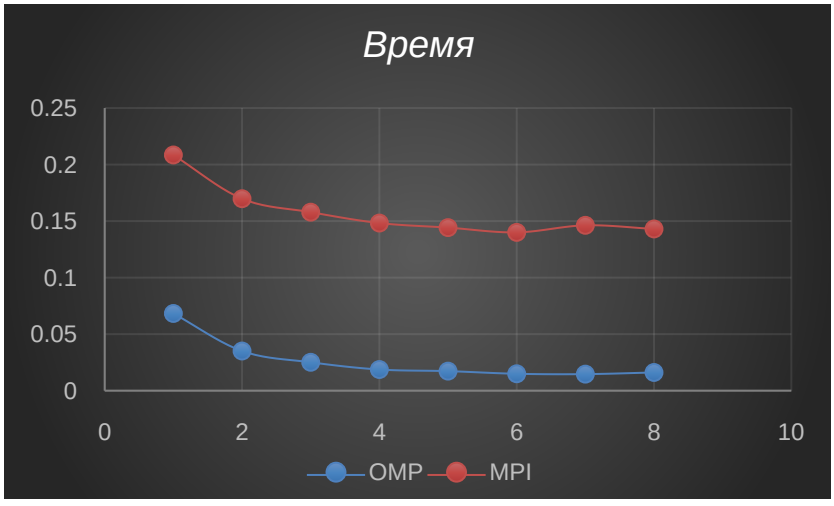
Реализация, учитывающая коммуникационные затраты

При таком сравнении мы будем измерять работу MPI-алгоритма вместе с пересылкой массивов по другим процессам, то есть без учитывания генерации основного массива мастер-потоком:

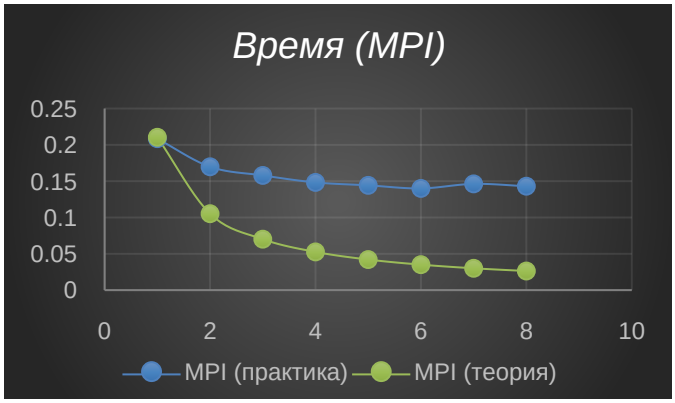
```
start = omp_get_wtime(); // 1 вариант

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

//start = omp_get_wtime(); // 2 вариант
const int wstart = (rank)*count / size;
const int wend = (rank + 1) * count / size;

for (int i = wstart; i < wend; i++)
{
    if (array[i] > lmax)
{
    if (array[i] > lmax)
    {
        lmax = array[i];
    }
}

//end = omp_get_wtime();    // 2 вариант

MPI_Reduce(&lmax, &max, 1, MPI_INTEGER, MPI_MAX, 0, MPI_COMM_WORLD);

ret = MPI_Finalize();
end = omp_get_wtime(); // 1 вариант

printf("time: %f\n", end - start);
```

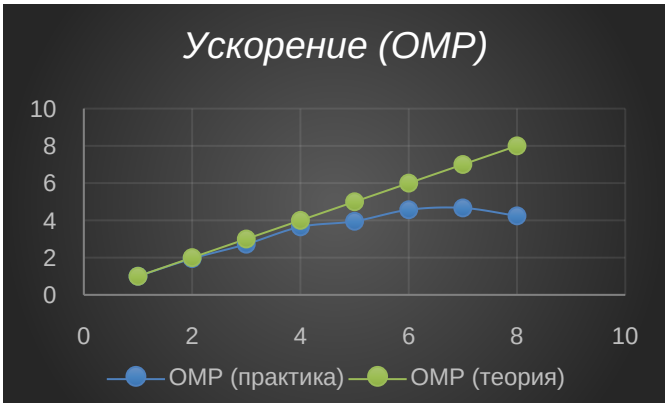
И если сравнить полученные результаты с таблицей из 1 лабораторной, можно увидеть что-то такое:

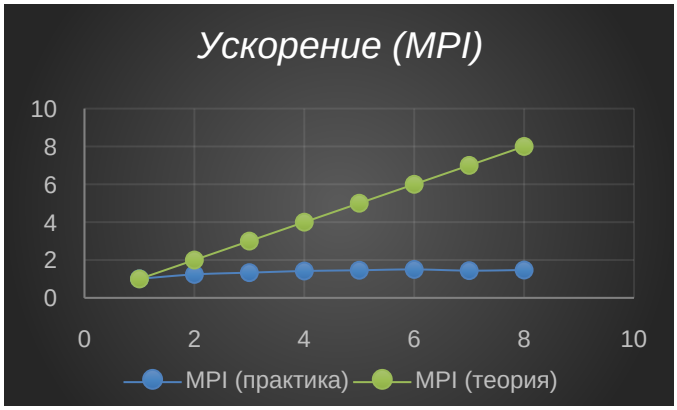
	OMP	MPI
Cons. AVG	0,068335	0,210432
1	0,0683	0,208501
2	0,035093	0,16981
3	0,025122	0,15791
4	0,018718	0,148281
5	0,017295	0,144241
6	0,014953	0,140016
7	0,014641	0,146274
8	0,016112	0,143071

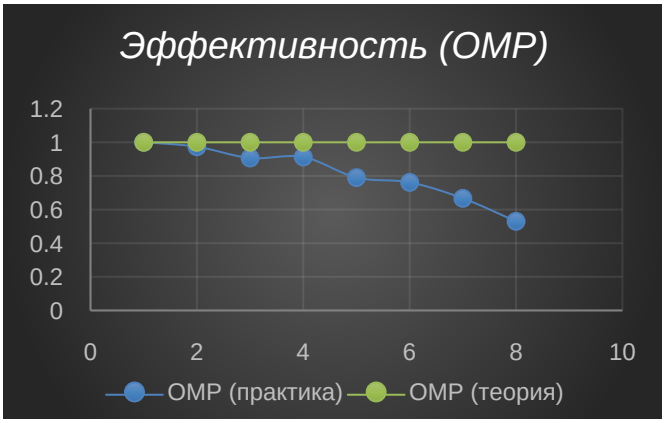


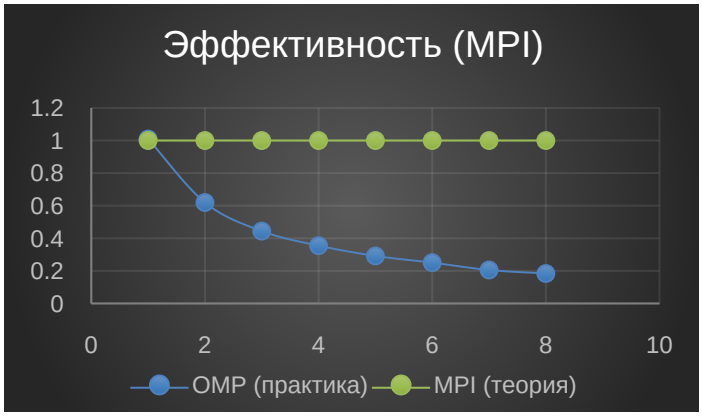
Как мы видим, время работы алгоритма на MPI в несколько раз больше, чем время на OMP. Кроме того, ни о каком ускорении в MPI не может идти и речи. Оно и понятно: пересылка всего массива занимает гораздо больше времени, чем просто его просмотр и поиск максимума.











Толку от такого таймирования для нас мало. Тогда давайте не учитывать вообще никакой коммуникации между процессами.

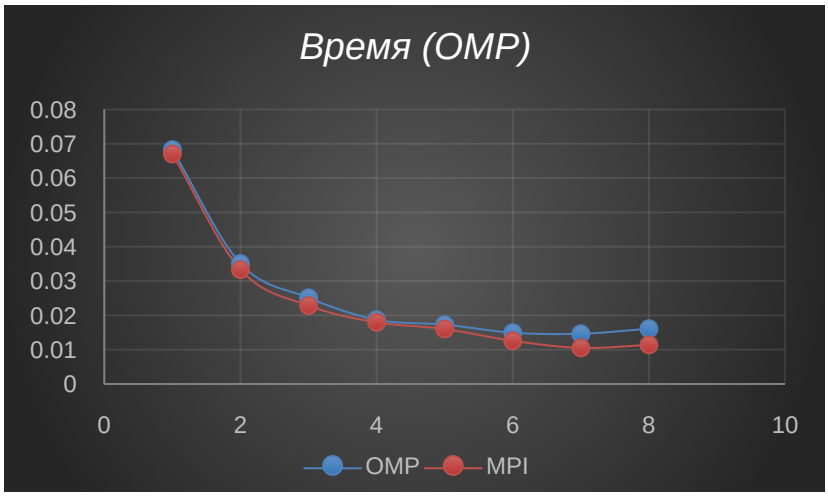
Реализация, игнорирующая коммуникационные затраты

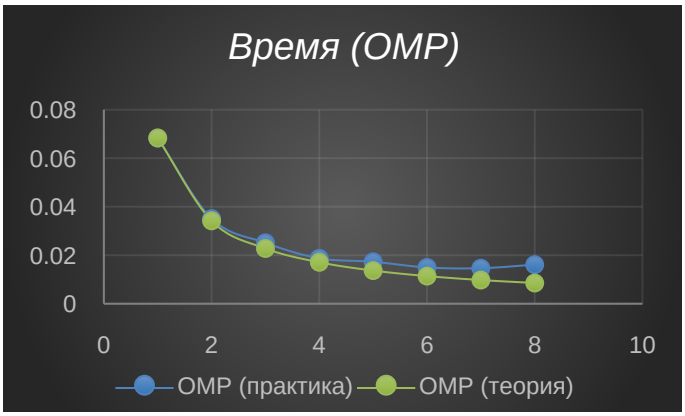
В этом случае мы будем игнорировать пересылку в MPI, а включим только операцию редукции (однако, мы все еще сравниваем с результатами 1 лабы, где в таймирование включалась и генерация потоков):

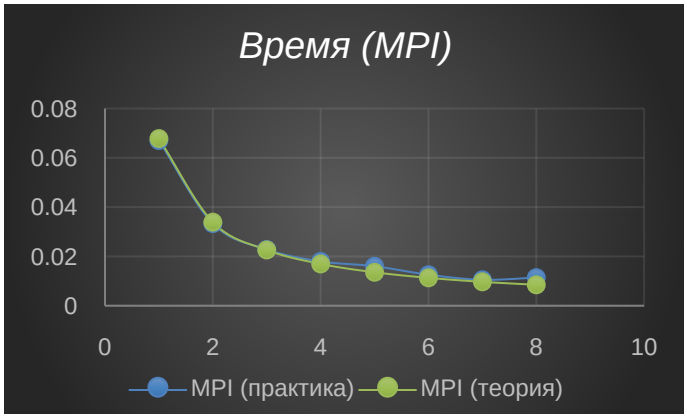
```
start = omp_get_wtime(); // 2 вариант
const int wstart = (rank)*count / size;
const int wend = (rank + 1) * count / size;

for (int i = wstart; i < wend; i++)
{
    if (array[i] > lmax)
{
    if (array[i] > lmax)
    {
        lmax = array[i];
    }
}



MPI_Reduce(&lmax, &max, 1, MPI_INTEGER, MPI_MAX, 0, MPI_COMM_WORLD);

end = omp_get_wtime();    // 2 вариант
```

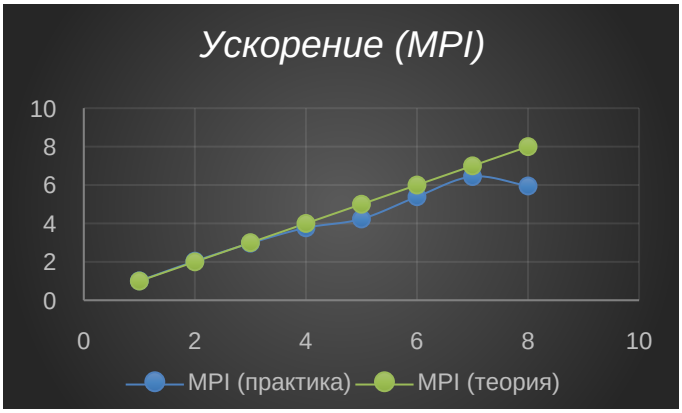
И практические результаты дают нам следующее:

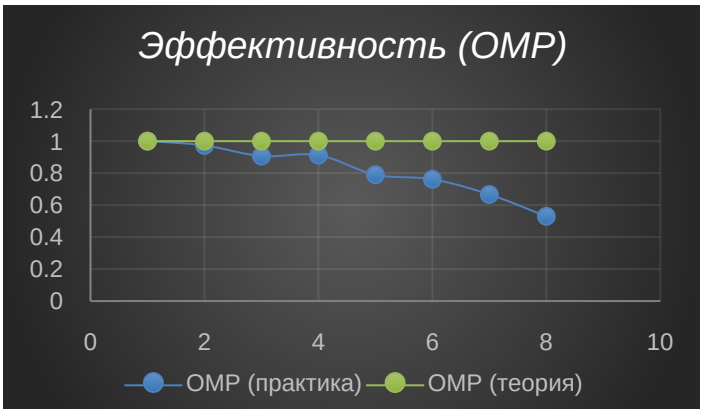
	OMP	MPI
Cons. AVG	0,068335	0,067825
1	0,0683	0,067014
2	0,035093	0,033319
3	0,025122	0,022831
4	0,018718	0,017981
5	0,017295	0,016024
6	0,014953	0,012581
7	0,014641	0,010529
8	0,016112	0,011397

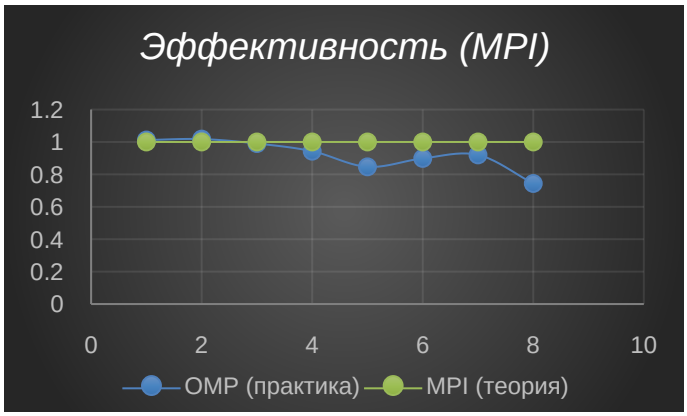












Как и ожидалось, время для OMP почти теоретическое, и меньше, чем для OMP, что можно было бы расценивать как успех, однако толку то от такого сравнения, если главная часть реализации кода на MPI проигнорирована... Это не адекватное сравнение, что нам еще раз показывает, что генерация потоков в OMP для данной задачи намного выгоднее.

Реализация, учитывающая генерацию массива

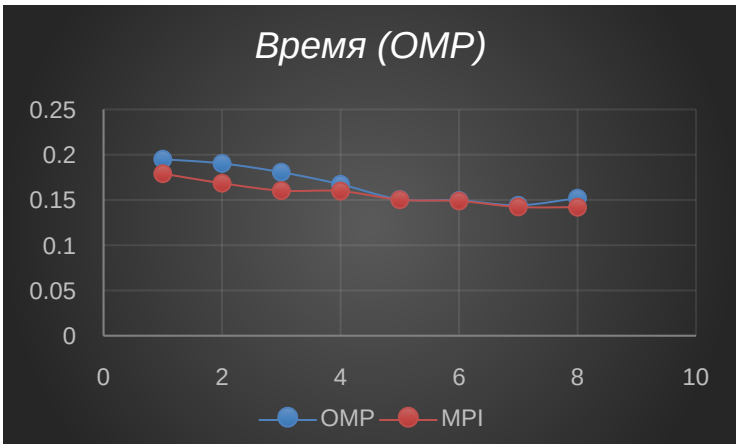
А теперь учтем в обеих реализациях создание массива. Только если для варианта на OMP он создается один раз,а после сгенерированные в параллельной области нити разделяют между собой его элементы, то для MPI требуется пересылка мастером сгенерированного массива. Однако немного схитрим и просто будем генерировать в каждом процессе этот же массив, без дополнительной траты ресурсов на пересылку (точнее, с более хорошей балансировкой задач между процессами):

```
start = omp_get_wtime(); // 3 вариант

/* Initialize the MPI */
ret = MPI_Init(&argc, &argv);

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);

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

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

//start = omp_get_wtime(); // 1 вариант

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

//start = omp_get_wtime(); // 2 вариант
const int wstart = (rank)*count / size;
const int wend = (rank + 1) * count / size;
```

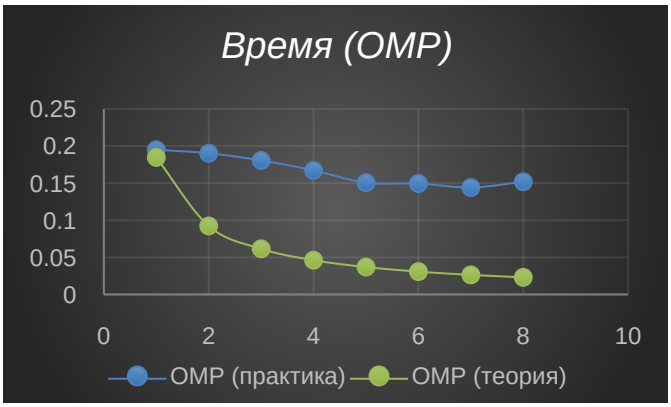
```
for (int i = wstart; i < wend; i++)
{
    if (array[i] > lmax)
    {
        if (array[i] > lmax)
        {
            lmax = array[i];
        }
    }

    MPI_Reduce(&lmax, &max, 1, MPI_INTEGER, MPI_MAX, 0, MPI_COMM_WORLD);

    //end = omp_get_wtime(); // 2 вариант

    ret = MPI_Finalize();
    end = omp_get_wtime(); // 1 вариант
```

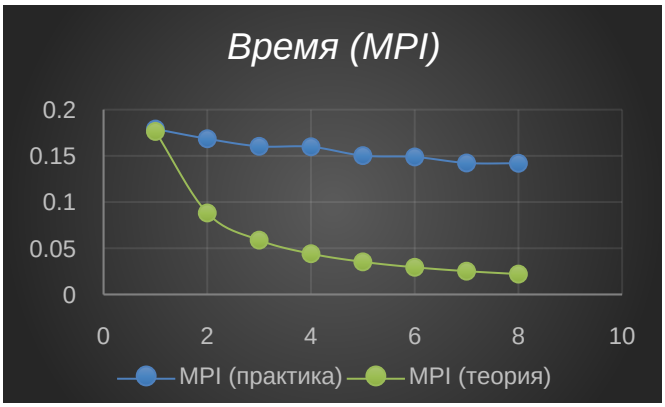
В этом случае результаты такие:

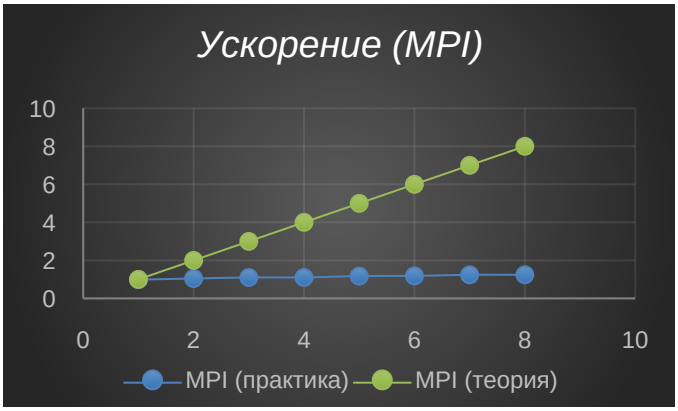
	OMP	MPI
Cons. AVG	0,18469	0,17639
1	0,19504	0,179
2	0,19031	0,16849
3	0,18064	0,16031
4	0,1673	0,15981
5	0,15065	0,150241
6	0,14953	0,14881
7	0,14416	0,14229
8	0,15194	0,14197

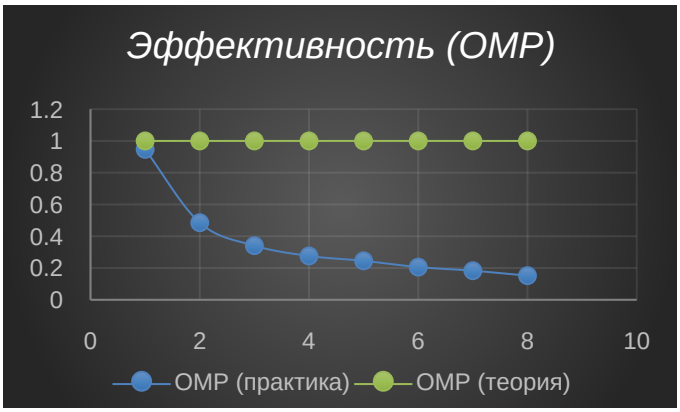


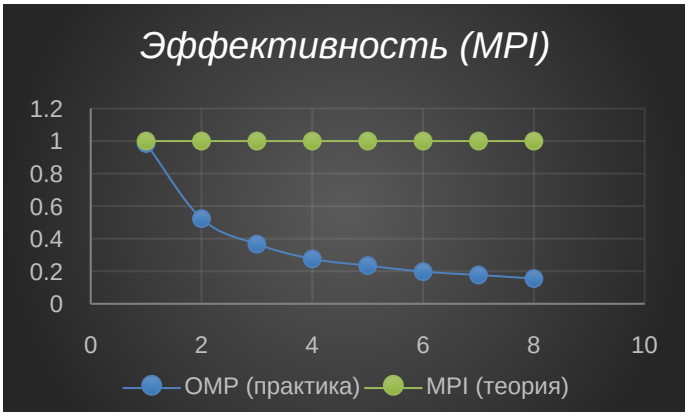
По времени работы они сопоставимы, однако:











Как и стоило ожидать, вдовесок к первому опыту, мы похерили и результаты таймирования OMP, что логично: мы должны измерять работу конкретно алгоритма, принимая во внимание только то, что для его реализации нужно при использовании той или иной технологии.

Выводы

Как мы поняли, о том, чтобы адекватно сравнить эффективность реализаций на OMP и на MPI не может идти и речи: для такой простой задачи, как нахождение максимума идеально подойдет именно OMP, т.к. MPI требует несравнимо больших коммуникационных затрат на передачу массива. Мы можем откинуть эту часть алгоритма, но тогда и картина будет недостаточно полной. Поэтому нужно иметь ввиду несколько пунктов при выборе технологии распараллеливания:

1. Насколько сложны параллельные вычисления по сравнению с трудозатратами на передачу данных?
2. Как именно организован алгоритм: принимает ли он какие-либо параметры как отдельная программа (а-ля библиотека, подключающаяся отдельно), либо должна быть включена напрямую в программу?

И на данном примере мы смогли прочувствовать эту разницу между двумя подходами к распараллеливанию: на общей памяти, и на разделенной памяти.