МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение

высшего образования

« Национальный исследовательский ядерный университет «МИФИ»»

ЛАБОРАТОРНАЯ РАБОТА №6:

«Коллективные операции в MPI»

Аверин Владислав

Группа Б19-505

Декабрь, 2021

Содержание

Характеристики лабораторного оборудования.....2
Различные подходы к рассмотрению задачи.....3
Сравнительный теоретический анализ всех подходов.....5
Реализация, практические результаты и таблицы.....8
Выводы.....13

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: -fopenmp (ради чистоты эксперимента без оптимизации и с использованием библиотеки OMP для функции omp_get_wtime(), которая измеряла время в 3 лабораторной)

==========

Что касается проверяемых массивов: они такие же, какие использовались в 3 лабораторной:

Начальное значение для инициализации ГПСЧ (srand() из <stdlib.h>): 666666

Шаг для нового значения для инициализации ГПСЧ: 1000

Кол-во различных массивов: 100

Размер массивов: 1e7

Различные подходы к рассмотрению задачи

Любую параллельную сортировку с использованием технологии MPI (да и других библиотек) можно реализовать двумя различными подходами: «глупо» - в лоб, напрямую; и «умно» (это разделение и классификация лично для меня, и я выбрал такие названия из-за их идеи и возможности применения абсолютно к любому типу сортировки). Мы хотим, чтобы естественно в основе параллельной сортировки лежала одноименная последовательная, которая будет выполняться независимо всеми процессорами, поэтому вопрос, который возникает при распараллеливании: 'Как использовать сортировку не для всей последовательности, а для ее части, и так, чтобы от этого был толк? (выполнялась независимость от других процессоров)'. Эти две идеи отвечают на этот вопрос, но с несколько разных точек зрения.

- ***«Глупая»*** идея состоит в том, чтобы разбить массив на p одинаковых блоков (p - количество процессоров), локально отсортировать каждый из блоков (что возможно сделать параллельно и независимо каждым процессором), собрать все блоки на одном из процессоров, а затем провести слияние всех блоков (а т.к. блоки уже упорядочены, то сложность такой операции всего-навсего $O(n)$).

Плюсы такой реализации: очень простой программный код (поэтому я и назвал ее «глупой» за свою прямолинейность); получение какого-то ускорения (какого именно, покажет практика);

Минусы: в данном случае это уже не совсем сортировка Шелла (т.к. используется сторонний метод слияния структур); неравномерность распределения нагрузки па процессорам (черт его знает, как там массив устроен: у кого-то может быть вообще изначально упорядоченный блок);

- ***«Умная»*** реализация включает ту же самую идею разделения блоков по процессорам, однако в этом случае упорядочивание элементов идет В НАЧАЛЕ выполнения алгоритма. Результатом такого упорядочивания должны являться блоки на процессорах, которые сами по себе локально не отсортированы, но любой элемент КАЖДОГО блока лежит в строго определенном характеристическом интервале (для массива из чисел это какое-то опорное число/числа). Таким образов, отсортировав эти блоки, нам достаточно будет последовательно конкатенировать их (при условии, что блоки упорядочены глобально, т.е опорные элементы сами по себе тоже упорядочены)

Примеры такой предварительной упорядоченности элементов приводятся в книге В.П. Гергеля *«Теория и практика параллельных вычислений»*, например, для нашей же сортировки Шелла или для параллельной быстрой сортировки. В идеале, для того алгоритма процессоры должны образовывать полный граф либо N-мерный гиперкуб, что накладывает большое ограничение.

Плюсы: понтово =), я серьезно, далее будет теоретический анализ всех подходов, а почему в этой книге выбрали именно такой способ предварительной упорядоченности, мне не до конца понятно.

Минусы: плохая работа на не полном графе или на не N-мерном гиперкубе (в этом случае на каждом шаге какие-то процессоры будут неизбежно простаивать); сложность в реализации (синхронизация, возможность работы для произвольного кол-ва процессоров, большое количество шагов для такой упорядоченности), а самое главное - даже в самой книге такая реализация на быстрой сортировке показала не утешающие $\sim 1{,}48$ ускорения, что уж говорить про Шелла. Все из-за огромного количества коммуникации между процессорами на этапе упорядочивания: пока мы создадим блоки, которые можно локально сортировать, пройдет больше времени, чем мы их сортировать будем; к тому же, блоки по размеру получатся не одинаковыми, поэтому это еще сильнее добавляет элемент случайности в нагруженность процессоров.

Конкретно для сортировки Шелла можно привести еще один способ реализации: такой же, как и в 3 лабораторной. На каждом шаге мы рассылаем соответствующие элементы по процессорам, они их сортируют с помощью Insertion Sort, а далее мастер собирает их все в том же порядке. Потом с другим шагом и так далее, т.е. концептуально идея та же: разделить каждую итерацию последовательного алгоритма между процессорами. НО! А давайте подумаем, что из этих трех вариантов выгоднее?

Сравнительный теоретический анализ всех подходов

Мы выделили три различных варианта распараллеливания, давайте их проанализируем по временным затратам (в особенности, на коммуникацию)

Вариант для сортировки Шелла требует из 3 лабораторной требует:

A) Рассылку элементов по процессорам. Сколько именно и как часто? Для стандартного алгоритма Шелла с начальным шагом $\mathrm{step} = \mathrm{N}/2$ step $/= 2$ пересылок должно быть естественно $\log_2 \mathrm{N}$. Учтем также, что в среднем будет пересылаться N/p элементов. Итого $\log_2 \mathrm{N}$ пересылок по N/p элементов.
B) Сортировка вставками локальных подмассивов на каждой итерации. Тут трудно сказать, насколько это затратно, так как с уменьшением шага сортировка все реже будет переставлять элементы, и говорить, что ее $\mathrm{O}(\mathrm{N}^2)$ скорее всего не корректно.Но можно воспользоваться результатами 3 лабораторной: ускорения более чем в $4{,}49$ раза добиться все равно не удалось, а это с использованием OpenMP (т.е. практически без учета пункта A - коммуникационных затрат).
C) Выполнение сбора всех блоков на мастере: это пересылка в среднем N/p элементов, причем все пересылки от процессоров должны организовать новый массив в той же упорядоченности, что и до рассылки.

Для «умного» варианта все немного лучше, но все равно печально:

A) Предварительное упорядочивание элементов: требует начальную рассылку (аналогично с (A) в пред. варианте) и $\log_2 \mathrm{p}$ итераций (именно p) на каждой из которых пересылается в среднем $\mathrm{N}/2\mathrm{p}$ элементов другому процессору (в среднем половина элементов блока).
B) Сортировка Шеллом каждого блока: Выполняется с такой же сложностью, что и последовательная сортировка, но с размер массива у нас $\sim \mathrm{N}/\mathrm{p}$, то есть для нашего выбора шага $\mathrm{O}(\mathrm{N}^2/\mathrm{p}^2)$.
C) Конкатенирование блоков: пересылка блока на мастер - N/p элементов.

Что же до «глупого» способа:

A) Начальная рассылка блоков (аналогично с (A) в пред. варианте).
B) Сортировка Шеллом каждого блока (аналогично (B), но тут точно будет N/p элементов).
C) Слияние блоков: пересылка N/p элементов на мастер и выполнение мастером слияния (сложность $\mathrm{O}(\mathrm{N})$)

ИТОГ:

Алгоритм по аналогии с 3 лабораторной сможет дать максимум ускорение в $4{,}49$ (т.к. коммуникация в MPI вещь намного более дорогая, чем в OMP), а с учетом наших коммуникаций, коих о-о-о-очень много (и по общему кол-ву пересылок, и по общему кол-ву пересылаемых элементов), это число явно не дойдет до чего-то фантастического.

«Умный» алгоритм, судя по книжке, не дает ускорения вовсе: время на коммуникацию намного превышает время выполнения одного распараллеленного цикла последовательной сортировки, а это с учетом того, что там коммуникации меньше, чем в алгоритме из 3 лабораторной, так что я не уверен, что сортировка вставками сможет настолько ускорить работу, чтобы покрыть затраты времени на пересылку и сбор блоков на каждом из итераций (к примеру, если $\mathrm{N} = 1\mathrm{e}8$, кол-во итераций уменьшения шагов будет порядка 26, то есть это 26 пересылок туда-обратно, не мало)

А вот с «глупой» реализацией все проще: там необходима пересылка элементов в начале, и одна пересылка блока с процессора на мастер. То есть в данном случае коммуникация будет минимальной, что позволяет нам говорить, что основная сложность алгоритма - это сортировка Шеллом каждого блока, т.е. $\mathrm{O}(\mathrm{N}^2/\mathrm{p}^2)$ (а может и не позволяет ¯_(ツ)_/¯, но вроде все логично)

Реализация, практические результаты и таблицы

В связи со всех вышеперечисленным реализуем «глупый» вариант распараллеливания.

(Я не очень парился насчет эффективности и красоты написанного, здесь я просто писал на ходу, сразу как считал нужным. Поэтому, не обессудьте за рациональность и понятность ;))

Мастер-процессор рассылает блоки массива по другим процессорам, сортирует свой блок последовательным Шеллом, А затем собирает на место старых блоков отсортированные блоки с других процессоров:

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

const int wstart = (rank)*size / procCount;
const int wend = (rank + 1) * size / procCount;
const int lenght = wend - wstart;

/* Send the array parts to all other processors */
if (!rank)
{
    for (int i = 1; i < procCount; ++i)
    {
        const int mpiStart = (i * size) / procCount;
        const int mpiEnd = (i + 1) * size / procCount;
        const int mpiLenght = mpiEnd - mpiStart;
        MPI_Send(array + mpiStart, mpiLenght, MPI_INT, i, 0, MPI_COMM_WORLD);
    }

    ShellSort(array, array + wend);
    for (int i = 1; i < procCount; ++i)
    {
        const int mpiStart = (i * size) / procCount;
        const int mpiEnd = (i + 1) * size / procCount;
        const int mpiLenght = mpiEnd - mpiStart;
        MPI_Recv(array + mpiStart, mpiLenght, MPI_INT, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
    //for (int j = 0; j < size; ++j)
    //    printf("%d, ", array[j]);
}
else
```

Для остальных процессоров почто то же самое - они принимают во временной буфер соответствующей длины блок массива, сортируют его Шеллом, и отсылают обратно:

```
  else
  {
      int *buf = (int *)malloc(lenght * sizeof(int));
      MPI_Recv(buf, lenght, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      ShellSort(buf, buf + lenght);
      MPI_Send(buf, lenght, MPI_INT, 0, 0, MPI_COMM_WORLD);
      free(buf);
  }
```

Остается произвести слияние блоков в один новый отсортированный массив (этот код можно добавить в код к мастер-процессору:

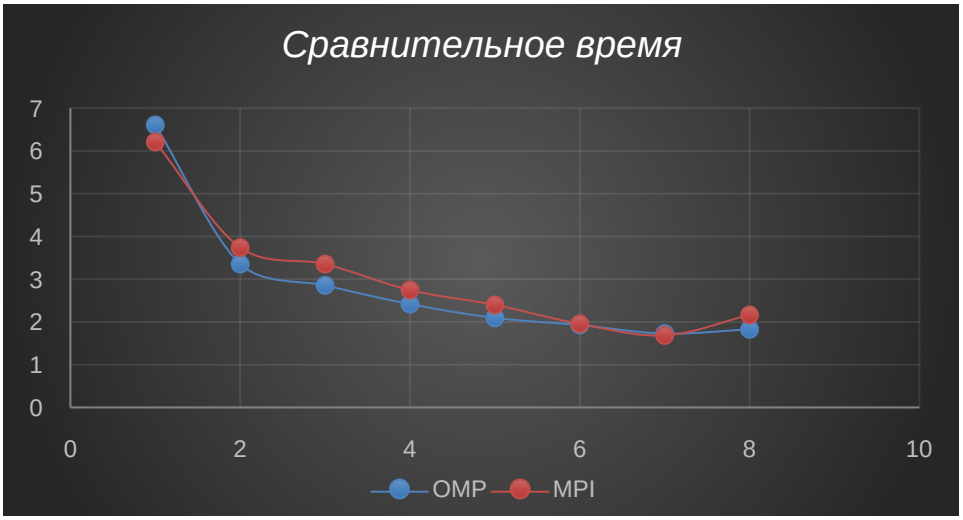
```
    }
    //for (int j = 0; j < size; ++j)
    //    printf("%d, ", array[j]);
    int* indexes = (int *)malloc(procCount * sizeof(int));
    /* Create list of the indexes of the blocks */
    for (int i = 0; i < procCount; ++i)
        indexes[i] = i * size / procCount;
    int *resultArray = (int*)malloc(size * sizeof(int));
    resultArray = mergeVectors(indexes, procCount, array);
}
```

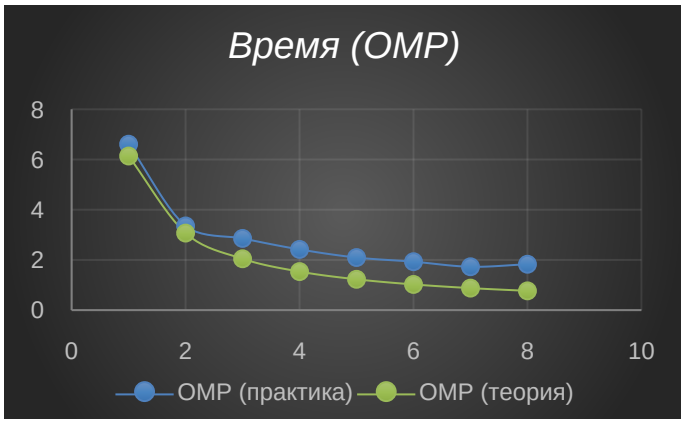
Функция int* mergeVectors(int* indexes, int procCount, int* oldArray) принимает массив индексов размерности кол-ва процессоров, где под i-тым индексом находится индекс (пардон за тавтологию) начала i-того блока (чтобы можно было проверять, что блок не пуст), кол-во процессоров и указатель на старый массив. Отсортированный массив она вернет в качестве результата.

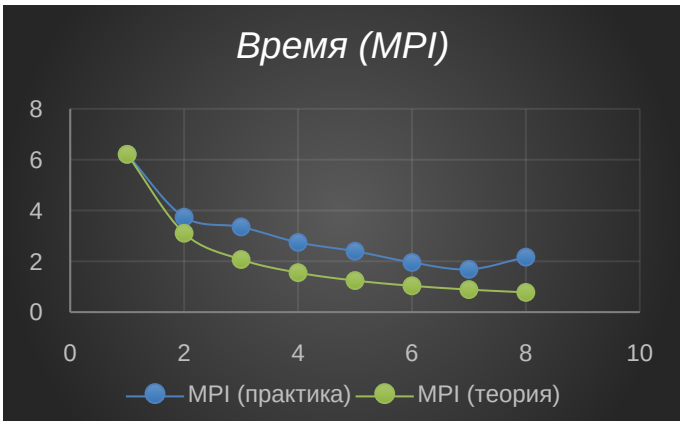
Так как в лабораторной 3 мы поняли, что от частоты встреч дубликатов мало что зависит (время для частоты 0.1 и 10 почти не отличались), то будем использовать массивы размерности 1e7 и с частотой 1 (т.е. диапазон возможных чисел ГПСЧ совпадает с размерностью массива). Такой размер взят из соображений коммуникационных затрат: для малых размеров они в MPI явно будут большими относительно самого распараллеливания.

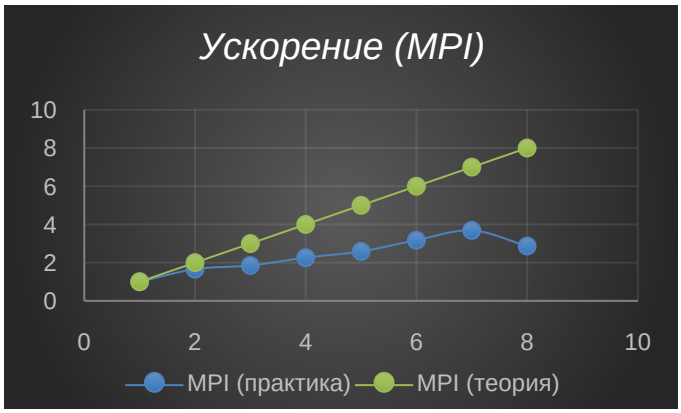
Экспериментальные полученные значения представлены ниже (и это, на самом деле, немного печально, я рассчитывал на большее...):

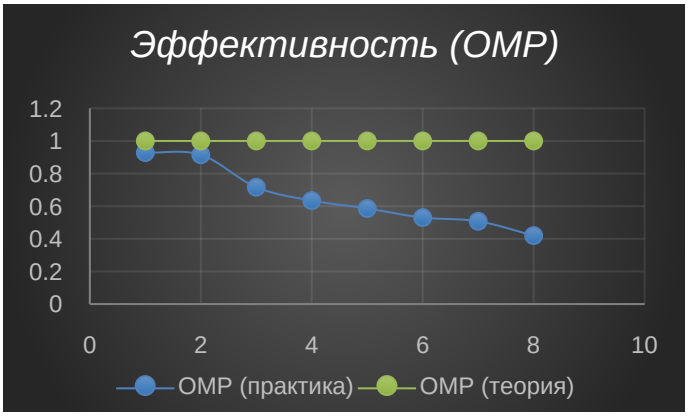
	OMP	MPI
Cons. AVG	6,13873	6,21043
1	6,60915	6,21043
2	3,35351	3,739
3	2,85631	3,3532
4	2,41816	2,7459
5	2,09492	2,3984
6	1,92882	1,956
7	1,72754	1,68452
8	1,82752	2,16453

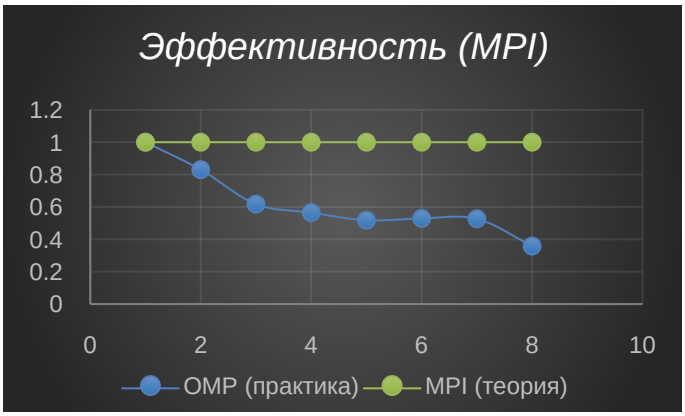








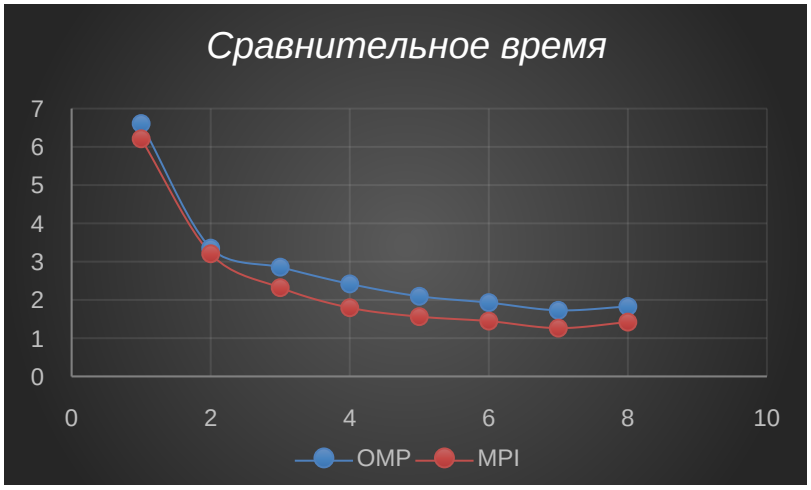


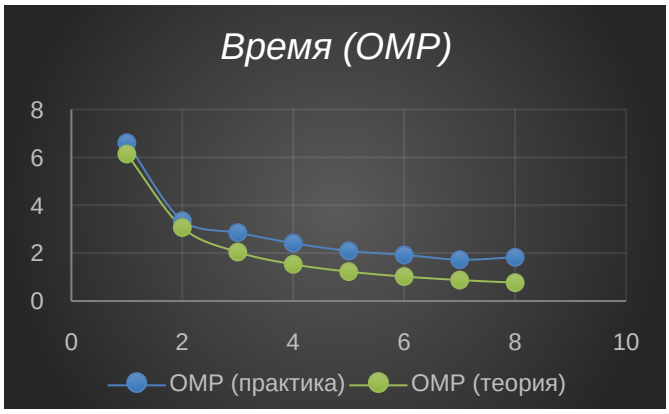


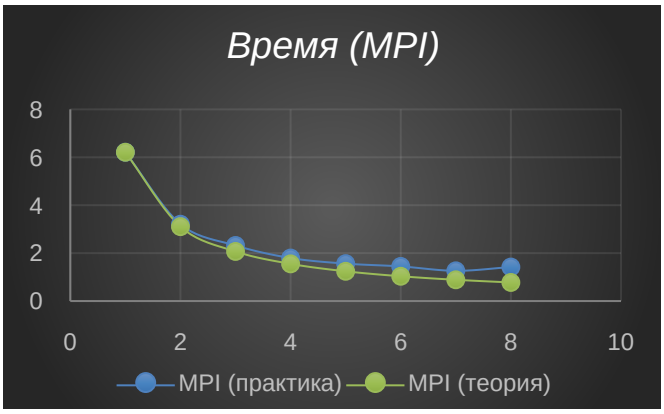
Это не есть прямо таки очень хорошо, я считал, что коммуникационные затраты не будут такими уж большими. Однако эксперименты и последующее таймирование алгоритма по частям показало, что на пересылку с мастера на остальные процессоры требует не мало времени относительно непосредственно сортировки.

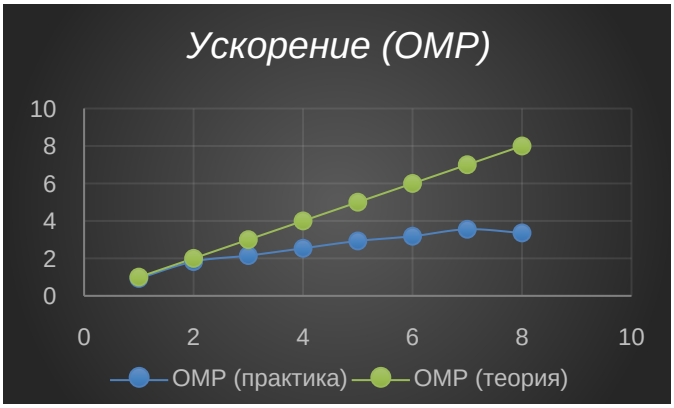
Можно еще немного схитрить и сделать финт ушами по аналогии с 5 лабораторной. Если предположить, что на вход программа получает не сам массив а значение инициализатора ГПСЧ, то мы можем генерировать весь массив целиком одновременно на всех процессорах, но обрабатывать только лишь ту часть, которую нужно. Это сэкономит очень много времени мастер-процессору (который, кстати, после анализа времени работы всех процессоров и показывал самое долгое время работы, что логично), экономя время на пересылках всех блоков по процессорам.

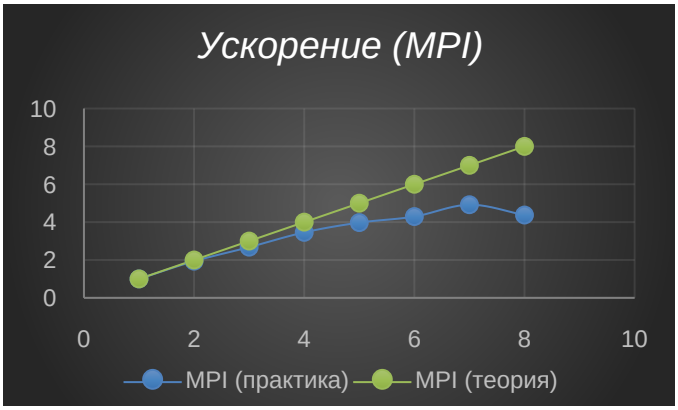
	OMP	MPI
Cons. AVG	6,13873	6,21043
1	6,60915	6,21043
2	3,35351	3,206
3	2,85631	2,32013
4	2,41816	1,79562
5	2,09492	1,56249
6	1,92882	1,44636
7	1,72754	1,26247
8	1,82752	1,4197

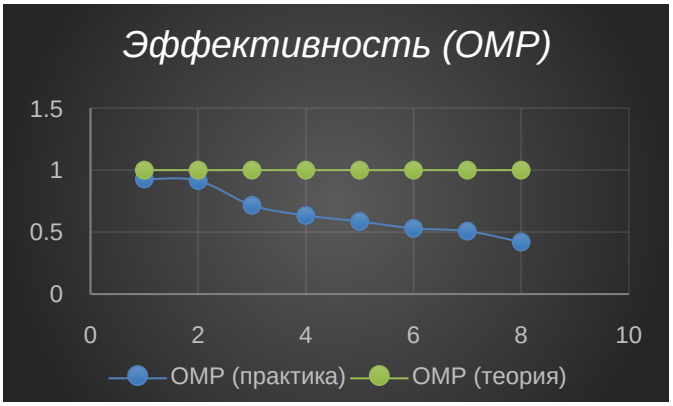












Вот тут уже немного лучше: максимальное ускорение достигло $5{,}12$. Не сказать, что идеально, но оно даже обогнало OMP.

Примечание: вообще, процесс слияния блоков тоже можно как-нибудь попытаться распараллелить, но я лично не знаю, как это сделать, т.к. по сути каждая следующая итерация зависит от другой, да и смысла в этом нет, мы и так получили достаточное ускорение, а выигрыш там будет минимальный, операция все таки линейная.

Выводы

Хотя теоретическое сравнение не до конца корректно (например, в нашем случае последовательная сортировка Шелла сортирует массив, размером в 8 раз меньше исходного, не в 8, а $\sim$ в 12.4 раз быстрее, => зависимость будет не линейная от кол-ва процессоров), нам все же удалось получить достаточно большое ускорение, которое даже было больше, чем в реализации на OMP. Возможно (и даже определенно) мой код не оптимален по эффективности, т.к. сейчас у меня и не стояла задача написать идеальный алгоритм: я просто выбрал самый простой в реализации способ (но как мне кажется, самый эффективный); однако ради чистоты эксперимента можно реализовать и другие варианты распараллеливания и посмотреть, насколько разнятся результаты. Но оставим это на будущее =)