«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»

Институт компьютерных наук и технологий

Высшая школа искусственного интеллекта

Отчет по лабораторным работам №1-4

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. Реализовать алгоритм с использованием C & MPI.
6. Реализовать алгоритм с использованием Python & MPI.
7. Провести исследование эффекта от использования многоядерности / многопоточности / многопроцессности на СКЦ, варьируя количество потоков/процессов и ядер от 1 до 4 (для MPI).

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] для одного узла и в диапазоне [1; 224] для четырех узлов – количество потоков/процессов, 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

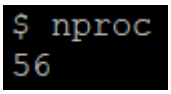
Для проверки корректности работы алгоритма были разработаны тесты на языке C.

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.

Особенности использования C & MPI

Создание процессов

Число процессов, на которых должна быть запущена программа, передаётся в качестве аргумента команде mpirun. Для того, чтобы программа выполнялась параллельно несколькими процессами, необходимо вызвать следующую функцию:

```
MPI_Init(&argc, &argv); // инициализация MPI
```

В результате её выполнения создаётся группа процессов и область их связи.

С помощью функций:

```
MPI_Comm_size(MPI_COMM_WORLD, &k); // общее количество процессов
MPI_Comm_rank(MPI_COMM_WORLD, &pid); // номер текущего процесса
```

каждый процесс получает информацию о своём номере и об общем числе процессов в группе.

Последовательное нахождение простых чисел в диапазоне $[2;\sqrt{N}]$ производится каждым процессом для снижения количества отправляемых сообщений.

Далее каждым процессом выполняется часть кода, реализующая логику пунктов 2 и 3 описанного ранее алгоритма.

Формирование результата

Результат формируется на главном процессе (с идентификатором 0) по следующему алгоритму.

1. Вычисляется число d – максимальная степень двойки, меньшая k – числа потоков.
2. Каждый процесс, идентификатор pid которого больше либо равен d, передаёт сначала размер своего буфера, а затем и его содержимое процессу с идентификатором, равным $pid - d$. После успешной передачи буфера, такой процесс-передатчик завершает свою работу.
3. Процесс-приёмник объединяет свой буфер с полученным буфером, упорядочивая значения в них по возрастанию.
4. Значение d уменьшается в два раза.
5. Шаги 2-4 повторяются до тех пор, пока d не станет меньше 1.

По завершению данного алгоритма на главном процессе окажется буфер, содержащий все найденные процессами простые числа, упорядоченные по возрастанию. Главный процесс добавляет значения из этого буфера в массив

primes, тем самым в массиве primes оказываются все простые числа от 2 до N.

Пример работы алгоритма приведён на Рис. 1. На нём под подразумевается количество процессов, которые ещё не передали никуда свои буферы. Те же процессы, которые передали и завершили работу, отмечены серым цветом.

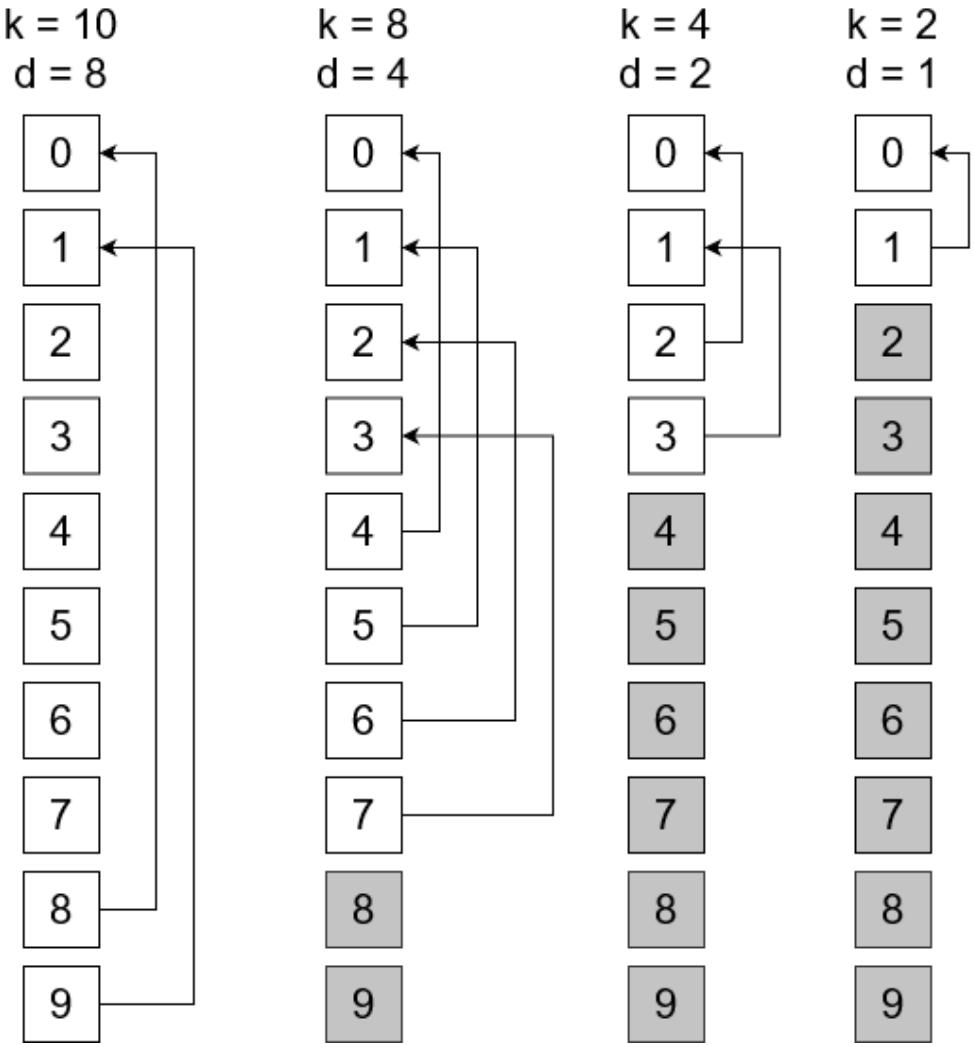


Рис. 1. Пример работы алгоритма сборки результата.

Особенности использования Python & MPI

Создание процессов

Число процессов, на которых должна быть запущена программа, передаётся в качестве аргумента команде mpirun.

С помощью функций:

```
MPI.COMM_WORLD.Get_size() # общее количество процессов
MPI.COMM_WORLD.Get_rank() # номер текущего процесса
```

каждый процесс получает информацию о своём номере и об общем числе процессов в группе.

Последовательное нахождение простых чисел в диапазоне $[2;\sqrt{N}]$ производится каждым процессом для снижения количества отправляемых сообщений.

Далее каждым процессом выполняется часть кода, реализующая логику пунктов 2 и 3 описанного ранее алгоритма.

Вместо режимов запуска, реализованных макросами, используются функции.

Формирование результата

Результат формируется на главном процессе (с идентификатором 0) по следующему алгоритму.

1. Вычисляется число d – максимальная степень двойки, меньшая k – числа потоков.
2. Каждый процесс, идентификатор pid которого больше либо равен d, передаёт содержимое своего буфера процессу с идентификатором, равным $pid - d$. После успешной передачи буфера, такой процесс-передатчик завершает свою работу.
3. Процесс-приёмник объединяет свой буфер с полученным буфером.
4. Значение d уменьшается в два раза.
5. Шаги 2-4 повторяются до тех пор, пока d не станет меньше 1.

По завершению данного алгоритма на главном процессе окажется буфер, содержащий все найденные процессами простые числа. Он упорядочивает их по возрастанию. Главный процесс добавляет значения из этого буфера в массив primes, тем самым в массиве primes оказываются все простые числа от 2 до N.

Исследование эффекта от многопоточности

Для исследования эффекта от многопоточности/многопроцессности, программа была многократно запущена 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).

C & MPI

В Таблицу 4 занесены средние значения по 10 измерениям времени выполнения алгоритма в секундах для каждой из возможных комбинаций входных параметров (кроме $k = 96$, так как на узле tornado-k40 максимум 56 ядер) на одном узле tornado-k40 с использованием C & MPI. Время было измерено с помощью утилиты time.

Таблица 4 – время выполнения алгоритма на одном узле tornado-k40 для C & MPI.

$k \backslash N$	100 000	1 000 000	10 000 000	100 000 000	1 000 000 000
1	0.2820	0.2245	2.080	42.9027	995.6364
2	0.1382	0.1736	1.1318	21.7509	499.7864
12	0.1482	0.1509	0.4509	7.1245	155.8145
24	0.1627	0.1582	0.3118	4.0073	81.6927
36	0.1791	0.1818	0.3718	4.2636	79.0873
48	0.1890	0.2090	0.3490	3.3610	70.1320
56	0.1991	0.2009	0.3136	2.4709	48.3847

Из значений в Таблице 4 можно сделать вывод, что время выполнения разработанного алгоритма меньше, если он запущен в параллельном режиме (более одного процесса), чем если в последовательном.

Для нахождения простых чисел до 100 000 оказалось эффективнее использовать 2 процесса, до 1 000 000 – 12 процессов, до 10 000 000 – 24 процесса, до 1 000 000 000 – 56 процессов.

Также, как и при использовании Linux pthreads и OpenMP, оказалось, что чем больше *N*, тем эффективнее (в основном) по времени использовать разработанный алгоритм. Так для *N* = 100 000 при использовании 2 потоков можно получить в среднем выигрыш во времени в 51% (0.1438 с), для *N* = 1 000 000 при использовании 12 потоков можно получить в среднем выигрыш во времени в 33% (0.0736 с) – выигрыш во времени по абсолютному значению и относительному ниже, чем для *N* = 100 000, для *N* = 10 000 000 при использовании 24 потоков можно получить в среднем выигрыш во времени в 85% (1.7682 с), для *N* = 100 000 000 при использовании 56 потоков можно получить в среднем выигрыш во времени в 94% (40.4318 с), для *N* = 1 000 000 000 при использовании 56 потоков можно получить в среднем выигрыш во времени в 95% (947.2517 с).

Можно сделать вывод, что на одном узле использование технологии C & MPI более эффективно для больших *N*, чем технологий Linux pthreads и OpenMP. Для *N* < 10 000 000 данная технология является менее эффективной.

В Таблицу 5 занесены средние значения по 10 измерениям времени выполнения алгоритма в секундах для каждой из возможных комбинаций входных параметров:

- *N*: 100000, 1000000, 10000000, 100000000, 1000000000
- *k*: 12, 24, 36, 48, 56, 112

на двух узлах tornado-k40 с использованием C & MPI. Время было измерено с помощью утилиты time.

Таблица 5 – время выполнения алгоритма на двух узлах tornado-k40 для C & MPI.

$k\backslash N$	100 000	1 000 000	10 000 000	100 000 000	1 000 000 000
12	0.9109	0.8627	1.1582	7.4827	153.8891
24	0.8627	0.8755	1.0173	5.0464	83.0373
36	0.8818	0.8927	1.0791	4.9964	81.2036
48	0.9018	0.9109	1.0545	4.0518	70.3145
56	0.9100	0.9164	1.0300	3.1282	48.5236
112	1.8470	1.7430	1.8127	3.0260	27.3700

Для *N* = 1 000 000 000 использование двух узлов и 112 потоков позволило снизить среднее время выполнения алгоритма до 27.37 – минимальное полученное в ходе эксперимента время выполнения алгоритма для *N* = 1 000 000 000. В остальных же случаях (при *N* < 1 000 000 000) использование двух узлов привело к увеличению времени выполнения алгоритма.

В Таблицу 6 занесены средние значения по 10 измерениям времени выполнения алгоритма в секундах для каждой из возможных комбинаций входных параметров:

- *N*: 100000, 1000000, 10000000, 100000000, 1000000000
- *k*: 36, 48, 56, 112, 168, 224

на четырех узлах tornado-k40 с использованием C & MPI. Время было измерено с помощью утилиты time.

Таблица 6 – время выполнения алгоритма на четырех узлах tornonado-k40 для C & MPI.

$k \backslash N$	100 000	1 000 000	10 000 000	100 000 000	1 000 000 000
36	0.9710	0.8980	1.0820	4.9620	80.7740
48	0.9010	0.9190	1.0610	4.1080	70.3740
56	0.9290	0.9360	1.0440	3.1280	48.5940
112	1.8510	1.8050	1.8620	3.0310	27.3190
168	2.0600	2.0100	2.0430	3.2180	27.3180
224	2.0110	1.9710	1.9990	2.7350	16.1920

Из значений в Таблице 6 видно, что запускать алгоритм на четырех узлах при малых *N* еще менее эффективно, чем на двух. При *N* = 1 000 000 000 и увеличении числа процессов до 224 (максимум для данного числа узлов), алгоритм в среднем выполняется за 16.192 с, что значительно меньше времени, полученного с помощью запуска программы на двух узлах.

Таким образом, для больших *N* использование технологии C & MPI и выполнение задачи на нескольких узлах дает значительный выигрыш во времени по сравнению с технологиями Linux pthreads и OpenMP.

Python & MPI

В Таблицу 7 занесены средние значения по 10 измерениям времени выполнения алгоритма в секундах для нескольких комбинаций входных параметров на одном узле tornado-k40 с использованием Python & MPI. Время было измерено с помощью утилиты time.

Таблица 7 – время выполнения алгоритма на одном узле tornado-k40 для Python & MPI.

$k\backslash N$	100 000	1 000 000	10 000 000	100 000 000
1	1.8720	5.7080	107.2630	2383.6320
2	1.0740	3.4510	56.0970	1217.2100
12	1.7890	2.5920	19.3000	374.1820
24	2.8670	3.2100	12.2709	198.2200
36	3.8790	4.1960	10.4290	134.1980
48	4.9000	5.2130	10.8410	130.6189
56	5.5960	5.8830	10.7109	118.0260

Нахождение простых чисел до *N* = 1 000 000 000 не было рассмотрено, поскольку уже при *N* = 100 000 000 вычисления заняли достаточно большое время, по сравнению с программами, реализующими технологии C & MPI, Linux pthreads и OpenMP.

Из значений в Таблице 7 видно, что время работы программы, реализующей технологию Python & MPI значительно превышает время работы всех предыдущих реализаций.

Для нахождения простых чисел до 100 000 оказалось эффективнее использовать 2 процесса, до 1 000 000 – 12 процессов, до 10 000 000 – 36 процесса, до 100 000 000 – 56 процессов, что почти в точности совпадает с результатами, полученными для C & MPI.

Также, как и при использовании Linux pthreads и OpenMP, оказалось, что чем больше *N*, тем эффективнее по времени использовать разработанный алгоритм. Так для *N* = 100 000 при использовании 2 потоков можно получить в среднем выигрыш во времени в 43% (0.798 с), для *N* = 1 000 000 при использовании 12 потоков можно получить в среднем выигрыш во времени в 55% (3.116 с), для *N* = 10 000 000 при использовании 36 потоков можно получить в среднем выигрыш во времени в 90% (96.834 с), для *N* = 100 000 000 при использовании 56 потоков можно получить в среднем выигрыш во времени в 95% (2253.0131 с).

В Таблицу 8 занесены средние значения по 10 измерениям времени выполнения алгоритма в секундах для каждой из возможных комбинаций входных параметров:

- *N*: 100000, 1000000, 10000000, 100000000, 1000000000
- *k*: 12, 24, 36, 48, 56, 112

на двух узлах tornado-k40 с использованием Python & MPI. Время было измерено с помощью утилиты time.

Таблица 8 – время выполнения алгоритма на двух узлах tornado-k40 для Python & MPI.

$k\backslash N$	100 000	1 000 000	10 000 000	100 000 000	1 000 000 000
12	2.7840	2.7490	18.0630	368.5230	8244.2320
24	3.0600	3.4390	12.9060	194.9840	4222.5102
36	4.0900	4.4070	10.5050	136.0250	3079.0730
48	5.1610	5.4040	11.0280	130.8590	2923.7110
56	5.7998	6.0410	10.9590	116.7310	2554.5040
112	6.3600	6.0290	8.7070	65.8390	1379.6410

Для *N* = 10 000 000 и *N* = 100 000 000 использование двух узлов и 112 потоков позволило снизить среднее время выполнения алгоритма по сравнению с запусками на одном узле. В остальных же случаях (при *N* < 10 000 000) использование двух узлов привело к увеличению времени выполнения алгоритма. При этом, время выполнения данной программы на двух узлах больше, чем время выполнения программ, реализующих технологии Linux pthreads, OpenMP и C & MPI на одном узле.

В Таблицу 9 занесены средние значения по 10 измерениям времени выполнения алгоритма в секундах для каждой из возможных комбинаций входных параметров:

- *N*: 100000, 1000000, 10000000, 100000000, 1000000000
- *k*: 36, 48, 56, 112, 168, 224

на четырех узлах tornado-k40 с использованием Python & MPI. Время было измерено с помощью утилиты time.

Таблица 9 – время выполнения алгоритма на четырех узлах tornado-k40 для Python & MPI.

$k \backslash N$	100 000	1 000 000	10 000 000	100 000 000	1 000 000 000
36	4.9618	4.3900	10.5573	135.0555	3139.4427
48	5.1645	5.4109	11.0582	129.4800	2941.4527
56	5.7973	6.0582	10.9864	117.7027	2561.1436
112	6.3373	6.0445	8.7118	66.5055	1394.4809
168	6.3782	6.1182	8.6191	64.7791	1351.4182
224	6.4136	6.0736	7.5918	39.0491	726.8564

Из значений в Таблице 9 видно, что запускать алгоритм на четырех узлах при малых *N* еще менее эффективно, чем на двух. При *N* = 10 000 000, *N* = 100 000 000 и *N* = 1 000 000 000 и увеличении числа процессов до 224 (максимум для данного числа узлов), алгоритм выполняется быстрее, нежели на двух узлах. Однако всё равно дольше, чем при использовании других рассмотренных технологий.

Таким образом, для любых *N* и числа узлов/потоков, использование технологии Python & MPI даёт значительное увеличение времени выполнения алгоритма выигрыш по сравнению с технологиями C & MPI, Linux pthreads и OpenMP.