Московский государственный университет
имени М.В. Ломоносова
Факультет вычислительной математики и кибернетики

Практикум по курсу
"Суперкомпьютеры и параллельная обработка данных»

Разработка параллельной версии программ, включающей в себя умножение векторов и сложение матриц

ОТЧЕТ
о выполненном задании

Студента 320 учебной группы факультета ВМК МГУ
Попова Алексея Павловича

Москва, 2019

Оглавление

Постановка задачи

1) Реализовать алгоритм параллельного подсчета ядра *gemver* с использованием *OpenMP*.
2) Исследовать масштабируемость полученной параллельной программы: построить графики зависимости времени исполнения от числа ядер/процессоров для различного объёма входных данных.
3) Для каждого набора входных данных найти количество ядер/процессоров, при котором время выполнения задачи перестаёт уменьшаться.
4) Определить основные причины недостаточной масштабируемости программы при максимальном числе используемых ядер/процессоров.
5) Построить трехмерные графики зависимости времени выполнения программы от размера входных данных и количества процессов.

Описание алгоритма

Простая последовательная версия алгоритма (1)

Начнем описание выполненной работы с определения полученного для улучшения алгоритма. Математически опишем производимые алгоритмом операции.

1) $A = A + u_1 \cdot v_1 + u_2 \cdot v_2$, где A - это матрица размера $n \times n$, а u_1 и u_2 - это столбы высоты n, v_1 и v_2 - это строки длины n.
2) $x = x + \beta \cdot A \cdot y$, где A - это матрица размера $n \times n$, а x и y - это вектор столбцы высоты n, β - вещественный коэффициент.
3) $x = x + z$, где x и z - это вектор столбцы высоты n.
4) $w = w + \alpha \cdot A \cdot x$, где A - это матрица размера $n \times n$, а w и x - это вектор столбцы высоты n, α - вещественный коэффициент.

Приведем основную часть текста программы на языке *c*.

```
int i, j;
for (i = 0; i < n; i++) {
   for (j = 0; j < n; j++) {
      A[i][j] = A[i][j] + u1[i] * v1[j] + u2[i] * v2[j];
   }
}
for (i = 0; i < n; i++) {
   for (j = 0; j < n; j++) {
      x[i] = x[i] + beta * A[j][i] * y[j];
   }
}
for (i = 0; i < n; i++) {
   x[i] = x[i] + z[i];
}
for (i = 0; i < n; i++) {
   for (j = 0; j < n; j++) {
      w[i] = w[i] + alpha * A[i][j] * x[j];
   }
}
```

Улучшенная последовательная версия алгоритма (2)

Очевидно, что описанный в предыдущем пункте алгоритм работает верно и выполняет нужные математические операции, но работа его на аппаратном уровне не может считаться оптимальной. Обратим внимание на то, что массив *A* во втором цикле обходится в неправильном порядке, что в свою очередь не позволяет работать КЭШу. Приведем далее текст программы, в которой исправлен этот недочет.

```
for (int i = 0; i < n; ++i) {
   for (int j = 0; j < n; ++j) {
      A[i][j] = A[i][j] + u1[i] * v1[j] + u2[i] * v2[j];
   }
}
for (int i = 0; i < n; ++i) {
   for (int j = 0; j < n; ++j) {
      x[j] = x[j] + beta * A[i][j] * y[i];
   }
}
for (int i = 0; i < n; ++i) {
   x[i] = x[i] + z[i];
}
for (int i = 0; i < n; ++i) {
   for (int j = 0; j < n; ++j) {
      w[i] = w[i] + alpha * A[i][j] * x[j];
   }
}
```

Обратим внимание, что такой программный код плохо подходит для последующего распараллеливания, так как цикл, в котором был изменен порядок обхода теперь создает зависимость, на каждом шаге все нити в возможной параллельной области будут одновременно работать с массивом *x* и на чтение и на запись.

Итоговая версия алгоритма для распараллеливания (3)

В предыдущем пункте описан нюанс, который не позволяет просто параллельно выполнить представленный код. Для его параллельного выполнения потребуется использовать редукцию, которая значительно замедляет работу. Поэтому было решено исправить этот алгоритм и перейти к блочному выполнению второго цикла.

```
int BLOCK_SIZE = n;
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        A[i][j] = A[i][j] + u1[i] * v1[j] + u2[i] * v2[j];
    }
}
for (int j1 = 0; j1 < n; j1 += BLOCK_SIZE) {
    for (int i = 0; i < n; ++i) {
        for (int j2 = 0; j2 < min(BLOCK_SIZE, n - j1); ++j2) {
            x[j1 + j2] = x[j1 + j2] + beta * A[i][j1 + j2] * y[i];
        }
    }
}
for (int i = 0; i < n; ++i) {
    x[i] = x[i] + z[i];
}
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        w[i] = w[i] + alpha * A[i][j] * x[j];
    }
}
```

Экспериментальным путем было выяснено, что для последовательной программы лучше всего подходит размер блока равный размеру данных, это можно объяснить большим количеством накладных расходов и большим размером КЭШа на машине в которой производилась проверка.

Наиболее важным отличием этого кода является то, что в нем совершенно очевидно можно распределить вычисления в каждом цикле, так как мы четко и ясно избавились от зависимости между нитями, по которым будут разделятся итерации внешнего цикла.

Простая параллельная версия алгоритма (4)

Совершенно очевидно, что алгоритм представленный в предыдущем пункте можно легко переделать в параллельный используя *#pragma omp for*. Но также можно без труда получить корректно работающий параллельный алгоритм из изначально предложенного. Для этого в нужных местах напишем *#pragma omp parallel* и *#pragma omp for*.

```
#pragma omp parallel
{
  #pragma omp for
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
      A[i][j] = A[i][j] + u1[i] * v1[j] + u2[i] * v2[j];
    }
  }
  #pragma omp for
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
      x[i] = x[i] + beta * A[j][i] * y[j];
    }
  }
  #pragma omp for
  for (int i = 0; i < n; ++i) {
    x[i] = x[i] + z[i];
  }
  #pragma omp for
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
      w[i] = w[i] + alpha * A[i][j] * x[j];
    }
  }
}
```

Интересно проверить насколько хорошо работает данный алгоритм на различных машинах. Поэтому далее в работе будут представлены тесты, в которых исследовано время работы алгоритмов с разным количеством потоков.

Итоговая параллельная версия алгоритма (5)

Ранее было написано, что наилучшим алгоритмом для параллельного выполнения является частично блочный алгоритм (Приложение 1), так как он лишен проблемы промаха в КЭШ. Поэтому он был выбран в качестве итогового.

```
if (n < 400) {
   omp_set_num_threads(1);
}
#pragma omp parallel
{
   int BLOCK_SIZE = n / omp_get_num_threads();
   #pragma omp for
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
         A[i][j] = A[i][j] + u1[i] * v1[j] + u2[i] * v2[j];
      }
   }
   #pragma omp for
   for (int j1 = 0; j1 < n; j1 += BLOCK_SIZE) {
      for (int i = 0; i < n; ++i) {
         for (int j2 = 0; j2 < min(BLOCK_SIZE, n - j1); ++j2) {
            x[j1 + j2] = x[j1 + j2] + beta * A[i][j1 + j2] * y[i];
         }
      }
   }
   #pragma omp for
   for (int i = 0; i < n; ++i) {
      x[i] = x[i] + z[i];
   }
   #pragma omp for
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
         w[i] = w[i] + alpha * A[i][j] * x[j];
      }
   }
}
```

В качестве размера блока был выбран n/Th приведенный к целому по правилам языка *c*, где *n* - размер данных, а *Th* - это количество нитей в параллельной области. Данный выбор можно легко обосновать эмпирически. Также для маленьких размеров данных нерационально создавать множество нитей, поэтому вычисление для маленьких наборов данных происходит последовательно.

Тестирование программы

В этом разделе представлены результаты тестирования, описанных в предыдущем разделе, программ и трехмерные графики зависимостей. Тестирование производилось на компьютере *Bluegene*, *Polus* и домашнем компьютере с *2.3 GHz Intel Core i5 7360U*.

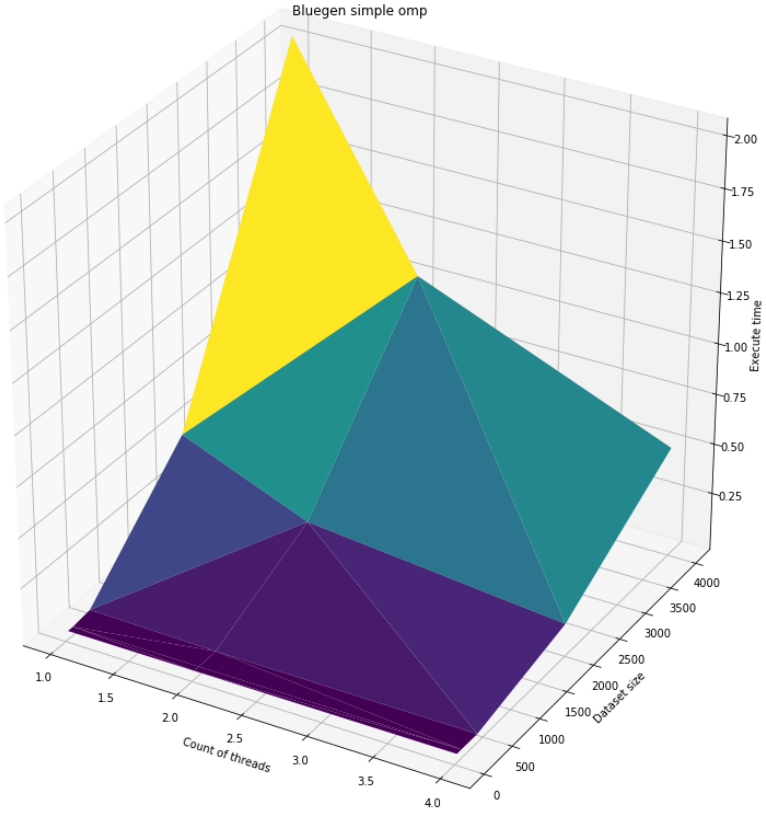
Конфигурации запущенных программ:

- *Bluegene* - 1, 2, 4 потока (с оптимизаторами *-O0, -O2, -O3, -O4*).
- *Polus* - 1, 2, 4, 8, 16, 32, 64, 128 потока (с оптимизаторами *-O0, -O2, -O3, -O4*).
- *2.3 GHz Intel Core i5 7360U* - 1, 2, 4 потока (с оптимизаторами *-O0, -O1, -O2, -O3*).

Важно заметить, что на *Intel Core i5 7360U* всего лишь два физических ядра, на каждом из которых можно запустить лишь по одной нити, поэтому при создании четырех нитей, две из них создаются в виртуальных ядрах.

Тестирование программ на *Bluegene*

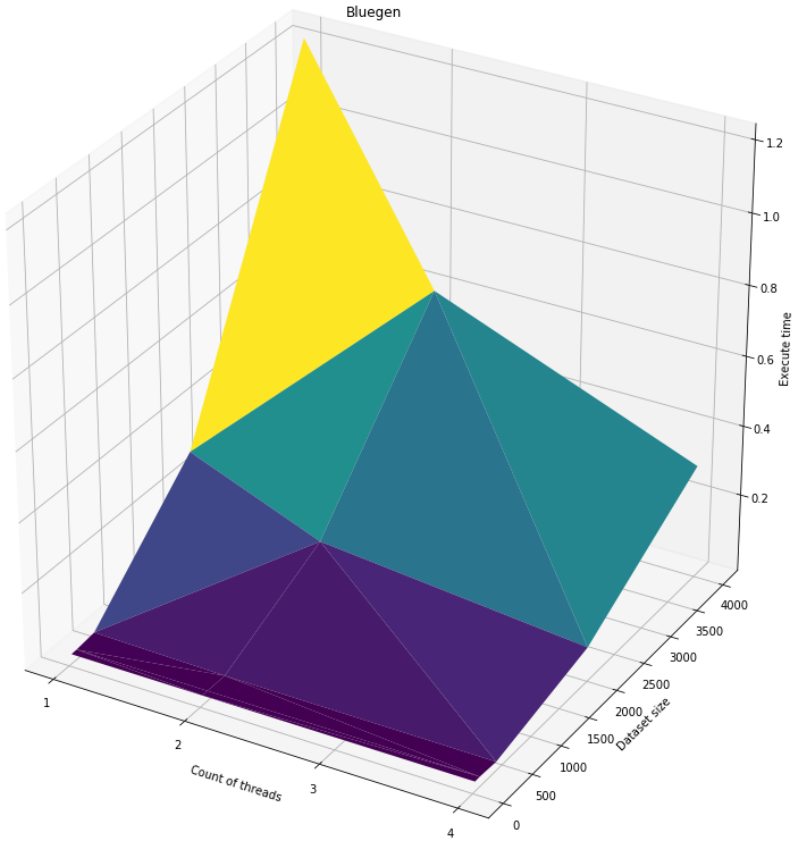
Отметим, что производилось тестирование нескольких версий программы, так как стояла необходимость показать превосходство итоговой версии программы. Также считаю важным обозначить количество тестов произведенных на каждой машине. Для получения наиболее точной картины было произведено по 20 запусков программы с каждой конфигурацией при помощи скрипта на языке *Python* (Приложение 2) и все полученные данные для каждой конфигурации были усреднены. Также изначально производилось тестирование с различными степенями оптимизации программы, но было замечено, что более мощные оптимизации не дают прироста производительности на итоговой версии программ. Важно отметить что тестирование все равно производилось с различными оптимизациями, но данные не были включены в итоговые таблицы.



Count of threads	Dataset size	Execute time
1.0000	40.0000	0.0068
1.0000	120.0000	0.0068
1.0000	400.0000	0.0197
1.0000	2000.0000	0.5084
1.0000	4000.0000	2.0395
2.0000	40.0000	0.0068
2.0000	120.0000	0.0068
2.0000	400.0000	0.0100
2.0000	2000.0000	0.2535
2.0000	4000.0000	1.0207
4.0000	40.0000	0.0068
4.0000	120.0000	0.0068
4.0000	400.0000	0.0061
4.0000	2000.0000	0.1272
4.0000	4000.0000	0.5104

Иллюстрация 1 (Простая параллельная версия программы)

На иллюстрации 1 представлен график выполнения простой параллельной версии программы из пункта (4). Также представлена таблица с данными по которым производилось построение графика. Построение графиков осуществлялось при помощи языка *Python* (Приложение 3).



Count of threads	Dataset size	Execute time
1.0000	40.0000	0.0057
1.0000	120.0000	0.0065
1.0000	400.0000	0.0131
1.0000	2000.0000	0.3022
1.0000	4000.0000	1.2182
2.0000	40.0000	0.0057
2.0000	120.0000	0.0065
2.0000	400.0000	0.0061
2.0000	2000.0000	0.1521
2.0000	4000.0000	0.5985
4.0000	40.0000	0.0057
4.0000	120.0000	0.0065
4.0000	400.0000	0.0030
4.0000	2000.0000	0.0762
4.0000	4000.0000	0.3019

Иллюстрация 2 (Итоговая параллельная версия программы)

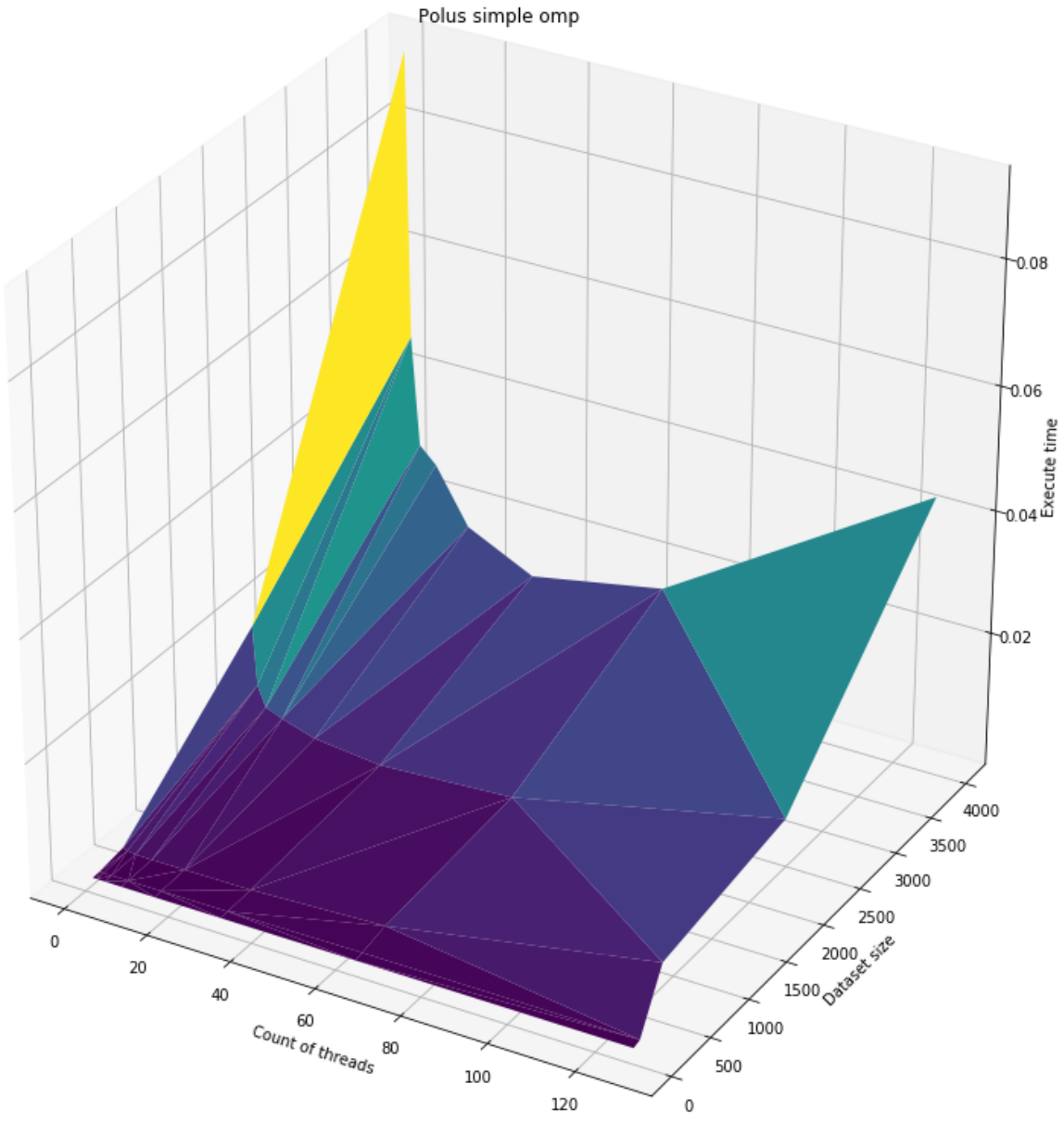
Аналогично предыдущей иллюстрации на иллюстрации 2 приведены график и таблица составленная по результатам замеров времени работы программы на компьютере *Bluegene*.

Итого в результаты тестирования на *Bluegene* было произведено 3200 запусков программ, по 1600 запусков на каждую версию. Из приведенных тестов видно, что время выполнения каждой из версий линейно уменьшается с увеличением числа потоков, таким образом оптимальное число потоков для этой системы и программы с использованием *OpenMP* равно четырем.

Обратим внимание на различие во времени выполнения программы из пункта 4 и из пункта 5. Эти различия хорошо заметны на иллюстрациях 1 и 2. Выигрыш во времени составил практически в два раза, что для программы с таким коротким временем выполнения более чем существенно. Эти выводы подтверждают, что блочный алгоритм для этой задачи подходит наилучшем образом.

Тестирование программ на *Polus*

В данном пункте верны все тезисы предыдущего пункта, поэтому сразу перейдем к приведению результатов тестирования. Для постановки задач на тестирование использовался скрипт на языке Python (Приложение 2) с незначительными изменениями.



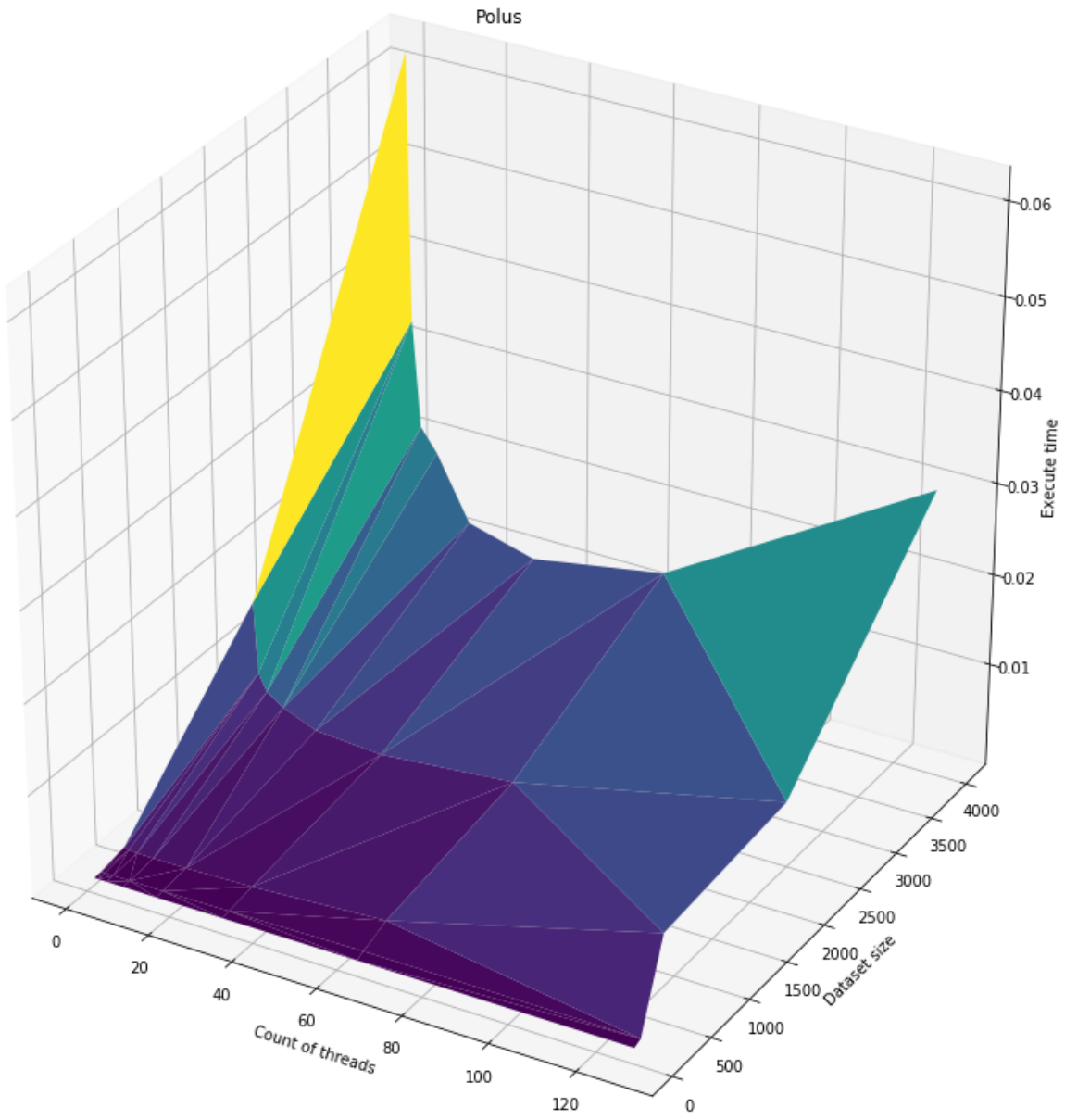
Изображение 1 (Простая параллельная версия программы)

На изображении 1 представлен график зависимости времени выполнения программы из пункта (4) от входных данных. График построен по данным представленным в таблице 1.

Count of threads	Dataset size	Execute time	Count of threads	Dataset size	Execute time
1.0000	40.0000	0.0000	**16.0000**	40.0000	0.0000
1.0000	120.0000	0.0001	**16.0000**	120.0000	0.0001
1.0000	400.0000	0.0007	**16.0000**	400.0000	0.0002
1.0000	2000.0000	0.0193	**16.0000**	2000.0000	0.0036
1.0000	4000.0000	0.0924	**16.0000**	4000.0000	0.0174
2.0000	40.0000	0.0000	**32.0000**	40.0000	0.0000
2.0000	120.0000	0.0001	**32.0000**	120.0000	0.0002
2.0000	400.0000	0.0004	**32.0000**	400.0000	0.0004
2.0000	2000.0000	0.0097	**32.0000**	2000.0000	0.0024
2.0000	4000.0000	0.0466	**32.0000**	4000.0000	0.0121
4.0000	40.0000	0.0000	**64.0000**	40.0000	0.0000
4.0000	120.0000	0.0001	**64.0000**	120.0000	0.0003
4.0000	400.0000	0.0002	**64.0000**	400.0000	0.0014
4.0000	2000.0000	0.0064	**64.0000**	2000.0000	0.0037
4.0000	4000.0000	0.0289	**64.0000**	4000.0000	0.0162
8.0000	40.0000	0.0000	**128.0000**	40.0000	0.0001
8.0000	120.0000	0.0001	**128.0000**	120.0000	0.0004
8.0000	400.0000	0.0002	**128.0000**	400.0000	0.0094
8.0000	2000.0000	0.0053	**128.0000**	2000.0000	0.0135
8.0000	4000.0000	0.0263	**128.0000**	4000.0000	0.0435

Таблица 1 (Простая параллельная версия программы)

Время округлено до 4 знаков после запятой, поэтому некоторые замеры имеют нулевые показатели времени, это сделано только в таблице для экономии места, в Приложении 4 можно увидеть точные значения времени, с округлением до 12 знаков после запятой. Из графика можно заметить, что наилучшие результаты программа показывает при выполнении на 32 потоках. Это можно объяснить тем, что создание большего числа потоков для программы с таким небольшим временем выполнения создает слишком большие накладные расходы, что в свою очередь значительно увеличивает время выполнения и уменьшает выигрыш от параллельного выполнения программы. Но это результаты простой параллельной программы, в итоговой версии получилось еще улучшить результаты, хотя количество потоков при которых время выполнения минимально - осталось неизменным и причины этого описаны выше. Также можно обратить внимание на то, что при меньшем размере входных данных наименьшее время выполнения достигается на меньшем числе потоков, что в свою очередь подтверждает описанные выше причины недостаточной масштабируемости программы.



Изображение 2 (Итоговая параллельная версия программы)

По графику на изображении 2 и данным из таблицы 2 можно заметить, что итоговая версия программы работает в 1.5 раза быстрее чем простая версия. Из этого наблюдения вытекает важный вывод о том, что блочный алгоритм действительно помогает ускорить выполнение параллельной программы внезависимости от системы на которой производится тестирование.

Count of threads	Dataset size	Execute time	Count of threads	Dataset size	Execute time
1.0000	40.0000	0.0000	**16.0000**	40.0000	0.0000
1.0000	120.0000	0.0001	**16.0000**	120.0000	0.0001
1.0000	400.0000	0.0006	**16.0000**	400.0000	0.0004
1.0000	2000.0000	0.0157	**16.0000**	2000.0000	0.0034
1.0000	4000.0000	0.0623	**16.0000**	4000.0000	0.0122
2.0000	40.0000	0.0000	**32.0000**	40.0000	0.0000
2.0000	120.0000	0.0001	**32.0000**	120.0000	0.0002
2.0000	400.0000	0.0004	**32.0000**	400.0000	0.0006
2.0000	2000.0000	0.0079	**32.0000**	2000.0000	0.0028
2.0000	4000.0000	0.0332	**32.0000**	4000.0000	0.0102
4.0000	40.0000	0.0000	**64.0000**	40.0000	0.0001
4.0000	120.0000	0.0001	**64.0000**	120.0000	0.0004
4.0000	400.0000	0.0003	**64.0000**	400.0000	0.0015
4.0000	2000.0000	0.0061	**64.0000**	2000.0000	0.0042
4.0000	4000.0000	0.0216	**64.0000**	4000.0000	0.0127
8.0000	40.0000	0.0000	**128.0000**	40.0000	0.0001
8.0000	120.0000	0.0001	**128.0000**	120.0000	0.0004
8.0000	400.0000	0.0003	**128.0000**	400.0000	0.0095
8.0000	2000.0000	0.0048	**128.0000**	2000.0000	0.0110
8.0000	4000.0000	0.0191	**128.0000**	4000.0000	0.0301

Таблица 2 (Итоговая параллельная версия программы)

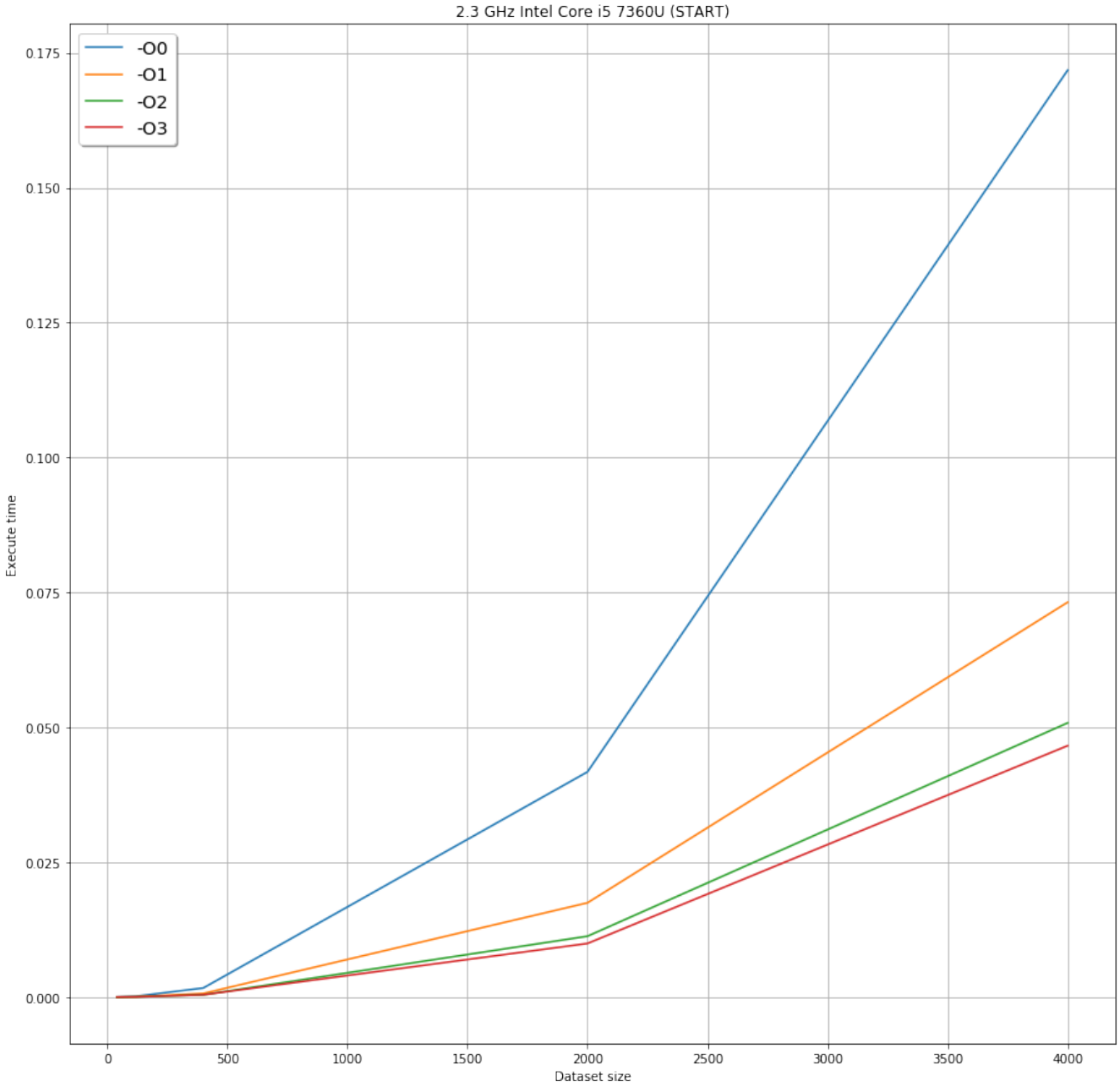
Всего в процессе тестировании на *Polus* программа была запущена 6400 раз. В это число входит запуск программы с различными оптимизаторами (к сожалению, время работы программы с оптимизациями не отличается от времени работы программы без них) и на различных входных данных.

Тестирование программ на *2.3 GHz Intel Core i5 7360U*

В этом разделе проведем сравнение различных оптимизаторов программы, также сравним время выполнения последовательных программ с различными оптимизациями. Данные измерения необходимо было произвести, чтобы оценить качество собственной оптимизации программы.

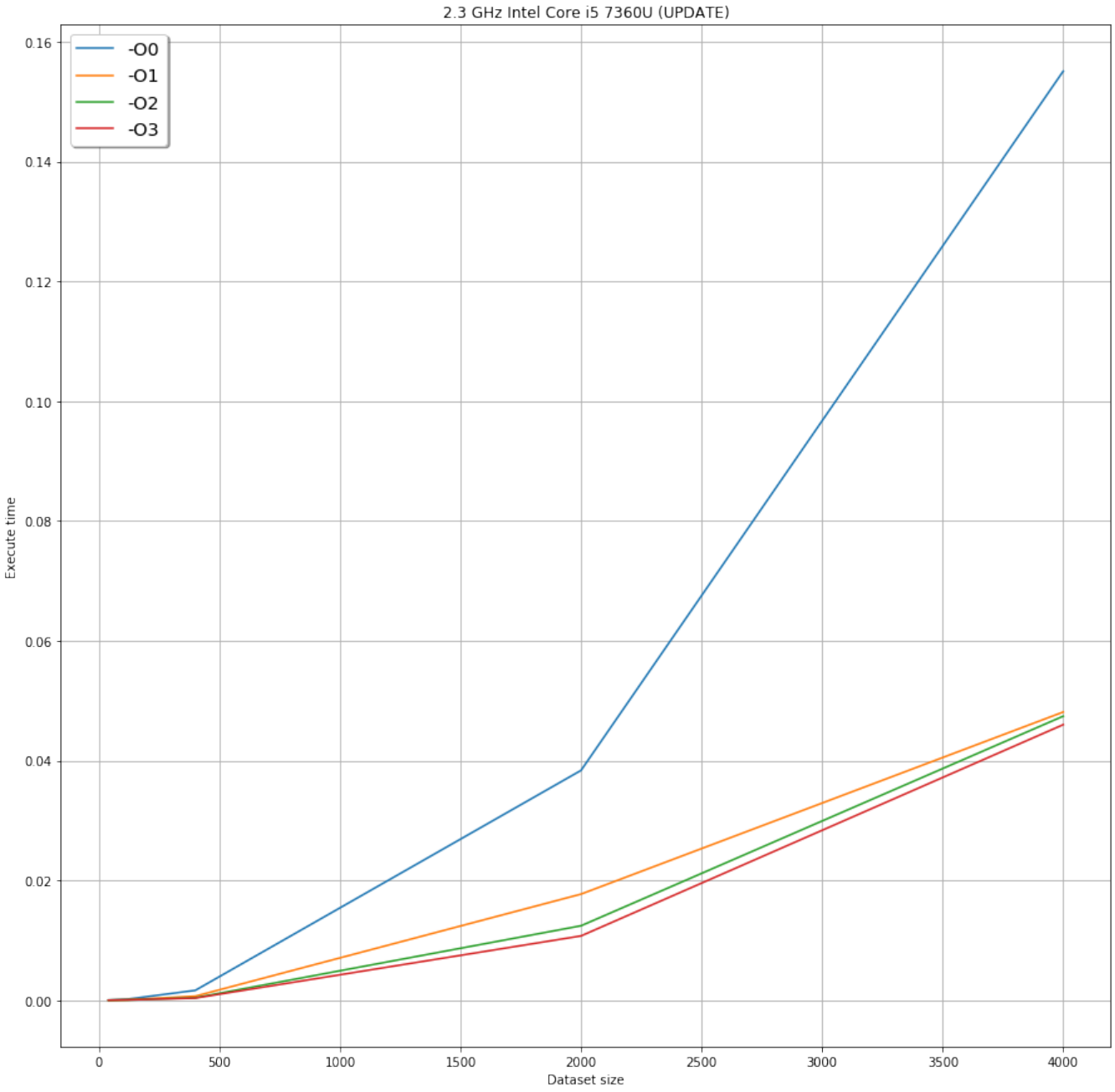
Схема тестирования в этом разделе не отличается от представленной в предыдущих разделах.

В первую очередь оценим качество оптимизации последовательной программы. Для сравнения рассмотрим результаты работы программ из пунктов (1) и (3), так как версия программ из пункта (3) стала базовой для распараллеливания.



Изображение 3 (Базовая последовательная версия программы)

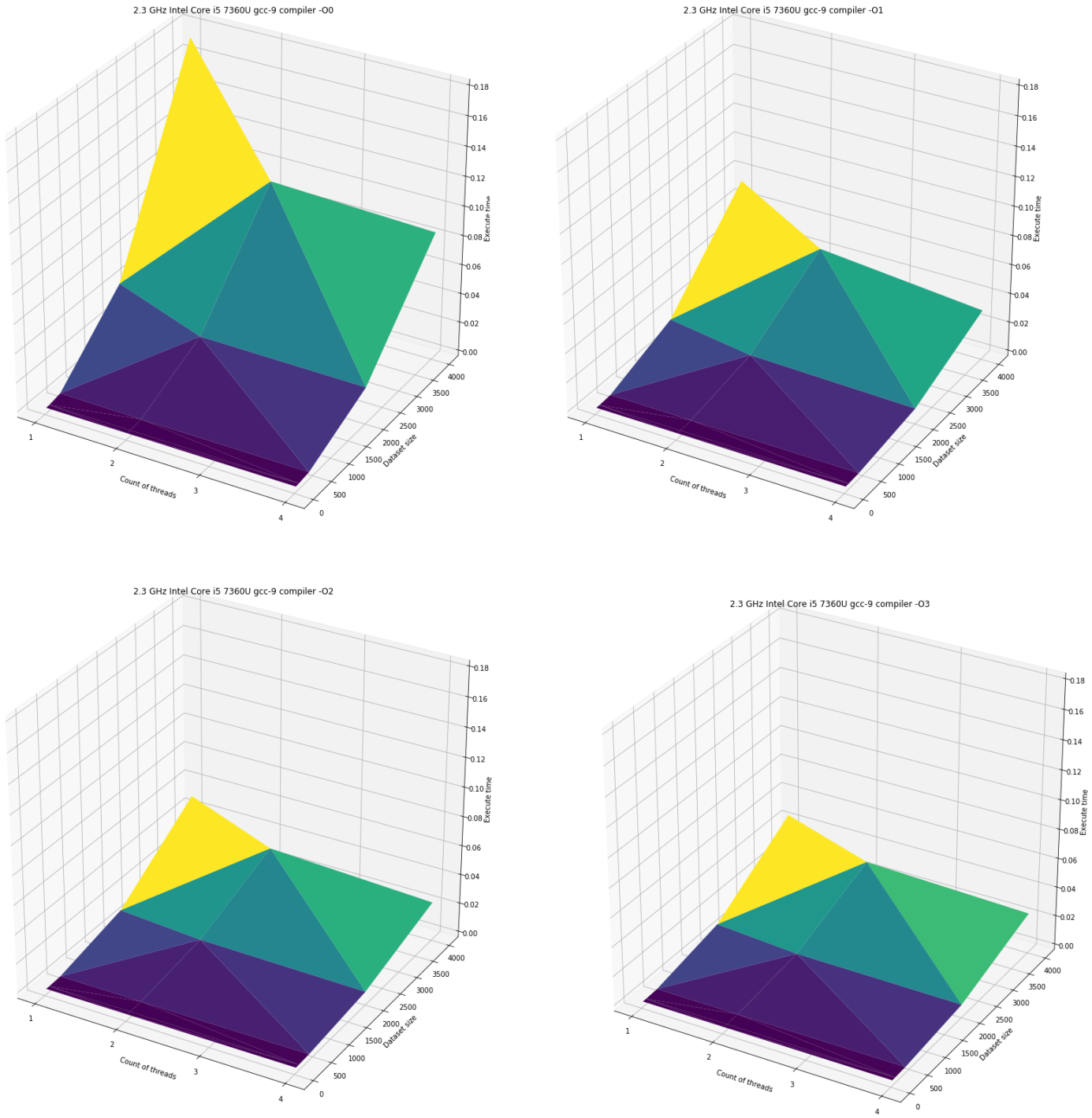
По данным представленным на изображении 3 очевидно, что начальная версия программы значительно ускоряется при изменении оптимизации с *-O1* на *-O2* это говорит о том, что программа изначально оптимизирована не слишком хорошо. Далее приведем график для программы из раздела (3).



Изображение 4 (Итоговая последовательная версия программы)

Из данных представленных на изображении 4 видно, что в итоговой версии программы оптимизации -O1, -O2, -O3 показывают очень близкие результаты, что говорит о том, что итоговая программа оптимизирована в достаточной степени. Также можно заметить в сравнении с данным на изображении 3, что получено также небольшое ускорение работы программы.

Далее приведем сравнение скорости работы итоговой параллельной версии программы с различными оптимизаторами на *2.3 GHz Intel Core i5 7360U*, не будем приводить таблицы с данными, по которым производилось построение графиков, с ними можно будет ознакомиться в приложении 3.



Изображение 5 (Итоговая параллельная версия программы с различными оптимизациями)

Из приведенных графиков очевидно, что результаты описанные для последовательной программы подтверждаются и в параллельной ее версии. Что в очередной раз доказывает правильность выбора блочного алгоритма (4000 запусков).

Анализ полученных результатов

Из проведенных исследований результаты которых приведены в предыдущем разделе можно сделать множество выводов.

Скорость выполнения программы перестает линейно возрастать в зависимости от количества потоков на которых выполняется программа по причине того, что накладные расходы на создание порции нитей значительно увеличиваются, и проанализировав результаты выполнения можно заметить, что при меньших размерах данных программа перестает ускоряться при меньшем числе потоков. Это подтверждает идею о том, что накладные расходы на создание нитей - основная причина недостаточной масщтабируемости программы.

Блочный алгоритм позволяет очень хорошо ускорить программу и избежать зависимости по данным при параллельном выполнении программы.

Анализируя графики и находя минимумы времени в проекции трехмерного графика на плоскость с соответствующим размером данных можно установить оптимальное число потоков для данного размера данных. Например, если рассматривать *Polus* и размер данных равный 4000, то можно заметить, что минимум времени в проекции трехмерного графика на плоскость соответствующую размеру данных равному 4000 достигается при 32 потоках.

Также из полученных данных видно, что произведенная оптимизация последовательной программы значительно ускоряет работу и является предельной.

Выводы

Работа по улучшению и разработке параллельной версии программы при помощи средств *OpenMP* позволила значительно ускорить полученную версию программы и провести ряд экспериментов с такими компьютерами как *Polus* и *Bluegene.*

OpenMP - это крайне удобная в использовании технология, которая позволяет быстро получить качественный результат и получить значительный прирост производительности на различных системах.

Приложение 1

Файл с программой приложен к отчету *(gemver.c, gemver.h)*.

Приложение 2

Файл скрипта запуска приложен к отчету *(script.py)*.

Приложение 3

Файл ноутбука с таблицами всех замеров и построением графиков приложен к отчету *(pad.ipynb)*.