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

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 (я забыл ее отключить во время замеров) -fopenmp -lm (для работы sqrt())

Реализация функции проверки числа на простоту

Существуют разные способы проверки на то, является ли число простым, или нет. Мы будем использовать полный перебор (перебор делителей) с некоторой модификацией. Т.к. полный перебор является очень времязатратной (зато очень прост в реализации), то можно сократить количество проверок засчет того, что сразу проверить, является ли чисто кратным чему-то, а потом не проверять делители, кратные этим числам. К примеру, можно взять проверку на кратность двум и трем, тем самым, сокращая количество проверок в каждых 6 делителях (вида $6\mathrm{k} + \mathrm{i}$, $\mathrm{i} = 0..5$) с 6 до 2 (проверять только числа вида $6k \pm 1$). Получаем вот такую простенькую функцию:

```
int isSimple(long value)
{
    if (!(value % 2))
        return 0;
    if (!(value % 3))
        return 0;
    int limit = floor(sqrt(value));
    int real_i = 1;
    for (int i = 1; real_i <= limit; ++i)
    {
        real_i = i * 6;
        if (!(value % (real_i - 1)))
            return 0;
        if (!(value % (real_i + 1)))
            return 0;
    }
    return 1;
}
```

Примечание: собственно говоря, кроме проверки кратности 2 и 3, можно добавить и проверки на кратность другим простым числам: 5, 7, 11 и тд. Для этого тоже есть специальные алгоритмы, которые учитывает проверки на кратность простым числам (ибо очевидно, что если число составное, то все его наименьший собственный делитель - простое число, там даже простенькая теорема об этом есть). Они называются перебором с запоминанием простых чисел и колесный метод. Но они дают просто ускорение определения, является ли число простым или составным, а нам в данной лабораторной надо не ускорить алгоритм, а проверить совместную работу MPI с OMP, поэтому обойдемся ускорением нашей функции $\sim$ в 3 раза засчет проверки на кратность 2 и 3.

Распараллеливание поиска всех простых чисел

Будем использовать для сохранения результатов буфер такого же размера, как и длина промежутка [*Nstart...Nend*]. Да, это катастрофическая растрата в памяти (дальше будет видно, что чтобы программа отработала минимум 30 секунд нужны большие Nstart и Nend и диапазон между ними), но зато не надо будет париться с синхронизацией OMP-процессов: будем сохранять в i-тую позицию блока-буфера искомый элемент, а т.к. простые числа принадлежат множеству натуральных, то нам нужны будут ячейки, в которых находится не 0 (мы изначально инициализируем общий массив и все буфера через calloc()). Это просто позволит нам сэкономить на времени, проиграв в памяти.

Теперь нам нужно определиться, как разделять Nstart...Nend между собой. Существует маленькая проблема, конкретно для данной задачи (которая потом будет видна на графиках). Если просто разделить последовательность на N блоков (N - кол-во сгенерированных MPI процессов), то процесс с большим номером получит большие по величине числа, а т.к. мы реализовывали полный перебор для проверки числа на простоту, то ее сложность O(n) (n - само число, т.к. нам нужно провести линейное кол-во проверок). Таким образом, даже стандартное разделение OMP в цикле на потоки не спасет от того, что время работы будет возрастать пропорционально номеру процесса. Вопрос только: на сколько? Есть ли смысл нам пытаться сбалансировать эту разницу? (Спойлер: в моем случае с моими данными не было)

Сам код, собственно, достаточно прост: обертка MPI дефолтная: MPI_Init(), распределение блоков по процессам (процесс с рангом 0 создает из buf массив полной длины, а остальные - только для хранения своего блока найденных простых чисел), перебор всех чисел в своем блоке и занесение их в буфер и сборка всех результатов с MPI_Finalize():

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

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

start = omp_get_wtime();

long *buf = NULL;
long wlenght = Nend - Nstart;
const int wstart = Nstart + (rank * wlenght) / procCount;
const int wend = Nstart + ((rank + 1) * wlenght) / procCount;

if (!rank)
{
    printf("MPI Comm Size: %d;\n", procCount);
    buf = calloc(wlenght, sizeof(long));
}
else
{
    buf = calloc(wend - wstart, sizeof(long));
}

long count = 0;
```

```
    long count = 0;

    omp_set_num_threads(omp_get_max_threads());
#pragma omp parallel num_threads(omp_get_max_threads()) reduction(+ \
                                                                  : count)
    {
#pragma omp for schedule(static)
        for (long i = wstart; i < wend; ++i)
            if (isSimple(i))
            {
                //printf("Rank: %d; num_thread: %d; чиселко: %d;\n", rank, omp_
                ++count;
                buf[i - wstart] = i;
            }
    }
```

```
if (!rank)
{
    for (int i = 1; i < procCount; ++i)
    {
        const int mpiStart = (i * wlenght) / procCount;
        const int mpiEnd = (i + 1) * wlenght / procCount;
        MPI_Recv(buf + mpiStart, mpiEnd - mpiStart, MPI_LONG, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
    //int c = 0;
    //for (int k = 0; k < wlenght; ++k)
    //    if (buf[k] != 0)
    //        ++c;
    //printf("%d - summ\n", c);      You, seconds ago • Uncommitted changes
    //for (int i = 0; i < wlenght; ++i)
    //    if (buf[i])
    //        printf("%d; ", buf[i]);
    free(buf);
}
else
{
    MPI_Send(buf, wend - wstart, MPI_LONG, 0, 0, MPI_COMM_WORLD);
    free(buf);
}

ret = MPI_Finalize();
```

ВАЖНО! Некоторые коммутаторы (mpirun в их числе) автоматически привязывают каждый созданный процесс к своему ядру. Поэтому процесс считает, что находится в одноядерной системе, где невозможно создать потоки (по крайней мере, правильно). Поэтому если мы хотим точно знать, что процесс может генерировать и использовать потоки, для mpirun (конкретно для mpirun) необходимо задать параметр --bind-to-(none/socket/core, в нашем случае можно указать none, чтобы полностью избавиться от привязки к ядру). Без этого директива #pragma omp for использовала только один поток (хотя в самой #pragma omp parallel принты могли выводиться и правильное количество раз). Поэтому сам вызов коммутатора должен включать следующие параметры:

```
mpirun --oversubscribe --bind-to none -np 8 build/lab7
```

--oversubscribe нужен для того, чтобы обойти ограничение на количество используемых ядер (у меня это 4)

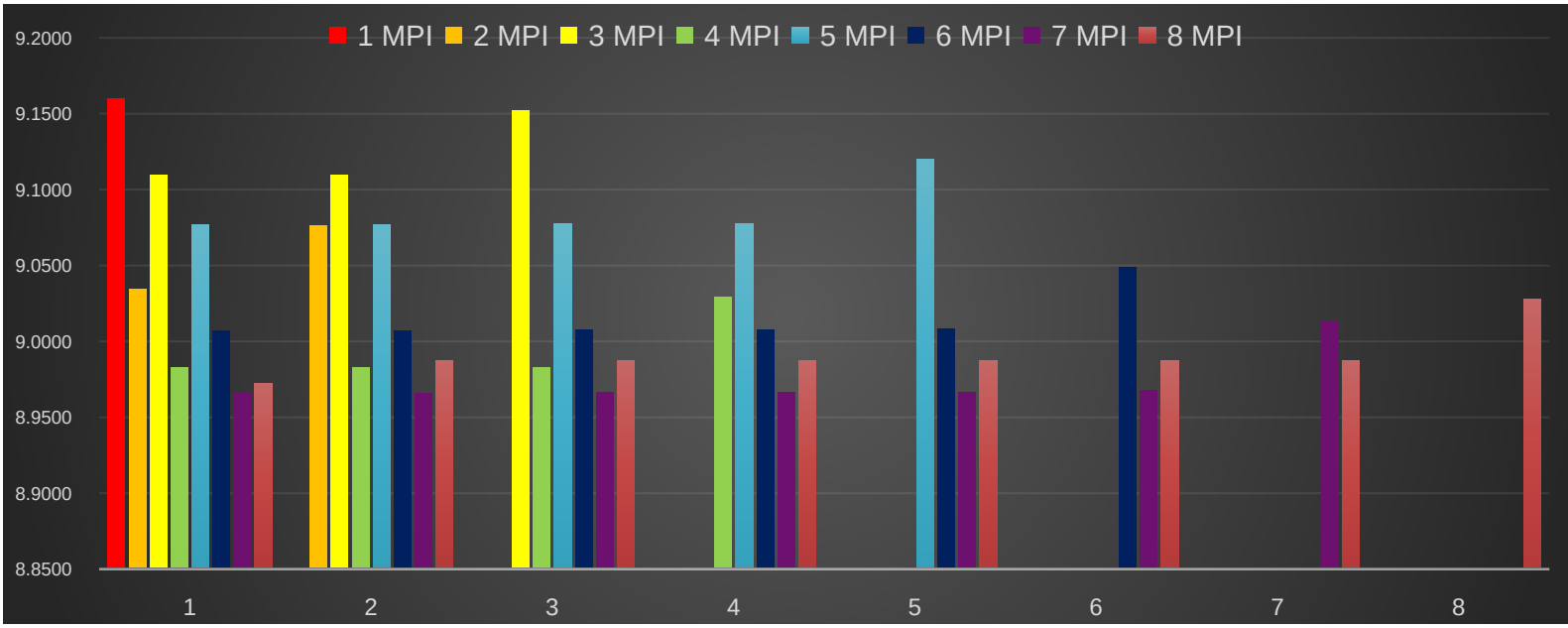
Так что если вдруг ваша программа не хочет использовать OMP, то возможно, что используемая технология MPI или сам коммутатор используют строгую привязку к ядрам процессора.

Практические результаты и таблицы

Для начала я убрал полностью обертку OMP и запустил программу просто через исполняемый файл как ./lab7, чтобы проверить, сколько времени потребуется последовательной реализации (обертку MPI убирать незачем, т.к. без коммутатора она просто игнорируется). Потом запустил 8 раз коммутатор со значениями параметра -np от 1 до 8. Получили вот такую табличку:

count of MPI-procceses	max time			1e7..1e8					
w/o	39,5493								
1	9,1604	9,1604		Prime number count: 5096876 (~ 5%)					
2	9,0763	9,0347	9,0763						
3	9,1525	9,1099	9,1099	9,1525					
4	9,0295	8,9828	8,9828	8,9828	9,0295				
5	9,1202	9,0769	9,0770	9,0773	9,0772	9,1202			
6	9,0493	9,0074	9,0074	9,0075	9,0078	9,0085	9,0493		
7	9,0134	8,9661	8,9662	8,9664	8,9664	8,9668	8,9680	9,0134	
8	9,0276	8,9723	8,9871	8,9872	8,9871	8,9872	8,9871	8,9871	9,0276

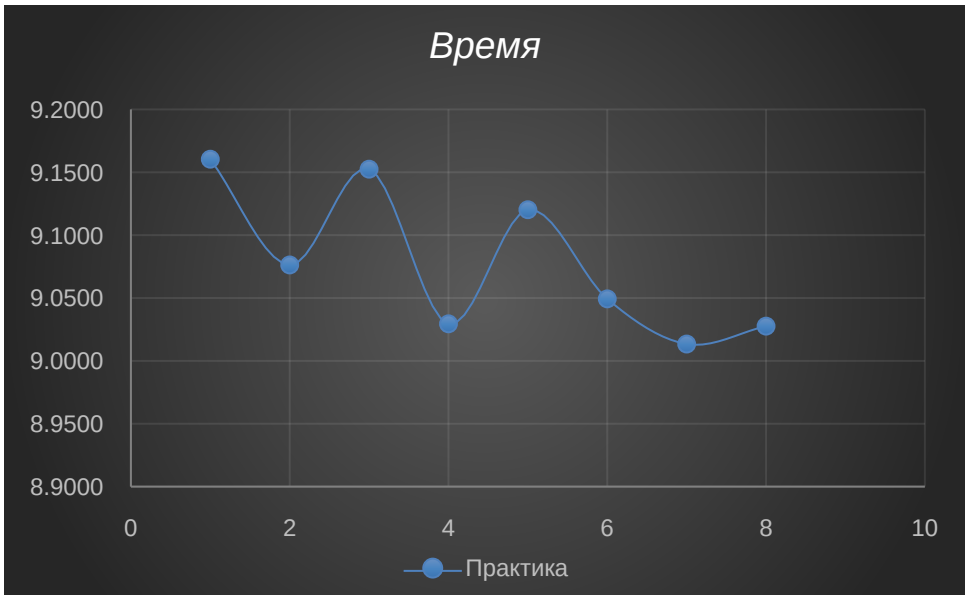
И если посмотреть на график распределения нагрузки:



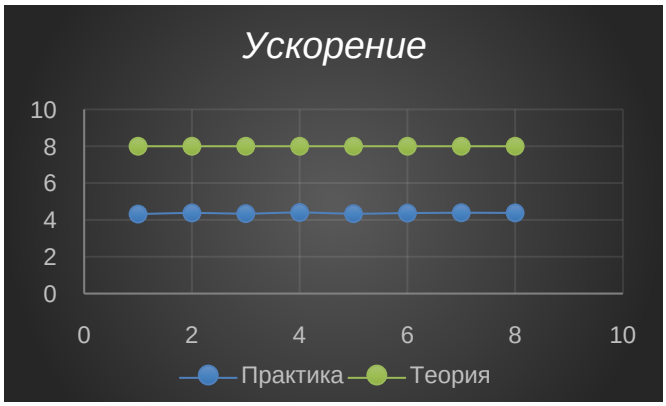
То видно, что каждый последующий процесс при запуске работает дольше предыдущих, но настолько незначительно (разница на пару %), что смысла балансировать нагрузку в данном случае нет.

Примечание: это только в данном случае с данным диапазоном. Можно подобрать такие числа, что этот разброс будет куда больше. Кроме того, запуск проводился с -O3; для запуска без оптимизации разница могла быть и в ~5%.

Кроме того, если взглянуть на время работы программы в зависимости от создаваемых процессов MPI, то можно заметить, что наилучшее время (точнее даже распределение) приходилось на значения, кратные 2 (2, 4, 6, еще 7 и 8), правда, выигрыш все равно очень незначительный - буквально 2% в лучшем случае, даже меньше.



Собственно, ускорение и эффективность отличались от должных теоретических (как O(n/p)) почти ровнов в 2 раза: ускорение в 2 раза меньше, а эффективность в 2 раза ниже (в районе 50% (или 0.5)):



Но справедливости ради стоит отметить, что даже без MPI OMP, которая должна была с этой задачей справиться идеально, давала максимальное ускорение только в 4,56 раза (на 8 потоках), так что в принципе полученное нами ускорение в 4,40 (при -np 4) не далеко от этого значения.

Выводы и общие выводы по курсу

В общем говоря, в данной работе не стояла задача как можно сильнее ускорить программу: нам необходимо было добиться правильной и оптимальной работы связки MPI + OMP. Насчет правильности я могу сказать, что оно работает (спасибо нашему преподавателю по лабораторным, Куприяшину Михаилу Андреевичу, который помог мне разобраться с --bind-to-, благодаря чему OMP правильно заработало), а вот про оптимальность... Я старался избегать лишних пересылок, жертвуя памятью каждого из процессов, и жертвуя так не хило, и не знаю, оправданно ли это. Возможно, стоило действительно создавать один экземпляр массива на одном из процессов, а потом рассылать его всем остальным, ПРИЧЕМ начиная с последнего. Так мы бы могли еще использовать ту несбалансированность в работе процессов: пока главный процесс рассылает блоки остальным, последний (которому требуется большее время на свою подзадачу) уже бы начинал обрабатывать числа. Но, оставим такую реализацию читателю (т.к. по сути кроме как начальных пересылок они ничем не отличается от моей); возможно, данный подход сможет сэкономить память, почти не проиграв в производительности.

Что же касается общих выводов по всем лабораторным... Для чего нам вообще нужно было изучать два разных инструмента распараллеливания - OMP и MPI? Казалось бы, используй себе OMP (с правильными параметрами, по типу того же schedule(static)) и сиди кайфуй. Вот только главная фишка MPI заключается именно в использовании кластерных систем. Предположим, у нас есть десятки, сотни, даже тысячи многоядерных систем. MPI дает нам возможность создать из всех них многоуровневый кластер благодаря технологии процессов. Мы делим какую-то трудоемкую задачу на подзадачи и выполняем ее на нашем кластере, и ... Хоба, полный перебор криптографических ключей, который в обычных условиях занимал бы пару лет, выполняется за один день. А потом, используя то, что, к примеру, каждый из компьютеров работает на 8 ядрах, включаем в каждую из подзадач еще и их распараллеливание по потокам (которое локально тратит очень малое время на объемные задачи), и получаем вместо дня 3 часа. Три ***ЧАСА*** вместо ***ДВУХ ЛЕТ***. Это самый примитивный пример, но в этой последней лабораторной мы смогли убедиться, что такая связка MPI Plus OMP работает, и работает ничем не хуже, чем использование одного из этих инструментов по отдельности. Осталось только собственный суперкомпьютер создать :)