«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»

Институт компьютерных наук и технологий

Высшая школа искусственного интеллекта

Отчет по лабораторной работе №1

по дисциплине «Параллельное программирование на суперкомпьютерных системах»

Студентка: Гальчич М.И. гр. 3540201/20201 _____

Преподаватель: Лукашин А.А. _____

Санкт-Петербург, 2022

Постановка задачи

В данной лабораторной работе необходимо.

1. Разработать алгоритм нахождения всех простых чисел от 2 до *N*, допускающий распараллеливание на несколько потоков / процессов.
2. Разработать тесты для проверки корректности алгоритма.
3. Реализовать алгоритм с использованием C & Linux pthreads.
4. Реализовать алгоритм с использованием C & OpenMP.
5. Провести исследование эффекта от использования многоядерности / многопоточности на СКЦ, варьируя количество потоков.

Ограничения:

1. *N* – натуральное число в диапазоне [2;1000000000] – верхний предел, до которого будут найдены простые числа.
2. *k* – натуральное число в диапазоне [1; 96] – количество потоков, выполняющих часть алгоритма, которая может быть распараллелена.

Разработанный алгоритм

На входе алгоритма: натуральное число *N* и натуральное число *k*.

На выходе алгоритма: массив *P* всех простых чисел от 2 до *N*.

Шаги алгоритма.

1. Находятся все простые числа в диапазоне [2;$\sqrt{N}$] (корень округляется в меньшую сторону) и записываются в массив *P*.
    a. В массив *P* помещается значение 2.
    b. Рассматриваются все натуральные числа в диапазоне [3;$\sqrt{N}$] с шагом 2 (чётные не рассматриваются, так как они заведомо составные).
    c. Для каждого рассматриваемого числа a проверяется его делимость на все числа из *P*, меньшие либо равные $\sqrt{a}$. Если число a не делится нацело ни на одно из этих чисел, a добавляется в конец массива *P*.

2. Каждый из *k* потоков получает на вход стартовое нечётное число (все эти числа различны) в диапазоне [a_{last}+2; a_{last}+2*k*], где a_{last} – последнее добавленное в *P* значение.

3. Каждый из *k* потоков поочерёдно проверяет все нечётные числа в диапазоне [a_{start}; *N*] с шагом 2*k* (данный шаг гарантирует отсутствие пересечения рассматриваемых чисел с другими потоками) на простоту. Если рассматриваемое число a делится без остатка хотя бы на одно число из *P*, меньшее либо равное $\sqrt{a}$ – поток переходит к следующему числу. Если же рассматриваемое число a не делится без остатка ни на одно число из *P* – значение a заносится в буфер данного потока.

4. После завершения работы всех потоков, их буферы объединяются, суммарный буфер упорядочивается по возрастанию и добавляется в конец массива *P*. Алгоритм завершается.

   Все значения в массиве *P* по окончанию работы алгоритма являются решением задачи.

Тесты

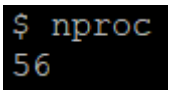
Для проверки корректности работы алгоритма были разработаны тесты на языке C.

Программа должна корректно отработать на всех возможных комбинациях входных параметров из следующего перечня:

- *N*: 2, 105, 239, 543, 1000, 5468, 7777, 10000
- *k*: 1, 2, 12, 24, 36, 48, 56, 96

Среди выборочных значений присутствуют граничные. Среди множества рассматриваемых значений *N* присутствуют простые, чётные, нечётные числа, числа с чётными и нечётными корнями (округлёнными в меньшую сторону). Среди значений *k* присутствует максимальное возможное число потоков на одном узле tornado k-40 (56) на узле cascade (96) без использования гипертредининга.

На скриншоте ниже приведён результат подсчёта числа ядер на узле tornado k-40.



На скриншоте ниже приведён результат подсчёта числа ядер на узле cascade.

Для верификации полученного по завершению работы алгоритма массива простых чисел, используется эталонный перечень всех простых чисел до 10000 (хранится в виде файла “etalon.txt”).

Тест считается пройденным, если при фиксированной комбинации входных параметров *N* и *k* каждое число из упорядоченного массива-результата в точности совпадает со значением из эталонного перечня на той же позиции вплоть до *N*.

Особенности реализации

Разработанную программу можно запустить на выполнение в трёх режимах: тестовом (testing_mode), интерактивном (interactive_mode) и экспериментальном (experiment_mode).

В тестовом режиме алгоритм выполняется 64 раза (по одному на каждую комбинацию *N* и *k* из тестовых входных параметров) и для каждого запуска осуществляется проверка результата выполнения алгоритма по эталонному файлу. Если результат в точности совпал с эталонным, в консоль будет выведено сообщение “Everything is OK!”. В противном случае, в консоль будет выведено: “The result is wrong!”.

В интерактивном режиме алгоритм выполняется один раз для значений *N* и *k*, введённых с консоли. По окончанию работы алгоритма в консоль выведется время его работы, а результат вычислений будет записан в файл “result.txt”.

В экспериментальном режиме алгоритм выполняется последовательно 40 раз для одной из комбинаций входных параметров и в консоль выводится время работы алгоритма на каждой итерации, а в конце среднее время работы алгоритма по 40 итерациям.

Массив простых чисел

Динамический массив, хранящий простые числа unsigned int (результат работы алгоритма) называется primes.

До выделения памяти под него, производится предварительная оценка количества простых чисел в диапазоне [2; *N*]. Это количество равно значению пи-функции от *N*. Гаусс и Лежандр оценили это значение как $\frac{N}{\ln N}$.

Использование на практике данной оценки в первоначальном виде может привести к выделению памяти под меньшее число простых чисел, чем на самом деле есть в диапазоне [2; *N*], поэтому в реализации был добавлен коэффициент 1.5: $\frac{N \times 1.5}{\ln N}$ – для гарантии того, что оценка превзойдет значение пи-функции для любого *N.*

Особенности использования Linux pthreads

Создание потоков

Для создания потока использовалась функция pthread_create. В качестве параметров потоку передается экземпляр структуры threadPack.

Поля структуры threadPack и их описание:

```
int t_id; // номер потока
int t_num; // максимальное количество чисел, обрабатываемых потоком
int h; // шаг просмотра чисел потоком
int primes_size; // размер массива простых чисел (уже найденных)
long N; // верхняя граница, до которой ищутся простые числа
long start; // начальная точка
long * buff; // буфер для результатов
```

Потоки создаются уже после последовательного нахождения простых чисел в диапазоне $[2;\sqrt{N}]$.

В качестве `t_num` каждому потоку передается значение $\frac{m}{k}$, если m кратно k и $\frac{m}{k}+1$ в противном случае, где k – число потоков, а $m=\frac{N-\sqrt{N}}{2k}+1$, если $\sqrt{N}$ чётный, а само N – нет, и $m=\frac{N-\sqrt{N}}{2k}$ в противном случае. Значение `t_num` используется для ограничения размера буфера, куда будут записаны найденные потоком простые числа.

В качестве потоковой функции была создана функция flowFunc, реализующая логику пунктов 2 и 3 описанного ранее алгоритма.

Формирование результата

После завершения работы всех потоков, значения из их буферов добавляются в конец массива простых чисел primes по возрастанию. Так как не все потоки полностью заполнят свои буферы простыми числами, то за последним найденным потоком простым числом в данной реализации ставится 0, до которого и производится считывание. Сборка результата происходит без использования параллелизма.

Особенности использования OpenMP

Распараллеливание алгоритма

Для распараллеливания алгоритма была использована директива:

```
#pragma omp parallel private(buff_curr) shared(primes, grand_buff)
num_threads(k)
```

Массив `grand_buff` хранит значения вычислений простых чисел (тип unsigned int) каждым потоком. К нему имеют доступ все потоки, но только к определенной части.

С помощью функции `omp_get_thread_num()` был получен номер текущего потока (от 0 до $k-1$), где k – число потоков. Каждый поток найдёт не более $amount = \frac{m}{k}$ простых чисел, если m кратно k и $amount = \frac{m}{k} + 1$ простых чисел в противном случае, где k – число потоков, а $m = \frac{N-\sqrt{N}}{2k} + 1$, если $\sqrt{N}$ чётный, а само N – нет , и $m = \frac{N-\sqrt{N}}{2k}$ в противном случае. Таким образом, каждый поток пишет в массив `grand_buff` только по индексам в диапазоне [`omp_get_thread_num() * amount; (omp_get_thread_num() + 1) * amount - 1`], т.е. в диапазон индексов [0; $amount - 1$] будут записаны простые числа, найденные 0-ым потоком, в диапазон индексов [$amount$; $amount * 2 - 1$] – простые числа, найденные 1-ым потоком и так далее.

Индекс, с которого начнётся запись в массив `grand_buff`, является приватной переменной для каждого потока и называется `buff_curr`.

Распараллеливание алгоритма производится уже после последовательного нахождения простых чисел в диапазоне [2;$\sqrt{N}$].

Внутри фигурных скобок после объявленной директивы выполняется код, реализующий логику пунктов 2 и 3 описанного ранее алгоритма.

Формирование результата

После завершения работы всех потоков, значения из массива `grand_buff` добавляются в конец массива простых чисел primes по возрастанию. Так как не все ячейки этого массива заполнены найденными простыми числами, то итоговый результат формируется из значений по числу потоков от начальной ячейки каждого потока до нулевого значения, которое по реализации следует за последним найденным потоком простым числом. Сборка результата происходит без использования паралеллизма.

Исследование эффекта от многопоточности

Для исследования эффекта от многопоточности, программа была запущена 40 раз на каждой из возможных комбинаций входных параметров из следующего перечня:

- *N*: 100000, 1000000, 10000000, 100000000, 1000000000
- *k*: 1, 2, 12, 24, 36, 48, 56, 96

Linux pthreads

В Таблицу 1 занесены средние значения по 40 измерениям времени выполнения алгоритма в секундах для каждой из возможных комбинаций входных параметров на узле tornado-k40 с использованием Linux pthreads. Время было измерено с помощью функции `gettimeofday()` библиотеки `sys/time.h`.

Таблица 1 – время выполнения алгоритма на узле tornado-k40 для Linux pthreads.

$k \backslash N$	100 000	1 000 000	10 000 000	100 000 000	1 000 000 000
1	0.0084	0.1059	1.9897	43.3675	1010.5294
2	0.0056	0.0737	1.0144	22.1249	507.7775
12	0.0039	0.0301	0.3344	6.5331	145.0124
24	0.0045	0.0350	0.2926	4.0962	79.9877
36	0.0060	0.0555	0.4059	3.9830	58.7371
48	0.0074	0.0618	0.5304	4.3412	56.6622
56	0.0083	0.0785	0.4401	4.6035	56.6725
96	0.0096	0.1166	1.0400	6.4618	87.1376

Из значений в Таблице 1 можно сделать вывод, что время выполнения разработанного алгоритма меньше, если он запущен в параллельном режиме (более одного потока), чем если в последовательном. Исключение составляет запуск алгоритма на 96 потоках. У tornado-k40 всего 56 ядер, поэтому попытка выделить под задачу больше потоков, чем фактически есть, приводит к большим накладным расходам, покрывающим выигрыш во времени от использования большого числа потоков.

Однако зависимость времени выполнения от числа потоков нелинейная. Более того, в ходе проведённого эксперимента выяснилось, что не всегда использование большего количества потоков приведёт к меньшему времени выполнения алгоритма. Так, для нахождения простых чисел до 1 000 000 оказалось эффективнее использовать 12 потоков, до 10 000 000 – 24 потока, до 100 000 000 – 36 потоков, до 1 000 000 000 – 48 потоков.

Также было выяснено, что чем больше *N*, тем эффективнее по времени использовать разработанный алгоритм. Так для *N* = 100 000 при использовании 12 потоков можно получить в среднем выигрыш во времени в 54% (0.0045 с), для *N* = 1 000 000 при использовании 12 потоков можно получить в среднем выигрыш во времени в 72% (0.0758 с), для *N* = 10 000 000 при использовании 24 потоков можно получить в среднем выигрыш во времени в 85% (1.6971 с), для *N* = 100 000 000 при использовании 36 потоков можно получить в среднем выигрыш во времени в 80% (39.3845 с), для *N* = 1 000 000 000 при использовании 48 потоков можно получить в среднем выигрыш во времени в 94% (953.8672 с).

Помимо основного эксперимента, было проведено исследование зависимости времени работы параллельной части алгоритма от некоторых комбинаций входных параметров. Результаты исследования отражены в Таблице 2. В ячейках среднее значение в секундах по 40 измерениям.

Таблица 2 – время выполнения параллельной части алгоритма на узле tornado-k40 для Linux pthreads.

$k\backslash N$	100 000	1 000 000	10 000 000
12	0.0022	0.0321	0.4422
24	0.0016	0.0223	0.2665
36	0.0014	0.0205	0.2229
48	0.0017	0.0164	0.1832
56	0.0019	0.0126	0.1475
96	0.0032	0.0134	0.1514

Из значений в Таблице 2 можно сделать вывод, что увеличение числа потоков действительно уменьшает время параллельных вычислений. Но в это время также входит и время на создание и ожидание окончания выполнения всех потоков (накладные расходы). При сравнительно небольшом объёме параллельных работ эти затраты покрывают выигрыш во времени от использования большого числа потоков. Время сборки результата также зависит от числа потоков (так как на каждой итерации находится минимальное значение среди всех буферов потоков), хоть и выполняется последовательно. Всё это объясняет, почему использование большего числа потоков в данной задаче не всегда приводит к уменьшению времени выполнения алгоритма.

OpenMP

В Таблицу 3 занесены средние значения по 40 измерениям времени выполнения алгоритма в секундах для каждой из возможных комбинаций входных параметров на узле tornado-k40 с использованием OpenMP. Время было измерено с помощью функции `gettimeofday()` библиотеки `sys/time.h`.

Таблица 3 – время выполнения алгоритма на узле tornado-k40 для OpenMP.

$k \backslash N$	100 000	1 000 000	10 000 000	100 000 000	1 000 000 000
1	0.0085	0.1022	1.9851	43.0277	1007.4579
2	0.0050	0.0562	0.9996	22.1756	503.6767
12	0.0024	0.0273	0.4782	6.8852	143.0836
24	0.0030	0.0246	0.3598	4.0197	79.0185
36	0.0040	0.0260	0.3634	3.4370	58.0613
48	0.0047	0.0287	0.4045	3.5011	56.9360
56	0.0073	0.0392	0.4026	3.4431	56.4420
96	0.0062	0.0623	0.4500	4.2982	62.1498

Значения в Таблице 3 не сильно отличаются от значений, полученных в Таблице 1. Также, как и при использовании Linux pthreads время выполнения разработанного алгоритма меньше, если он запущен в параллельном режиме (более одного потока), чем если в последовательном.

Для нахождения простых чисел до 100 000 оказалось эффективнее использовать 12 потоков, до 10 000 000 – 24 потока, до 100 000 000 – 36 потоков, до 1 000 000 000 – 56 потоков, что почти в точности совпадает с выводами, полученными при использовании Linux pthreads.

Также, как и при использовании Linux pthreads, оказалось, что чем больше *N*, тем эффективнее по времени использовать разработанный алгоритм. Так для *N* = 100 000 при использовании 12 потоков можно получить в среднем выигрыш во времени в 72% (0.0061 с), для *N* = 1 000 000 при использовании 24 потоков можно получить в среднем выигрыш во времени в 76% (0.0776 с), для *N* = 10 000 000 при использовании 24 потоков можно получить в среднем выигрыш во времени в 82% (1.6253 с), для *N* = 100 000 000 при использовании 36 потоков можно получить в среднем выигрыш во времени в 92% (39.5907 с), для *N* = 1 000 000 000 при использовании 56 потоков можно получить в среднем выигрыш во времени в 94% (951.0159 с).