ИНСТИТУТ ИНТЕЛЛЕКТУАЛЬНЫХ КИБЕРНЕТИЧЕСКИХ СИСТЕМ

Кафедра
«Криптология и кибербезопасность»

ОТЧЕТ
ОБ ИССЛЕДОВАТЕЛЬСКОЙ РАБОТЕ ПО ДИСЦИПЛИНЕ
«ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ»

«Реализация алгоритма поиска простых чисел при помощи технологии MPI»

Исполнитель:
студент гр. Б19-505 Аверин В.Д
подпись, дата

Преподаватель: Борзунов Г.И.
подпись, дата

Преподаватель Куприяшин М.А.
по лабораторным работам:
подпись, дата

Москва — 2021

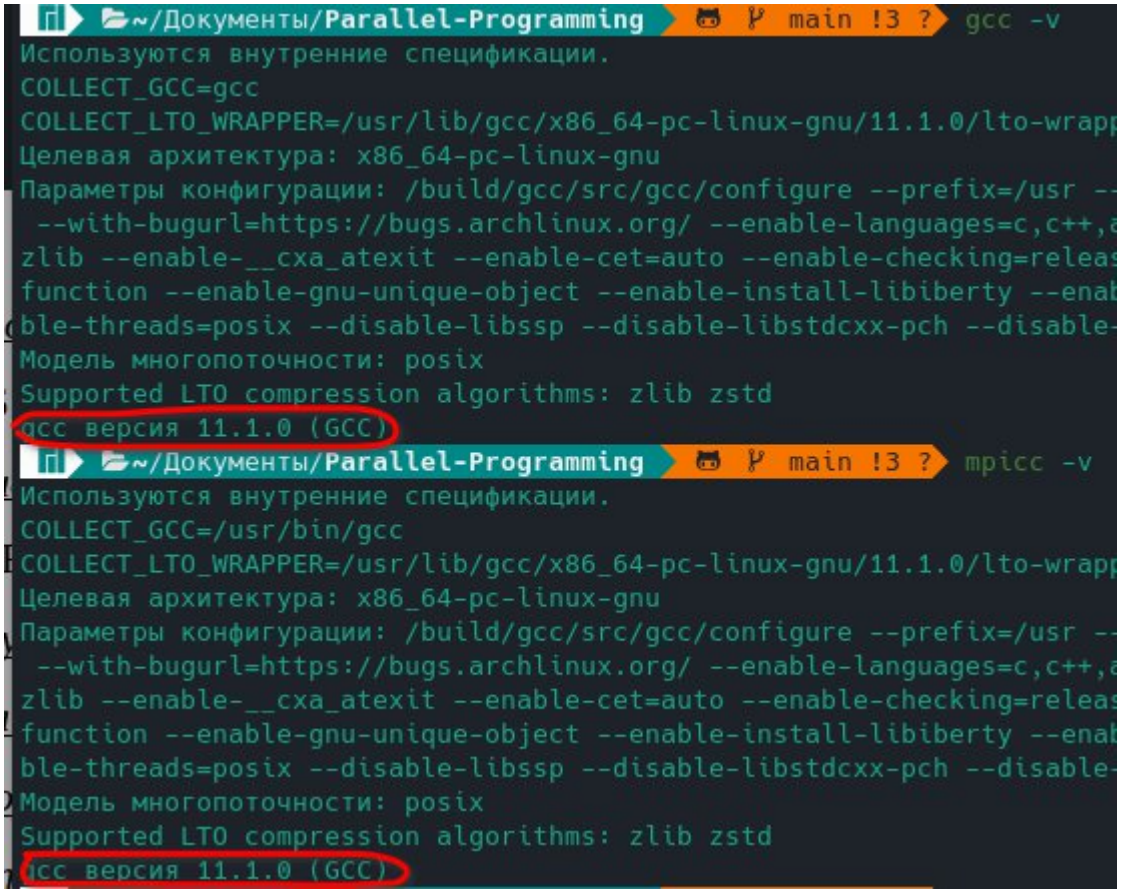
Содержание

Введение

Для поиска проверки числа на простоту придумано достаточно много различных алгоритмов, но наиболее простым является перебор всех возможных делителей данного числа. Если при делении числа N на числа от 2 до N какой-то из результирующих остатков оказывается равен нулю, то N делится на него нацело, => N является составным (можно разложить согласно основной теореме арифметики на простые сомножители). За счет того, что операция проверки каждого числа не зависит ни от чего, кроме как самого числа, то поиск всех простых чисел в каком-либо диапазоне хорошо распараллеливается на независимые процессы. Чем мы и будем пользоваться в данной работе.

В нашей последней лабораторной работе я уже реализовывал программу поиска всех простых чисел в заданном диапазоне, совмещающую работу технологий OMP и MPI (https://github.com/Infernalum/Parallel-Programming), поэтому целью данной работы будет реализация аналога того же алгоритма, но основанного на коде из книги «Параллельные алгоритмы для решения задач защиты информации» Бабенко Л.К., Ищукова Е.А. с. 118..125. Текст данного фрагмента книги приложен к отчету во вложениях. Кроме того, наша задача будет определить время работы, ускорение, и эффективность параллельной реализации. И сравнить полученные результаты с полученными мною в лабораторной #7.

Характеристики лабораторного оборудования



Процессор: 11th Gen Intel Core i7-118G7 3.00Ghz (8CPUs)

RAM: 16 Gb DDR4

Операционная система: OS Linux Manjaro KDE Plasma 5.23.4; версия ядра 5.15.7-1-MANJARO (64-bit)

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

Компилятор: gcc (mpi-надстройщик mpicc) 11.1.0

Редактор кода: Visual Studio Code 1.60.1

Параметры командной строки: -lm (для работы sqrt())

Параметры оптимизации отключены для чистоты эксперимента (т.к. оптимизация не мало зависит от конкретной операционной системы)

Питание: от сети

Предложенная реализация

Предложенная в книге реализация принимает число porog - граница, до которой необходимо найти простые числа. Она создает и записывает в текстовый файл все простые числа в отрезке от 2 до porog включительно. Каждый из процессов получает от главного процесса значение переменной diapazon; и будет перебирать числа от (rank - 1) * diapazon + 2 до rank * diapazon - 1, где rank - ранг процесса. Главный же процессор возьмет на себя последний блок чисел от (pocSize - 1) * diapazon + 2 до (procSize * diapazon) + ost, где procSize - общее количество созданных процессов, а ost - остаток от деления porog на procSize. Таким образом, все процессы будут перебираться одинаковое кол-во чисел, кроме главного, который возьмет на себя и остаточный блок (т.к. вся последовательность от 0 до porog может и не распределиться на на одинаковые целые блоки).

Для начала давайте применим реализацию, предложенную в самой книге (файл prime_example.c; исправления минорны - убраны дерективы #include, связанные со спецификой ОС Windows) и попробуем измерить характеристики, соотвествующие параллельным вычислениям.

Важно сделать следующие замечания:

1) Значение переменной porog (кол-во проверяемых чисел; диапазон), как мы поняли по выполненным лабораторным, должен быть действительно достаточно большим, чтобы коммуникационные затраты мало влияли на саму работу программы. Если принять porog = 6001, как было предложено, то смысл распараллеливания пропадет, т.к. такой малый диапазон сам по себе последовательно перебирается достаточно быстро.

2) Исходя из п.1, можно предположить, что для данной цели хорошо бы подошел инструмент OMP, но об этом ниже;

porog = 6001 тоже будет проверен, но очевидно, что ускорение там будет очень незначительным.

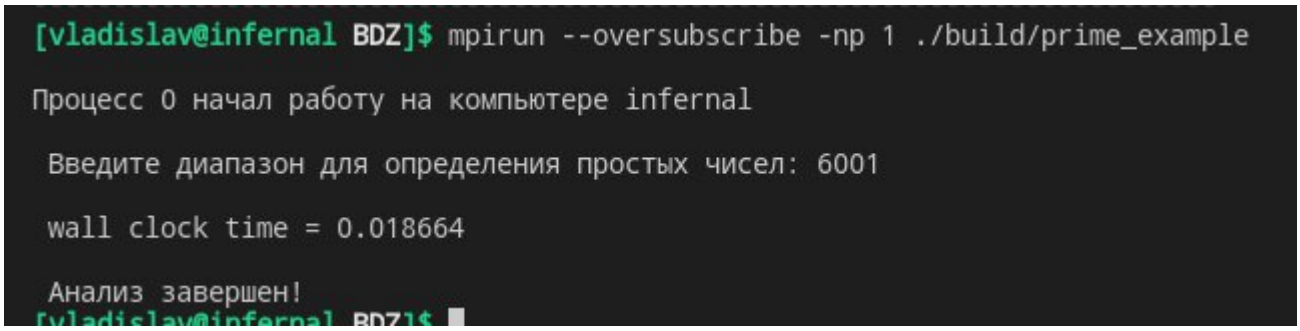
Итак, мы собираем файл prime_exaple.c со следующими параметрами:

mpicc -lm prime_example.c -o prime_example

```
Запуск сборки...
mpicc -lm /home/vladislav/Документы/Parallel-Programming/BDZ/prime_example.c -o /home/vladislav/Документы/Parallel-Programming/BDZ/build/prime_example
```

Далее запускаем собранный объектный файл через коммутатор OpenMPI mpirun:

mpirun --oversubscribe -np 1 prime_example



--oversubscribe используется, чтобы обойти ограничение в создание MPI программой большего кол-ва процессов, чем это предусмотрено по умолчанию (у меня это значение 4).

Практические результаты для предложенной реализации

Таблицы представлены для значения porog = 6001 и porog = 100000 = 1e6

Программа была запущена на значениях параметра -np в пределах 1..8:

```
[vladislav@infernal Parallel-Programming]$ mpirun --oversubscribe -np 1 ./BDZ/build/prime_example
Процесс 0 начал работу на компьютере infernal
 Введите диапазон для определения простых чисел: 6001
 wall clock time = 0.018008
 Анализ завершен!
```

```
[vladislav@infernal Parallel-Programming]$ mpirun --oversubscribe -np 2 ./BDZ/build/prime_example
Процесс 1 начал работу на компьютере infernal
Процесс 0 начал работу на компьютере infernal
 Введите диапазон для определения простых чисел: 6001
 Процесс 1 получил данные от главного процесса, diapazon = 3000
 wall clock time = 0.004138
 Анализ завершен!
```

```
[vladislav@infernal Parallel-Programming]$ mpirun --oversubscribe -np 3 ./BDZ/build/prime_example
Процесс 0 начал работу на компьютере infernal
Процесс 1 начал работу на компьютере infernal
Процесс 2 начал работу на компьютере infernal
 Введите диапазон для определения простых чисел: 6001
 Процесс 1 получил данные от главного процесса, diapazon = 2000
 Процесс 2 получил данные от главного процесса, diapazon = 2000
 wall clock time = 0.004101
 Анализ завершен!
```

```
[vladislav@infernal Parallel-Programming]$ mpirun --oversubscribe -np 4 ./BDZ/build/prime_example
Процесс 0 начал работу на компьютере infernal
Процесс 1 начал работу на компьютере infernal
Процесс 2 начал работу на компьютере infernal
Процесс 3 начал работу на компьютере infernal
 Введите диапазон для определения простых чисел: 6001
 Процесс 2 получил данные от главного процесса, diapazon = 1500
 Процесс 1 получил данные от главного процесса, diapazon = 1500
 Процесс 3 получил данные от главного процесса, diapazon = 1500
 wall clock time = 0.004682
 Анализ завершен!
```

```
[vladislav@infernal Parallel-Programming]$ mpirun --oversubscribe -np 5 ./BDZ/build/prime_example
Процесс 4 начал работу на компьютере infernal
Процесс 1 начал работу на компьютере infernal
Процесс 3 начал работу на компьютере infernal
Процесс 2 начал работу на компьютере infernal
Процесс 0 начал работу на компьютере infernal
 Введите диапазон для определения простых чисел: 6001
 Процесс 2 получил данные от главного процесса, diapazon = 1200
 Процесс 1 получил данные от главного процесса, diapazon = 1200
 Процесс 3 получил данные от главного процесса, diapazon = 1200
 Процесс 4 получил данные от главного процесса, diapazon = 1200
 wall clock time = 0.004295
 Анализ завершен!
```

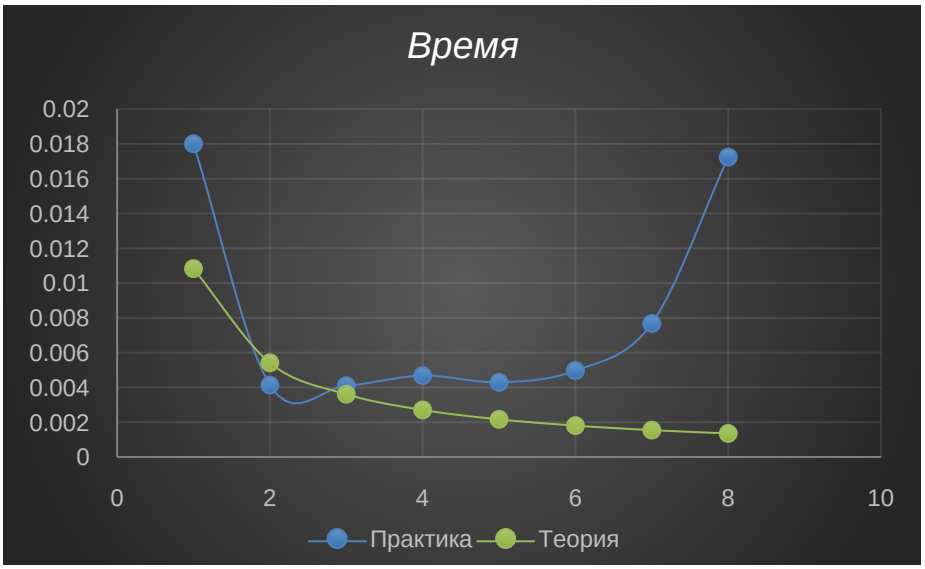
```
[vladislav@infernal Parallel-Programming]$ mpirun --oversubscribe -np 6 ./BDZ/build/prime_example
Процесс 1 начал работу на компьютере infernal
Процесс 3 начал работу на компьютере infernal
Процесс 4 начал работу на компьютере infernal
Процесс 2 начал работу на компьютере infernal
Процесс 5 начал работу на компьютере infernal
Процесс 0 начал работу на компьютере infernal
 Введите диапазон для определения простых чисел: 6001
 Процесс 3 получил данные от главного процесса, diapazon = 1000
 Процесс 5 получил данные от главного процесса, diapazon = 1000
 Процесс 1 получил данные от главного процесса, diapazon = 1000
 Процесс 2 получил данные от главного процесса, diapazon = 1000
 Процесс 4 получил данные от главного процесса, diapazon = 1000
 wall clock time = 0.004971
 Анализ завершен!
```

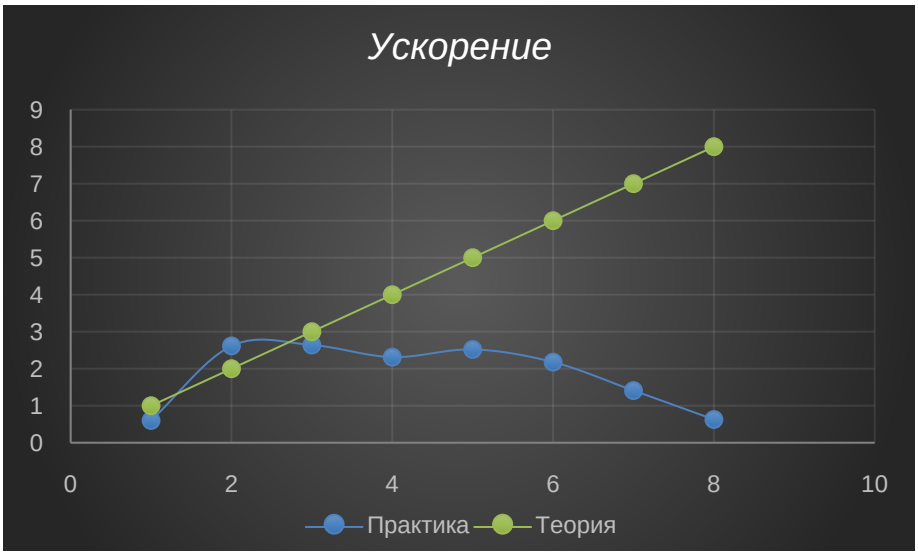
```
[vladislav@infernal Parallel-Programming]$ mpirun --oversubscribe -np 7 ./BDZ/build/prime_example
Процесс 2 начал работу на компьютере infernal
Процесс 6 начал работу на компьютере infernal
Процесс 0 начал работу на компьютере infernal
Процесс 1 начал работу на компьютере infernal
Процесс 3 начал работу на компьютере infernal
Процесс 4 начал работу на компьютере infernal
Процесс 5 начал работу на компьютере infernal
 Введите диапазон для определения простых чисел: 6001
 Процесс 4 получил данные от главного процесса, diapazon = 857
 Процесс 5 получил данные от главного процесса, diapazon = 857
 Процесс 6 получил данные от главного процесса, diapazon = 857
 Процесс 1 получил данные от главного процесса, diapazon = 857
 Процесс 2 получил данные от главного процесса, diapazon = 857
 Процесс 3 получил данные от главного процесса, diapazon = 857
 wall clock time = 0.007681
 Анализ завершен!
```

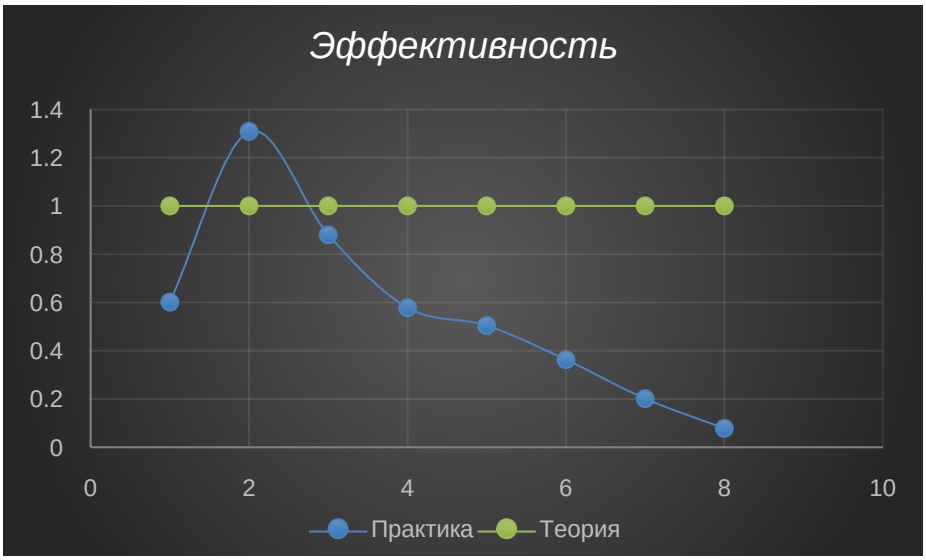
```
[vladislav@infernal Parallel-Programming]$ mpirun --oversubscribe -np 8 ./BDZ/build/prime_example
Процесс 6 начал работу на компьютере infernal
Процесс 7 начал работу на компьютере infernal
Процесс 1 начал работу на компьютере infernal
Процесс 2 начал работу на компьютере infernal
Процесс 0 начал работу на компьютере infernal
Процесс 3 начал работу на компьютере infernal
Процесс 4 начал работу на компьютере infernal
Процесс 5 начал работу на компьютере infernal
 Введите диапазон для определения простых чисел: 6001
 Процесс 2 получил данные от главного процесса, diapazon = 750
 Процесс 3 получил данные от главного процесса, diapazon = 750
 Процесс 4 получил данные от главного процесса, diapazon = 750
 Процесс 5 получил данные от главного процесса, diapazon = 750
 Процесс 7 получил данные от главного процесса, diapazon = 750
 Процесс 1 получил данные от главного процесса, diapazon = 750
 Процесс 6 получил данные от главного процесса, diapazon = 750
 wall clock time = 0.017241
 Анализ завершен!
```

Так как каждый из процессов (кроме главного) обрабатывает [N/p] чисел, => теоретическое ускорение p, а эффективность - p/p = 1.

Построим графики характеристик (для кривой теоретического времени первое время взято как сборка и запуск программы с помощью обычного gcc):







Как мы видим, результаты очень специфические, и никак не согласовываются с теорией. Это можно объяснить слишком малым объемом вычислений: при таком малом значении porog у нас скорее всего вывод сообщений в выходной поток и файл занимает дольше времени, чем сами вычисления.

Но что будет, если значение porog увеличить, и присвоить, например, 1000000 (1e6):

```
Процесс 0 начал работу на компьютере infernal
 Введите диапазон для определения простых чисел: 1000000
 wall clock time = 59.296687
 Анализ завершен!
```

```
[vladislav@infernal Parallel-Programming]$ mpirun --oversubscribe -np 2 ./BDZ/build/prime_example
Процесс 0 начал работу на компьютере infernal
Процесс 1 начал работу на компьютере infernal
 Введите диапазон для определения простых чисел: 1000000
 Процесс 1 получил данные от главного процесса, diapazon = 500000
 wall clock time = 45.535751
 Анализ завершен!
```

```
 Анализ завершен!
[vladislav@infernal Parallel-Programming]$ mpirun --oversubscribe -np 3 ./BDZ/build/prime_example
Процесс 1 начал работу на компьютере infernal
Процесс 2 начал работу на компьютере infernal
Процесс 0 начал работу на компьютере infernal
 Введите диапазон для определения простых чисел: 1000000
 Процесс 1 получил данные от главного процесса, diapazon = 333333
 Процесс 2 получил данные от главного процесса, diapazon = 333333
 wall clock time = 34.621044
 Анализ завершен!
```

```
[vladislav@infernal Parallel-Programming]$ mpirun --oversubscribe -np 4 ./BDZ/build/prime_example
Процесс 0 начал работу на компьютере infernal
Процесс 2 начал работу на компьютере infernal
Процесс 1 начал работу на компьютере infernal
Процесс 3 начал работу на компьютере infernal
 Введите диапазон для определения простых чисел: 1000000
 Процесс 1 получил данные от главного процесса, diapazon = 250000
 Процесс 2 получил данные от главного процесса, diapazon = 250000
 Процесс 3 получил данные от главного процесса, diapazon = 250000
 wall clock time = 28.188326
 Анализ завершен!
```

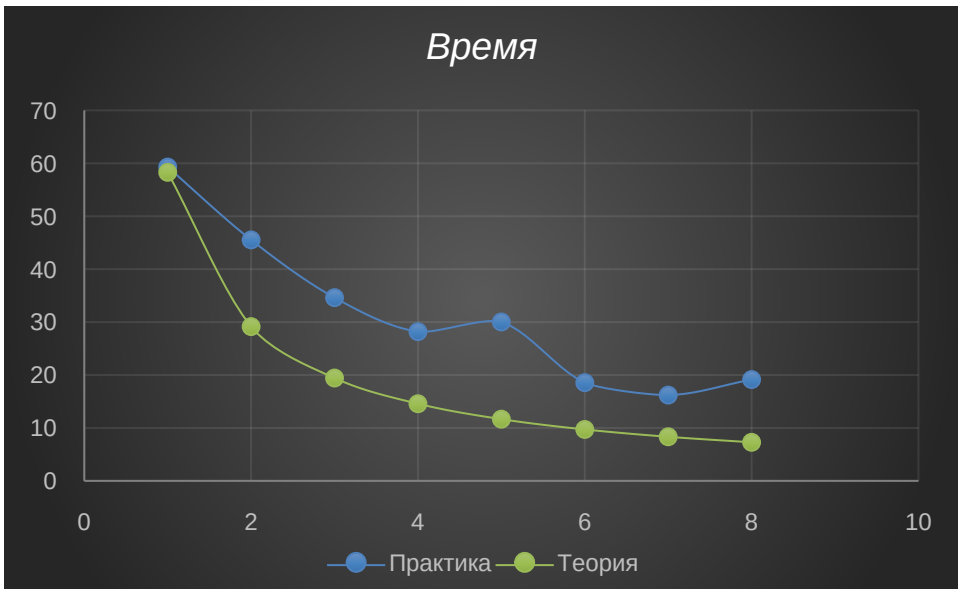
```
[vladislav@infernal Parallel-Programming]$ mpirun --oversubscribe -np 5 ./BDZ/build/prime_example
Процесс 0 начал работу на компьютере infernal
Процесс 2 начал работу на компьютере infernal
Процесс 3 начал работу на компьютере infernal
Процесс 4 начал работу на компьютере infernal
Процесс 1 начал работу на компьютере infernal
 Введите диапазон для определения простых чисел: 1000000
 Процесс 1 получил данные от главного процесса, diapazon = 200000
 Процесс 2 получил данные от главного процесса, diapazon = 200000
 Процесс 3 получил данные от главного процесса, diapazon = 200000
 Процесс 4 получил данные от главного процесса, diapazon = 200000
 wall clock time = 30.023091
 Анализ завершен!
```

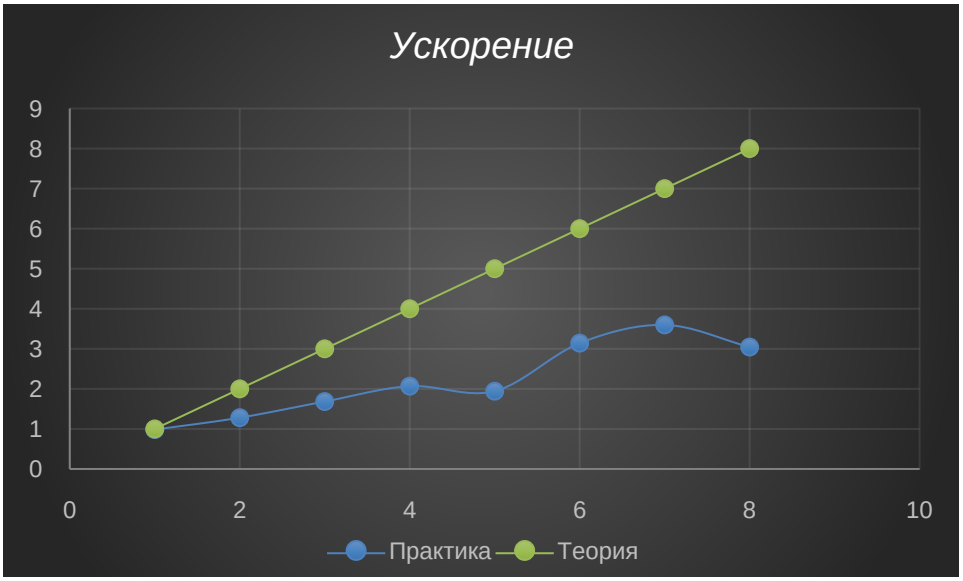
```
[vladislav@infernal Parallel-Programming]$ mpirun --oversubscribe -np 6 ./BDZ/build/prime_example
Процесс 0 начал работу на компьютере infernal
Процесс 5 начал работу на компьютере infernal
Процесс 1 начал работу на компьютере infernal
Процесс 2 начал работу на компьютере infernal
Процесс 3 начал работу на компьютере infernal
Процесс 4 начал работу на компьютере infernal
 Введите диапазон для определения простых чисел: 1000000
 Процесс 3 получил данные от главного процесса, diapazon = 166666
 Процесс 4 получил данные от главного процесса, diapazon = 166666
 Процесс 5 получил данные от главного процесса, diapazon = 166666
 Процесс 1 получил данные от главного процесса, diapazon = 166666
 Процесс 2 получил данные от главного процесса, diapazon = 166666
 wall clock time = 18.555280
 Анализ завершен!
```

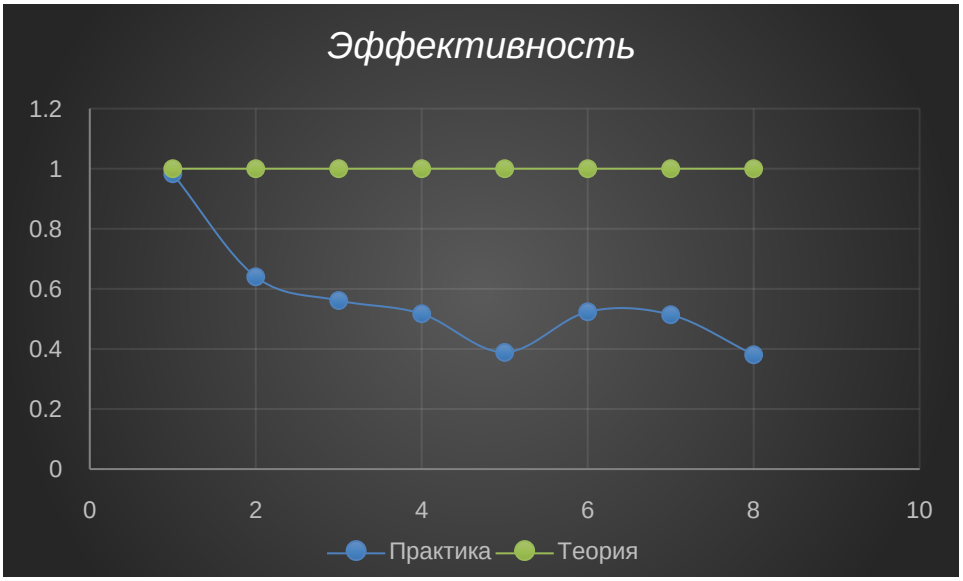
```
[vladislav@infernal Parallel-Programming]$ mpirun --oversubscribe -np 7 ./BDZ/build/prime_example
Процесс 1 начал работу на компьютере infernal
Процесс 3 начал работу на компьютере infernal
Процесс 4 начал работу на компьютере infernal
Процесс 5 начал работу на компьютере infernal
Процесс 6 начал работу на компьютере infernal
Процесс 0 начал работу на компьютере infernal
Процесс 2 начал работу на компьютере infernal
 Введите диапазон для определения простых чисел: 1000000
 Процесс 1 получил данные от главного процесса, diapazon = 142857
 Процесс 2 получил данные от главного процесса, diapazon = 142857
 Процесс 3 получил данные от главного процесса, diapazon = 142857
 Процесс 4 получил данные от главного процесса, diapazon = 142857
 Процесс 5 получил данные от главного процесса, diapazon = 142857
 Процесс 6 получил данные от главного процесса, diapazon = 142857
 wall clock time = 16.205472
 Анализ завершен!
```

```
[vladislav@infernal Parallel-Programming]$ mpirun --oversubscribe -np 8 ./BDZ/build/prime_example
Процесс 0 начал работу на компьютере infernal
Процесс 1 начал работу на компьютере infernal
Процесс 7 начал работу на компьютере infernal
Процесс 5 начал работу на компьютере infernal
Процесс 6 начал работу на компьютере infernal
Процесс 2 начал работу на компьютере infernal
Процесс 3 начал работу на компьютере infernal
Процесс 4 начал работу на компьютере infernal
 Введите диапазон для определения простых чисел: 1000000
 Процесс 1 получил данные от главного процесса, diapazon = 125000

 Процесс 4 получил данные от главного процесса, diapazon = 125000
 Процесс 5 получил данные от главного процесса, diapazon = 125000
 Процесс 6 получил данные от главного процесса, diapazon = 125000
 Процесс 2 получил данные от главного процесса, diapazon = 125000
 Процесс 7 получил данные от главного процесса, diapazon = 125000
 Процесс 3 получил данные от главного процесса, diapazon = 125000
 wall clock time = 19.146287
 Анализ завершен!
```

В таком случае графики характеристик будут вот такими:







При данном значении porog макисмальное ускорение достигло 3,59 (на 7 процессах), а эффективность в среднем равна 0,5.

Но самое интересное не это. На мой взгляд у этого кода предостаточно, я бы даже сказал, катастрофически много недочетов.

♦ Самое важное: здесь реализована т.н. «защита от дураков». Посмотрим на данную строку в книге (самый последней фрагмент):

```
    // Ожидаем пока все процессы вызовут эту функцию
    free(chisla_buf); // Освобождаем динамическую память
}
```

Здесь они высвобождают память от массива int* chisla_buf. Вот только тут и речи не может идти о ее высвобождении, так как это буфер ГЛАВНОГО процесса, а данный вызов free() используется в ОСТАЛЬНЫХ процессах, т.е. у которых rank != 0. Соотвественно, мы пытаемся очистить неинициализированную память, что приведет к ошибке сегментирования. Да, программа выполнит таймирование и запишет все числа в файл, только в конце все равно будет ошибка. Поэтому там необходимо выполнить free(chisla) - очистку правильного массива:

```
    // Ожидаем, пока все процессы вызовут эту функцию
    free(chisla); // Освобождаем динамическую память
}
```

♦ Зачем это?....

```
// Первый раз j = z.
buf = i/j;
if ((buf*j)==i) // Если число поделилось нацело,
// то оно не является простым
```

По мне, так это максимально странная проверка на деление без остатка. Для этого есть оператор ‘%’, который, к тому же, работает вроде бы быстрее, чем две операции (деление нацело и умножение). Ну допустим. Самое главное: а зачем проверять ВСЕ делители?.... Достаточно проверить все числа от 2 до sqrt(N), где N - наше число. Ибо если делитель найдется среди чисел, больших корня, то автоматически второй делить должен быть в диапазоне от 2 до корня. А они НА ПОЛНОМ СЕРЬЕЗЕ проверяли все возможные числа от 2 до N....

♦ Что делает первый процесс? Мало того, что он должен разослать всем процессам значение переменной diapazon, так ему еще и достается самый объемный по количеству чисел блок (diapazon + ost). И это еще не все. Этот блок - самый последний, а значит, и числа в нем самые большие из всей последовательности. А значит, перебираются они дольше. То есть 1 процесс будет работать быстрее всего

(ему достанется блок с самыми алыми числами), а 0 процессор - дольше всего. Причем НАМНОГО дольше. Это максимально несбалансированная нагрузка, отсюда и малое ускорение (конечно, не настолько, чтобы эффективность снизилась в 2 раза, но это тоже играет свою роль);

♦ Мне не нравится структура кода. Серьёзно. Почему все функции в мейне? Для чего инициализировать все переменные в начале кода? Кто будет в них разбираться с самого начала? Почему стоят циклы с послеусловием там, где можно было использовать for? Не говоря уже о таких странностях, как эта:

```
kol++;
start= start+1;
```

Зачем?... Почему?....

Причем, в другом месте (для не главных процессов) декремент используется в обоих случаях:

```
kol++; // и переходим к
start++;
```

Такое ощущение, что код писали просто чтобы написать, и чтобы работало, даже не рефакторив его.

Собственная реализация поиска

Но самая главная проблема того кода: плохой выбор алгоритма проверки числа на простоту.

В последней седьмой лабораторной, целью которой была исследовать работу связки MPI+OMP (https://github.com/Infernalum/Parallel-Programming) точно также необходимо было реализовать поиск простых чисел в заданном диапазоне.

А теперь давайте немного ее изменим так, чтобы формат ввода-вывода (в файл) был абсолютно такой же, как в представленном в книге примере (и изменим мою функцию проверки числа на простоту: чтобы немного ускорить работу - все таки целью лабораторной было не получить максимально быструю работу, а изучить связку, я сразу проверял число на кратность 2 и 3. Это позволяло из 6 последовательных чисел проверять 2: вида 6k ± 1, что ускорит работу в 3 раза). Единственное, в этой реализации Nstart и Nend у меня задаются в самом коде программы, а не через поток ввода (просто чтобы не делать лишней работы по синхронизации; это все равно на работу самого алгоритма влияет мизерно: надо только передать полученные главным процессом значения Nstart и Nend).

Кроме того, для более полной наглядности сбалансированности будем выводить время работы каждого из процессов.

Полный код моей реализации представлен в файле findprime.c во вложениях:

Параметры все те же самые: Nstart = 1; Nend = 1e6; структура параметров сборки/запуска та же самая:

```
[vladislav@infernal Parallel-Programming]$ mpirun --oversubscribe -np 1 ./BDZ/build/findprime

Процесс 0 начал работу на компьютере infernal
Nstart = 1; Nend = 1000000
 wall clock time = 0.211022
```

Напомню, что в предыдущей реализации с тем же параметрами программе требовалось 59,2 секунды.... А все потому, что проверять все числа от 1 до N для каждого чисел не рационально от слова абсолютно. Поэтому я буду использовать свой код, и получу ее характеристики.

Т.к. диапазон в 1e6 чисел она перебирает достаточно быстро, я возьму диапазон, на перебор которого потребуется хотя бы те же 59 секунд: это от 1 до 7e7:

```
Процесс 0 начал работу на компьютере infernal
Nstart = 1; Nend = 70000000
 wall clock time (proc #0) = 61.090804
```

```
[vladislav@infernal Parallel-Programming]$ mpirun --oversubscribe -np 2 ./BDZ/build/findprime

Процесс 0 начал работу на компьютере infernal
Nstart = 1; Nend = 70000000
Процесс 1 начал работу на компьютере infernal

 wall clock time (proc #1) = 39.343187

 wall clock time (proc #0) = 39.650840
```

```
[vladislav@infernal Parallel-Programming]$ mpirun --oversubscribe -np 3 ./BDZ/build/findprime

Процесс 2 начал работу на компьютере infernal

Процесс 0 начал работу на компьютере infernal
Nstart = 1; Nend = 70000000
Процесс 1 начал работу на компьютере infernal

 wall clock time (proc #1) = 23.202854

 wall clock time (proc #2) = 28.869444

 wall clock time (proc #0) = 29.160025
```

```
[vladislav@infernal Parallel-Programming]$ mpirun --oversubscribe -np 4 ./BDZ/build/findprime


Процесс 1 начал работу на компьютере infernal

Процесс 2 начал работу на компьютере infernal
Процесс 3 начал работу на компьютере infernal

Процесс 0 начал работу на компьютере infernal
Nstart = 1; Nend = 70000000
 wall clock time (proc #1) = 15.848813

 wall clock time (proc #2) = 19.916948

 wall clock time (proc #3) = 23.061588

 wall clock time (proc #0) = 23.387373
[vladislav@infernal Parallel-Programming]$
```

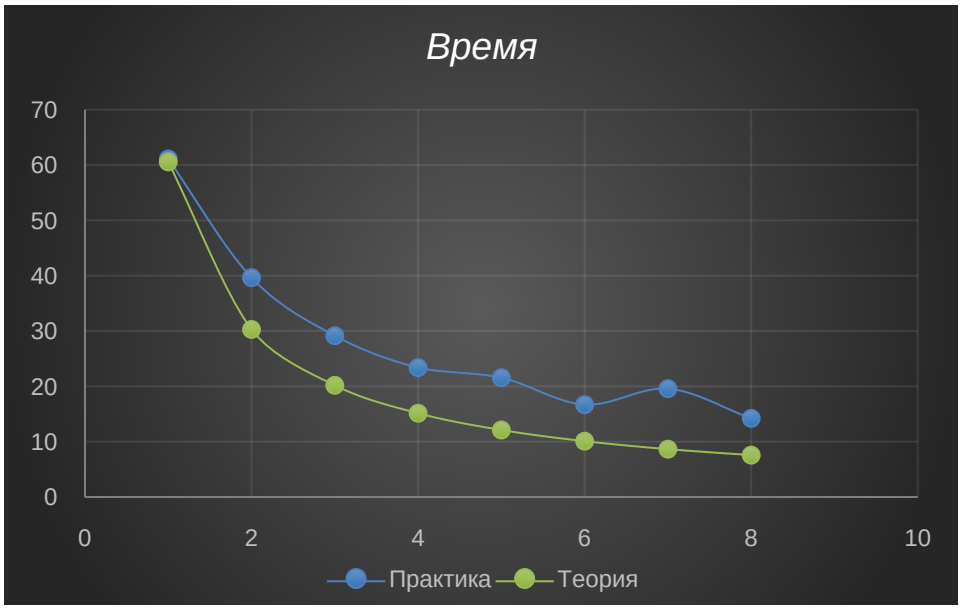
```
[vladislav@infernal Parallel-Programming]$ mpirun --oversubscribe -np 5 ./BDZ/build/findprime

Процесс 1 начал работу на компьютере infernal

Процесс 2 начал работу на компьютере infernal

Процесс 0 начал работу на компьютере infernal
Nstart = 1; Nend = 70000000
Процесс 3 начал работу на компьютере infernal

Процесс 4 начал работу на компьютере infernal

 wall clock time (proc #1) = 19.052691

 wall clock time (proc #2) = 19.539796

 wall clock time (proc #3) = 19.612197

 wall clock time (proc #4) = 21.270579

 wall clock time (proc #0) = 21.573663
```

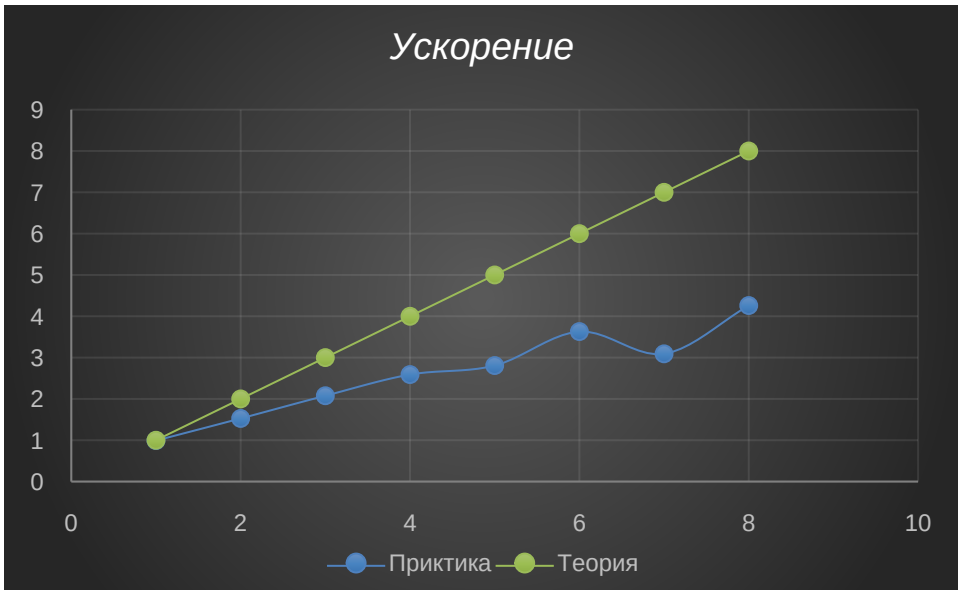
```
[vladislav@infernal Parallel-Programming]$ mpirun --oversubscribe -np 6 ./BDZ/build/findprime

Процесс 1 начал работу на компьютере infernal

Процесс 0 начал работу на компьютере infernal
Nstart = 1; Nend = 70000000
Процесс 2 начал работу на компьютере infernal

Процесс 3 начал работу на компьютере infernal

Процесс 4 начал работу на компьютере infernal

Процесс 5 начал работу на компьютере infernal

 wall clock time (proc #1) = 14.192134

 wall clock time (proc #2) = 14.531143

 wall clock time (proc #3) = 15.554108

 wall clock time (proc #4) = 15.365881

 wall clock time (proc #5) = 16.390342

 wall clock time (proc #0) = 16.685366
```

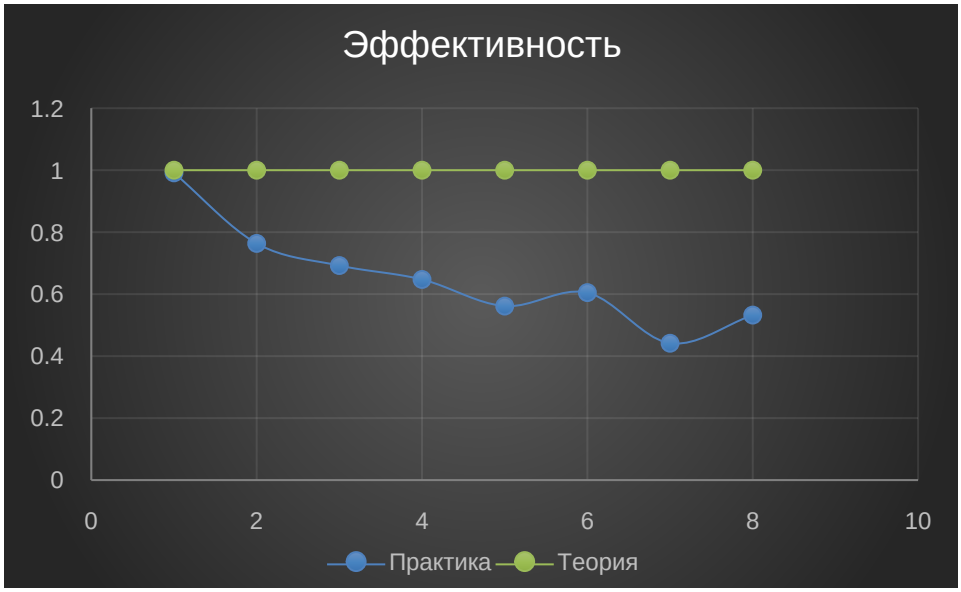
```
[vladislav@infernal Parallel-Programming]$ mpirun --oversubscribe -np 7 ./BDZ/build/findprime

Процесс 4 начал работу на компьютере infernal

Процесс 5 начал работу на компьютере infernal

Процесс 6 начал работу на компьютере infernal

Процесс 0 начал работу на компьютере infernal
Nstart = 1; Nend = 70000000
Процесс 1 начал работу на компьютере infernal

Процесс 2 начал работу на компьютере infernal

Процесс 3 начал работу на компьютере infernal

 wall clock time (proc #1) = 12.443092

 wall clock time (proc #2) = 12.467916

 wall clock time (proc #3) = 15.350417

 wall clock time (proc #4) = 15.374161

 wall clock time (proc #5) = 18.366898

 wall clock time (proc #6) = 19.304839

 wall clock time (proc #0) = 19.591886
```

```
[vladislav@infernal Parallel-Programming]$ mpirun --oversubscribe -np 8 ./BDZ/build/findprime

Процесс 7 начал работу на компьютере infernal

Процесс 0 начал работу на компьютере infernal
Nstart = 1; Nend = 70000000
Процесс 1 начал работу на компьютере infernal

Процесс 2 начал работу на компьютере infernal


Процесс 4 начал работу на компьютере infernal

Процесс 5 начал работу на компьютере infernal

Процесс 6 начал работу на компьютере infernal
Процесс 3 начал работу на компьютере infernal

 wall clock time (proc #1) = 10.297031

 wall clock time (proc #2) = 11.757980

 wall clock time (proc #3) = 11.778149

 wall clock time (proc #4) = 12.342518

 wall clock time (proc #5) = 12.542689

 wall clock time (proc #6) = 12.560297

 wall clock time (proc #7) = 13.934463

 wall clock time (proc #0) = 14.217897
[vladislav@infernal Parallel-Programming]$
```

Выбираем наибольшее время работы процесса в каждом из запусков (которое будет, естественно, все еще у главного процесса):







Если честно, моя реализация мне нравится больше: мне кажется, она понятнее, у меня нет барьерных функций, и работа главного процесса более сбалансирована (что подтверждают результаты 7 лабораторной). Кроме того, в моем случае удалось достичь ускорения большего, чем в предыдущей реализации (максимальное ускорение 4,25 при 8 процессах), а средняя эффективность равна 0.65, что внушительно больше, чем в предыдущей реализации. Но я абсолютно уверен, что моя реализация тоже далека от хотя бы приемлимой.

Выводы

Было проведено исследование двух различных реализаций по сути одного и того же алгоритма поиска простых чисел в определенном диапазоне. Получены характеристики, свойственные параллельным кодам, а так же проведен сравнительный анализ двух реализаций.

Но в данном случае, использование MPI не самое оптимальное решение. Куда более проще использовать потоки OMP, которые смогли бы сами распределить весь цикл перебора чисел друг другу. Однако, как я уже отмечал в выводах лабораторной #7, цель MPI - организация кластерных вычислений. Поэтому использование его на одной локальной машине почти всегда не так эффективно, чем OpenMP.

Дополнительно нужно сказать, что полный перебор методом грубой силы (brute force) - далеко не оптимальный вариант поиска простых чисел. Существую куда полее быстрые алгоритмы. Однако в данной работе стояла цель проверить способность к распараллеливанию самой простой реализации. Используя, например, полученный файл с простыми числами, возможно намного быстрее искать простые числа в диапазоне от 7e7 до 5e15 (приблизительно; как корень из максимального найденного простого числа). А так как простых чисел до 7e7 всего 3840555 (т.е. где-то 5% от общего кол-ва чисел от 1 до 7e7), происходить перебор будет гораздо быстрее.

Список литературы

- 1. Бабенко Л.К., Ищукова Е.А., Сидоров И.Д. Параллельные алгоритмы для решения задач защиты информации. – М.: Горячая линия – Телеком, 2014. – . с. 118 .. 125.
- http://mech.math.msu.su/~shvetz/54/inf/perl-problems/chPrimes_sIdeas.xhtml
- https://ru.wikipedia.org/wiki/Тест_простоты