МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
имени М.В.Ломоносова

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ

Практикум по курсу
" Суперкомпьютеры и параллельная обработка данных"

Разработка параллельной версии программы для задачи "Уравнение теплопроводности в трехмерном пространстве" ("Heat equation over 3D data domain")

ОТЧЕТ

о выполненном задании
студента 320 учебной группы факультета ВМК МГУ

Шарипова Саита Равильевича

Лектор: доцент, к.ф.-м.н. Бахтин Владимир Александрович

Москва, 2019 г

Оглавление

Постановка задачи

1. Реализовать параллельную версию программы для задачи "Уравнение теплопроводности в трехмерном пространстве" ***("Heat equation over 3D data domain")*** с помощью технологий параллельного программирования *OpenMP*.
2. Протестировать полученную программу на вычислительных комплексах *Bluegene/P*, *Polus*, а также на личном ПК.
3. Исследовать эффективность полученной программы.
4. Исследовать масштабируемость полученной программы.
5. Построить графики зависимости времени исполнения от числа ядер/процессоров для различного объёма входных данных.
6. Для каждого набора входных данных найти количество ядер/процессоров, при котором время выполнения задачи перестаёт уменьшаться.
7. Определить основные причины недостаточной масштабируемости программы при максимальном числе используемых ядер/процессоров.

Описание алгоритма

1. Опишем полученную для распараллеливания последовательную программу:
   Были получены два файла: заголовочный файл – *heat-3d.h* и файл с исходным кодом – *heat-3d.c*.
   Исходный код *heat-3d.c* содержал функции:

```
void bench_timer_start() - инициализация времени старта
void bench_timer_stop() - инициализация времени финиша
static double rtclock() - функция подсчета времени
void bench_timer_print() - вывод времени выполнения программы
static void init_array (int n, double A[n][n][n], double B[n][n][n]) -
инициализация массивов A и B
static void print_array(int n, double A[n][n][n]) - печать массива
static void kernel_heat_3d(int tsteps, int n, double A[n][n][n], double
B[n][n][n]) - основная функция, работающая с массивами A и B
```

Самой ресурсозатратной в плане выполняемых вычислений в предоставленной для распараллеливания программе является функция `static void kernel_heat_3d`. Код этой функции предоставлен ниже:

```
static void kernel_heat_3d(int tsteps, int n, double A[n][n][n], double B[n][n][n]) {
    int t, i, j, k;

    for (t = 1; t <= TSTEPS; t++) {

        for (i = 1; i < n - 1; i++) {
            for (j = 1; j < n - 1; j++) {
                for (k = 1; k < n - 1; k++) {
                    B[i][j][k] = 0.125 * (A[i + 1][j][k] - 2.0 * A[i][j][k] + A[i - 1][j][k])
                               + 0.125 * (A[i][j + 1][k] - 2.0 * A[i][j][k] + A[i][j - 1][k])
                               + 0.125 * (A[i][j][k + 1] - 2.0 * A[i][j][k] + A[i][j][k - 1])
                               + A[i][j][k];
                }
            }
        }

        for (i = 1; i < n - 1; i++) {
            for (j = 1; j < n - 1; j++) {
                for (k = 1; k < n - 1; k++) {
                    A[i][j][k] = 0.125 * (B[i + 1][j][k] - 2.0 * B[i][j][k] + B[i - 1][j][k])
                               + 0.125 * (B[i][j + 1][k] - 2.0 * B[i][j][k] + B[i][j - 1][k])
                               + 0.125 * (B[i][j][k + 1] - 2.0 * B[i][j][k] + B[i][j][k - 1])
                               + B[i][j][k];
                }
            }
        }

    }
}
```

В данной функции ***t*** раз выполняются два тройных цикла. В первом тройном цикле обновляется массив B, полагаясь на значения в массиве A. Во втором тройном цикле обновляется массив A, полагаясь на значения в массиве B. Сразу отметим, что обход матрицы в “правильном” порядке – циклы сначала по первому измерению, потом по второму и третьему соответственно. В плане вычислений здесь множество операций – в будущем можно будет “привести подобные”. Причем операции довольно дорогие, так как речь идет о умножении/сложении double. Также “неприятным моментом” является то, что при вычислениях обращение к элементам массивов происходит к соседним элементам по всем трем измерениям – при больших объемах данных это может

плохо сказаться, так как вся матрица в КЭШ не поместится.

Сложность данного алгоритма составляет:

$$t * n^3 * 2 \text{ цикла } * 15 \text{ операций} = 30 * t * n^3$$

При n = 200 и t = 1000 – самые большие значения, на которых будет производиться тестирование, в данной будет произведено:

$$30 * t * n^3 = 30 * 1000 * 200^3 = 240 * 10^9 = 240 \text{ млрд операций.}$$

Изложив основные аспекты предоставленной для распараллеливания программы, перейдем к описанию предпринятых к ней модификациям и эвристикам.

Предпринятые модификации к последовательной программе

Предоставленная последовательная программа имела ряд незначительных дефектов, которые желательно устранить перед распараллеливанием:

1) В основной функции программы - `static void kernel_heat_3d(int tsteps, int n, double A[n][n][n], double B[n][n][n])` в фрагменте `for (t = 1; t <= TSTEPS; t++) { …` вместо TSTEPS следует написать tsteps, иначе теряется смысл передачи этого параметра в аргументы функции. Также для предоставленной ниже реализации алгоритма это особенно актуально, так как в ней идентификатора TSTEPS не будет вовсе.
2) В соответствии с рекомендацией в описании задания: “Для замера времени рекомендуется использовать вызовы функции omp_get_wtime, общее время работы должно определяться временем самого медленного из процессов/нитей.” несмотря на реализованные функции: `void bench_timer_start()`, `void bench_timer_stop()`, `static double rtclock()` и `void bench_timer_print()` было принято решение - для подсчета времени выполнения программы использовать функцию `omp_get_wtime()`.
3) В предоставленном заголовочном файле *heat-3d.h* были обозначены следующие датасеты для тестирования:

Параметры датасетов для тестирования					
	MINI	SMALL	MEDIUM	LARGE	EXTRALARGE
n	10	20	40	120	200
t	20	40	100	500	1000
Параметр командной строки	1	2	3	4	5

Было принято решение задавать датасет с помощью параметра командной строки. Соответствующая информация предоставлена в таблице.

4) В функции `static void print_array(int n, double A[n][n][n])`, печатающей массив A находился фрагмент

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        for (int k = 0; k < n; k++) {

            if ((i * n * n + j * n + k) % 20 == 0) {
                fprintf(stderr, "\n");
            }

            fprintf(stdout, "%0.2lf ", A[i][j][k]);


        }
        fprintf(stdout, "\n");
    }
}
```

по сути “неправильно” печатающий перевод строки (то есть в конце каждого цикла по ***k***) лишь в случае, когда ***n***=20. Этот фрагмент, подчеркнутый красным, был заменен на фрагмент, подчеркнутый зеленым.

Построение параллельной версии программы

1) В функции `static void kernel_heat_3d(int tsteps, int n, double A[n][n][n], double B[n][n][n])` производится множество операций при релаксации массива A и B. Заметим, что вычисления типа `2.0 * A[i][j][k]`, умножения на `0.125` и другие выполняются несколько раз. Приведем подобные тем самым сократив число вычислений. (Возможно, что компилятор и оптимизатор делают это и сами, однако при данной модификации скорость работы увеличилась, это можно объяснить тем, что в данном фрагменте

множество умножений вперемешку с сложением, что усложняет автоматическое развертывание операций):

```
B[i][j][k] = 0.125 * (A[i + 1][j][k] - 2.0 * A[i][j][k] + A[i - 1][j][k])
           + 0.125 * (A[i][j + 1][k] - 2.0 * A[i][j][k] + A[i][j - 1][k])
           + 0.125 * (A[i][j][k + 1] - 2.0 * A[i][j][k] + A[i][j][k - 1])
           + A[i][j][k];
```

```
double residue;
B[i][j][k] = A[i][j][k] * 0.25;
residue = A[i + 1][j][k] + A[i - 1][j][k] + A[i][j + 1][k]
          + A[i][j - 1][k] + A[i][j][k + 1] + A[i][j][k - 1];
residue *= 0.125;
B[i][j][k] += residue;
```

В итоге вместо 15 операций в цикле мы выполняем 8.

2) В функции `static void kernel_heat_3d(int tsteps, int n, double A[n][n][n], double B[n][n][n])` нельзя распараллеливать внешний цикл по ***t***, так как он задает шаги в нашей функции. Внутри этого цикла обновляются значения массивов A и B. Причем обновление этих массивов опирается на их предыдущие значения. Значит для сохранения изначальной работоспособности программы важно, чтобы цикл по ***t***, выполнялся последовательно.
3) Посмотрим, что происходит внутри цикла по ***t*** – в нем два тройных цикла, похожих по структуре. В первом обновляется массив B, полагаясь на значения в массиве A. Во втором тройном цикле обновляется массив A, полагаясь на значения в массиве B. Важно учесть, что на каждом шаге сначала должен обновляться массив B, а уже за ним массив A. Поэтому задавать общие нити для этих двух циклов нельзя. (либо необходимо устанавливать барьер между тройными циклами). Итак, так как зависимости по данным нет, то значит мы можем распараллелить внешний цикл по ***i*** в каждом из тройных циклов с помощью директивы `#pragma omp parallel for private(i, j, k)`, тем самым

получив ***базовую версию параллельной программы*** ***kernel_heat_3d_parallel_base***:

Базовая версия параллельной программы:

```
static void kernel_heat_3d_parallel_base(
            int tsteps, int n,
            double A[n][n][n], double B[n][n][n]) {
    int t, i, j, k;

    for (t = 1; t <= tsteps; t++) {

       #pragma omp parallel for private(i, j, k)
        for (i = 1; i < n - 1; i++) {
            for (j = 1; j < n - 1; j++) {
                for (k = 1; k < n - 1; k++) {
                    double residue;
                    B[i][j][k] = A[i][j][k] * 0.25;
                    residue = A[i + 1][j][k] + A[i - 1][j][k]
                              + A[i][j + 1][k] + A[i][j - 1][k]
                              + A[i][j][k + 1] + A[i][j][k - 1];
                    residue *= 0.125;
                    B[i][j][k] += residue;
                }
            }
        }

        #pragma omp parallel for private(i, j, k)
        for (i = 1; i < n - 1; i++) {
            for (j = 1; j < n - 1; j++) {
                for (k = 1; k < n - 1; k++) {
                    double residue;
                    A[i][j][k] = B[i][j][k] * 0.25;
                    residue = B[i + 1][j][k] + B[i - 1][j][k]
                              + B[i][j + 1][k] + B[i][j - 1][k]
                              + B[i][j][k + 1] + B[i][j][k - 1];
                    residue *= 0.125;
                    A[i][j][k] += residue;
                }
            }
        }
    }
}
```

4) Для небольших датасетов, ***n*** <= 40, т. е. для MINI, SMALL и MEDIUM в базовой параллельной версии алгоритма разобьем внутренние циклы по ***k*** на полосы ширины 4. Тогда цикл по ***k*** будет перебирать полосы (**Разделение на полосы**), а внутри этого цикла для увеличения скорости пропишем операции для каждого элемента полосы вручную. Это приведет приводит к сближению

обращений к ячейкам памяти, при этом производительность повышается за счет кэширования. На практике при тестировании эта манипуляция действительно увеличивает скорость программы по сравнению с базовым параллельным алгоритмом. Получим функцию ***kernel_heat_3d_parallel_mini***:

Улучшенная версия параллельной программы для небольших датасетов:

```
static void kernel_heat_3d_parallel_mini(
       int tsteps, int n, double A[n][n][n], double B[n][n][n]) {
    int t, i, j, k;

    for (t = 1; t <= tsteps; t++) {

        #pragma omp parallel for private(i, j, k)
        for (i = 1; i < n - 1; i++) {
            for (j = 1; j < n - 1; j++) {
                for (k = 1; k + 3 < n - 1; k+=4) {
                    double residue;
                    B[i][j][k] = A[i][j][k] * 0.25;
                    residue = A[i + 1][j][k] + A[i - 1][j][k]
                            + A[i][j + 1][k] + A[i][j - 1][k]
                            + A[i][j][k + 1] + A[i][j][k - 1];
                    residue *= 0.125;
                    B[i][j][k] += residue;

                    B[i][j][k + 1] = A[i][j][k + 1] * 0.25;
                    residue = A[i + 1][j][k + 1] + A[i - 1][j][k + 1]
                            + A[i][j + 1][k + 1] + A[i][j - 1][k + 1]
                            + A[i][j][k + 2] + A[i][j][k];
                    residue *= 0.125;
                    B[i][j][k + 1] += residue;

                    B[i][j][k + 2] = A[i][j][k + 2] * 0.25;
                    residue = A[i + 1][j][k + 2] + A[i - 1][j][k + 2]
                            + A[i][j + 1][k + 2] + A[i][j - 1][k + 2]
                            + A[i][j][k + 3] + A[i][j][k + 1];
                    residue *= 0.125;
                    B[i][j][k + 2] += residue;

                    B[i][j][k + 3] = A[i][j][k + 3] * 0.25;
                    residue = A[i + 1][j][k + 3] + A[i - 1][j][k + 3]
                            + A[i][j + 1][k + 3] + A[i][j - 1][k + 3]
                            + A[i][j][k + 4] + A[i][j][k + 2];
                    residue *= 0.125;
                    B[i][j][k + 3] += residue;
                }
            }
        }


        #pragma omp parallel for private(i, j, k)
        for (i = 1; i < n - 1; i++) {
            for (j = 1; j < n - 1; j++) {
                for (k = 1; k + 3 < n - 1; k+=4) {
                    double residue;
```

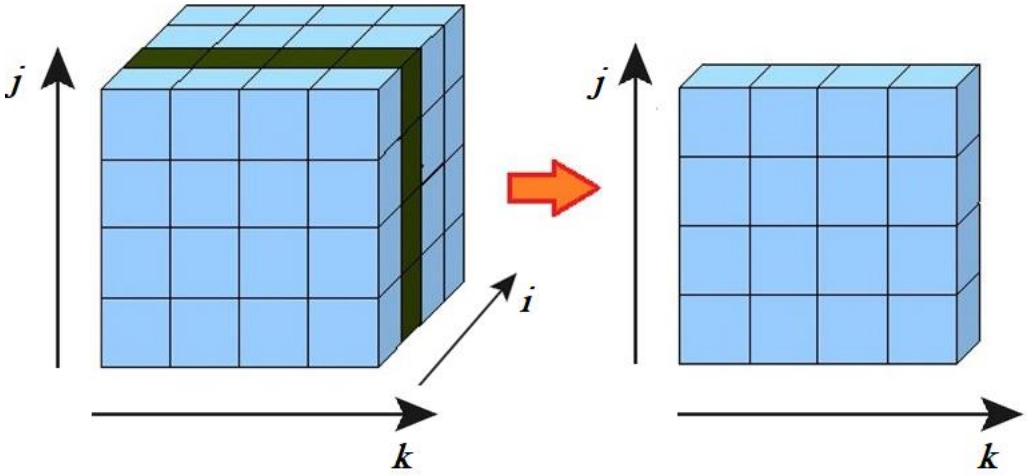
```
                    A[i][j][k] = B[i][j][k] * 0.25;
                    residue = B[i + 1][j][k] + B[i - 1][j][k]
                            + B[i][j + 1][k] + B[i][j - 1][k]
                            + B[i][j][k + 1] + B[i][j][k - 1];
                    residue *= 0.125;
                    A[i][j][k] += residue;

                    A[i][j][k + 1] = B[i][j][k + 1] * 0.25;
                    residue = B[i + 1][j][k + 1] + B[i - 1][j][k + 1
                            + B[i][j + 1][k + 1] + B[i][j - 1][k + 1]
                            + B[i][j][k + 2] + B[i][j][k];
                    residue *= 0.125;
                    A[i][j][k + 1] += residue;

                    A[i][j][k + 2] = B[i][j][k + 2] * 0.25;
                    residue = B[i + 1][j][k + 2] + B[i - 1][j][k + 2]
                            + B[i][j + 1][k + 2] + B[i][j - 1][k + 2]
                            + B[i][j][k + 3] + B[i][j][k + 1];
                    residue *= 0.125;
                    A[i][j][k + 2] += residue;

                    A[i][j][k + 3] = B[i][j][k + 3] * 0.25;
                    residue = B[i + 1][j][k + 3] + B[i - 1][j][k + 3]
                            + B[i][j + 1][k + 3] + B[i][j - 1][k + 3]
                            + B[i][j][k + 4] + B[i][j][k + 2];
                    residue *= 0.125;
                    A[i][j][k + 3] += residue;
                }
            }
        }
    }
}
```

5) Для больших датасетов, $\boldsymbol{n} > 40$, т. е. для LARGE и EXTRALARGE применим метод **разделения на блоки** - область итераций разбивается на прямоугольные блоки. Если представить трехмерные массивы `A[n][n][n], B[n][n][n]` как кубы, где измерение $\boldsymbol{i}$ задаёт плоскости, то в данном алгоритме будем разбивать каждую плоскость с измерениями $\boldsymbol{j}$, $\boldsymbol{k}$ на блоки размера *block_size*×*block_size*, где оптимальный размер блока определим эмпирическим путем. При таком подходе распределение данных в памяти компактируется и, следовательно, повышается эффективность кэширования на вычислительной системе с общей памятью или размещение данных в распределенной памяти.



При условии хранения в памяти массивов по строкам, что характерно для языка C - доступ к элементам вида `A[i][j][k + 1]` осуществляется с шагом 1 (каждый элемент размещается рядом с предыдущим элементом в соответствии с внутренним циклом по ***k***). Однако доступ к элементам вида `A[i][j + 1][k]` осуществляется с шагом ***n*** (длина всей строки). Таким образом при ссылках на элементы вида `A[i][j + 1][k]` и `A[i + 1][j][k]` кэш используется неэффективно (в кэш загружается излишне много значений элементов, из которых реально нужна всего ***1/n*** для `A[i][j + 1][k]`).

Разбиение матрицы на блоки делит область итераций на прямоугольники (квадраты в частном случае), размер данных в которых не превышают размера кэша. При желании полностью поместить в кэш *block_size*×*block_size* (само собой, *block_size* < ***n***) чисел следует изменить алгоритм следующим образом:

Параллельная для больших датасетов при сравнительно небольшом числе нитей:

```
static void kernel_heat_3d_parallel_normal(
        int tsteps, int n, double A[n][n][n], double B[n][n][n], int block_size) {
    int t, i, j, k, kk, jj;

    for (t = 1; t <= tsteps; t++) {

        #pragma omp parallel for private(i, j, k, kk, jj)
        for (jj = 1; jj < n - 1; jj += block_size) {
        for (i = 1; i < n - 1; i++) {
                for (kk = 1; kk < n - 1; kk += block_size) {
                    for (j = jj; j < MIN(n - 1, jj + block_size); j++) {
                        for (k = kk; k < MIN(n - 1, kk + block_size); k++) {
                            double residue;
                            B[i][j][k] = A[i][j][k] * 0.25;
                            residue = A[i + 1][j][k] + A[i - 1][j][k]
```

```
                                + A[i][j + 1][k] + A[i][j - 1][k]
                                + A[i][j][k + 1] + A[i][j][k - 1];
                            residue *= 0.125;
                            B[i][j][k] += residue;
                        }
                    }
                }
            }
        }

        #pragma omp parallel for private(i, j, k, kk, jj)
        for (jj = 1; jj < n - 1; jj += block_size) {
        for (i = 1; i < n - 1; i++) {
                for (kk = 1; kk < n - 1; kk += block_size) {
                    for (j = jj; j < MIN(n - 1, jj + block_size); j++) {
                        for (k = kk; k < MIN(n - 1, kk + block_size); k++) {
                            double residue;
                            A[i][j][k] = B[i][j][k] * 0.25;
                            residue = B[i + 1][j][k] + B[i - 1][j][k]
                                    + B[i][j + 1][k] + B[i][j - 1][k]
                                    + B[i][j][k + 1] + B[i][j][k - 1];
                            residue *= 0.125;
                            A[i][j][k] += residue;
                        }
                    }
                }
            }
        }
    }
}
```

Параллельная для больших датасетов при сравнительно большом числе нитей:

```
static void kernel_heat_3d_parallel_big(
        int tsteps, int n, double A[n][n][n], double B[n][n][n], int block_size) {
    int t, i, j, k, kk, jj;

    for (t = 1; t <= tsteps; t++) {

        #pragma omp parallel private(i, j, k, kk, jj)
        for (i = 1; i < n - 1; i++) {
            for (jj = 1; jj < n - 1; jj += block_size) {
                for (kk = 1; kk < n - 1; kk += block_size) {
                    for (j = jj; j < MIN(n - 1, jj + block_size); j++) {
                        for (k = kk; k < MIN(n - 1, kk + block_size); k++) {
                            double residue;
                            B[i][j][k] = A[i][j][k] * 0.25;
                            residue = A[i + 1][j][k] + A[i - 1][j][k]
                                    + A[i][j + 1][k] + A[i][j - 1][k]
                                    + A[i][j][k + 1] + A[i][j][k - 1];
                            residue *= 0.125;
                            B[i][j][k] += residue;
                        }
                    }
                }
            }
        }
```

```
        #pragma omp parallel private(i, j, k, kk, jj)
        for (i = 1; i < n - 1; i++) {
            for (jj = 1; jj < n - 1; jj += block_size) {
                for (kk = 1; kk < n - 1; kk += block_size) {
                    for (j = jj; j < MIN(n - 1, jj + block_size); j++) {
                        for (k = kk; k < MIN(n - 1, kk + block_size); k++) {
                            double residue;
                            A[i][j][k] = B[i][j][k] * 0.25;
                            residue = B[i + 1][j][k] + B[i - 1][j][k]
                                    + B[i][j + 1][k] + B[i][j - 1][k]
                                    + B[i][j][k + 1] + B[i][j][k - 1];
                            residue *= 0.125;
                            A[i][j][k] += residue;
                        }
                    }
                }
            }
        }
    }
}
```

Теперь во внутренних циклах доступен квадрат из *block_size*×*block_size* элементов матриц, который полностью помещается в кэш.

Приведены две версии параллельного алгоритма для большого датасета – при сравнительно большом и малом числе нитей. Это связано с тем, что на больших датасетов n довольно велико, а число блоков заведомо меньше n. Если в расположении вычислительной системы сравнительно много нитей, то выгоднее направить их на распараллеливание внешнего цикла по i, содержащего n витков, так как при распараллеливании цикла по блокам в этом случае некоторое число оставшихся нитей просто бы простаивали. Если же изначально нитей сравнительно мало, то напротив выгоднее распараллеливать цикл по блокам – в этом случае мы его вынесем за цикл по i, сделав внешним.

Для LARGE DATASET размер блока задавался как *block_size* = 5, при n = 120.

Для EXTRALARGE DATASET размер блока задавался как *block_size* = 10, при n = 120. Размер блока подбирался экспериментальным путем.

Данное решение положительно проявило себя на практике в плане времени вычислений в сравнении с базовой параллельной программой.

6) В итоге была реализована функция, которая в зависимости от размера датасета и числа нитей в распоряжении вычислительной машины вызывает соответствующую параллельную функцию:

```
static void parallel_program(int tsteps, int n, double A[n][n][n], double B[n][n][n],
                             int id_dataset, int block_size, int count_threads) {
    if (id_dataset == 1 || id_dataset == 2 || id_dataset == 3) {
        // небольшой датасет
        kernel_heat_3d_parallel_mini(tsteps, n, A, B);
    } else if ((id_dataset == 4 || id_dataset == 5) && count_threads <= n / block_size)
{
        // большой датасет и число нитей позволяют распараллелить по блокам
        kernel_heat_3d_parallel_normal(tsteps, n, A, B, block_size);
    } else {
        // большой датасет, а число нитей позволяет распараллелить внешний цикл
        kernel_heat_3d_parallel_big(tsteps, n, A, B, block_size);
    }
}
```

Тестирование

Информация о тестировании

Было произведено тестирование изначальной последовательной программы и полученной параллельной программы. Тестирование производилось на суперкомпьютерные вычислительные ресурсы факультета ВМК - *Bluegene/P* и *Polus*, а также на личном компьютере.

Bluegene/P – массивно-параллельная вычислительная система, которая состоит из двух стоек, включающих 8192 процессорных ядер (2 x 1024 четырехъядерных вычислительных узлов). Один вычислительный узел содержит четыре микропроцессорных ядра PowerPC 450 (4-way SMP), 4 потока.

Polus – параллельная вычислительная система, состоящая из 5 вычислительных узлов. Один вычислительный узел содержит 2 десятиядерных процессора IBM POWER8 (каждое ядро имеет 8 потоков) всего 160 потоков.

Персональный компьютер имеет характеристики: Intel(R) Core(TM) i5-7200(U) CPU @ 2.50GHz 2.70 GHz, 2 ядра, 4 потока.

Тестирование проводилось на 1-ом узле каждого суперкомпьютера, поэтому в распоряжении программы было 4 потока на *Bluegene/P* и ПК, и 160 потоков на *Polus*.

Итак, программа тестировалась на вычислительных машинах со следующим числом потоков:

Bluegene/P – 1, 2, 3, 4 потока.
Polus – 1, 2, 4, 8, 16, 32, 64, 100, 128, 200 потоков.
Персональный компьютер – 1, 2, 3, 4 потока.

Замеры времени производились с помощью функции omp_get_wtime().

Для каждого набора входных данных (MINI, SMALL, MEDIUM, LARGE, EXTRALARGE) происходило от 5 до 10 итерации просчёта с последующим усреднением времени.

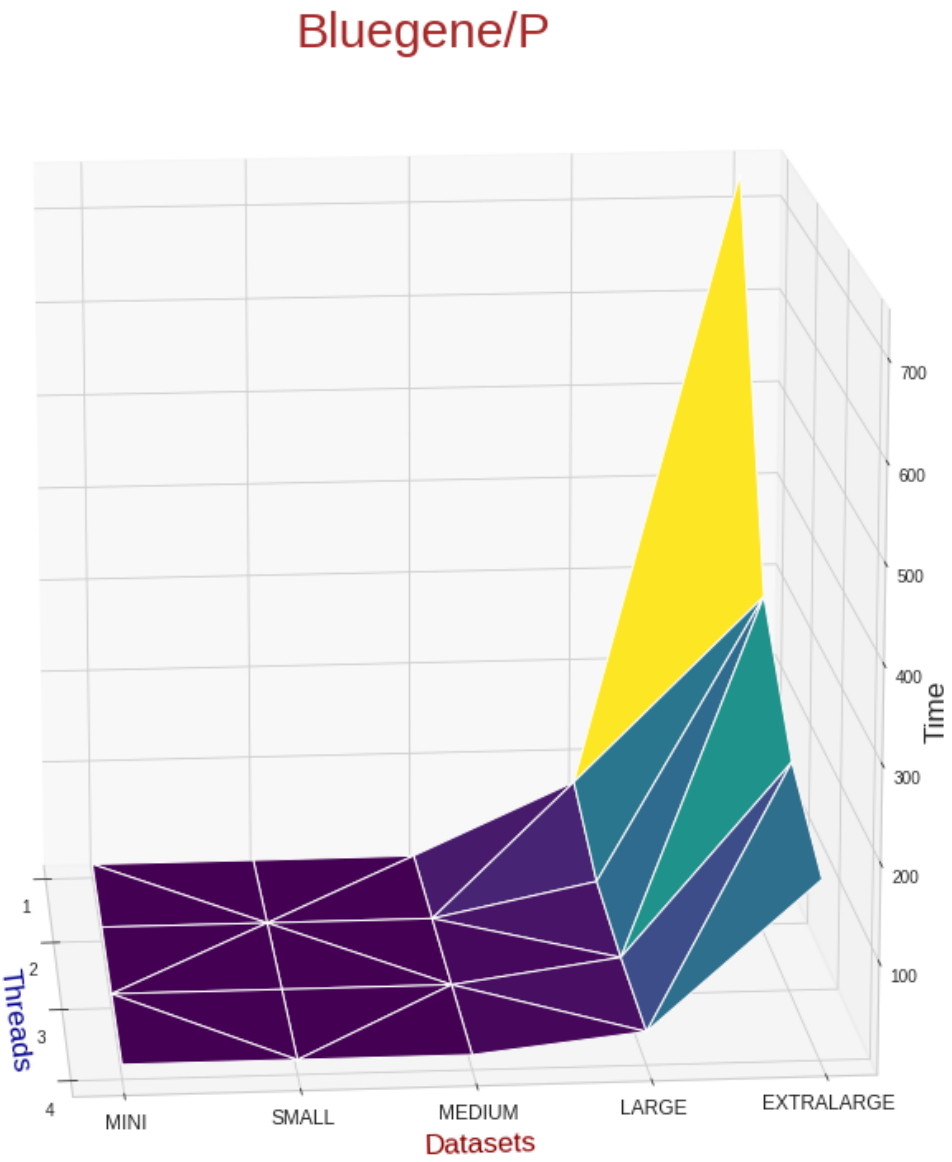
На персональном компьютере также было произведено тестирование с оптимизатором *-O4*.

Далее для каждой вычислительной машины будет предоставлены таблица и график, построенный по её значениям с помощью программы, написанной на языке *Python* с использованием библиотеки *matplotlib* (*jupyter-notebook* с текстом программы в приложении). В таблице зеленым и красным цветами отмечены наилучшее и наихудшее время выполнение программы соответственно для каждого датасета.

Тестирование на Bluegene/P

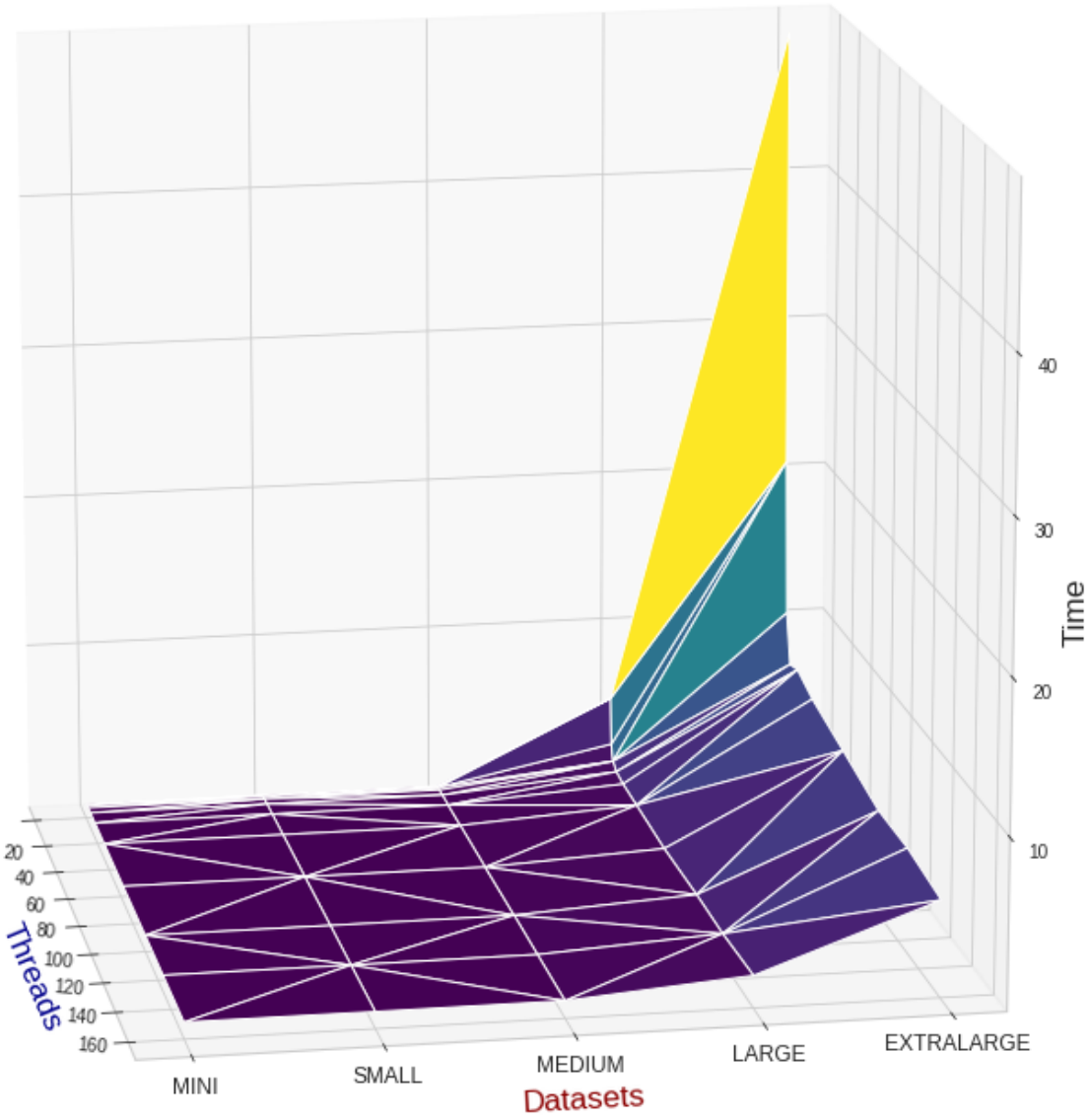
Blugene/P					
Датасет / Число нитей	MINI	SMALL	MEDIUM	LARGE	EXTRALARGE
Sequental	0.001006	0.022771	0.531696	78.023404	733.972330
1	0.001037	0.022687	0.531385	77.989666	733.494111
2	0.000884	0.011546	0.256541	35.885004	337.431614
3	0.000779	0.007900	0.181304	24.354680	224.963574
4	0.000697	0.006754	0.135992	19.272706	170.472973

Из приведенных данных видно, что выполнение программы уменьшается с увеличением числа потоков, а оптимальное число потоков равно четырем.



Тестирование на Polus

Polus					
Датасет / Число нитей	MINI	SMALL	MEDIUM	LARGE	EXTRALARGE
Sequental	0.000095	0.001618	0.037763	5.486201	50.482243
1	0.000108	0.001508	0.034807	5.458145	49.673351
2	0.000152	0.000894	0.018504	2.400558	21.083382
4	0.000209	0.000763	0.014952	1.390190	10.995179
8	0.000344	0.000900	0.010161	1.006221	7.772443
16	0.000669	0.002988	0.011065	0.847479	8.053529
32	0.001359	0.003176	0.012603	0.778199	7.370462
64	0.004173	0.005920	0.023444	0.641834	6.583668
100	0.031290	0.036018	0.068199	0.788291	5.681133
128	0.031729	0.035005	0.068004	0.775733	5.692817
160	0.043137	0.046734	0.079454	1.031876	5.193472

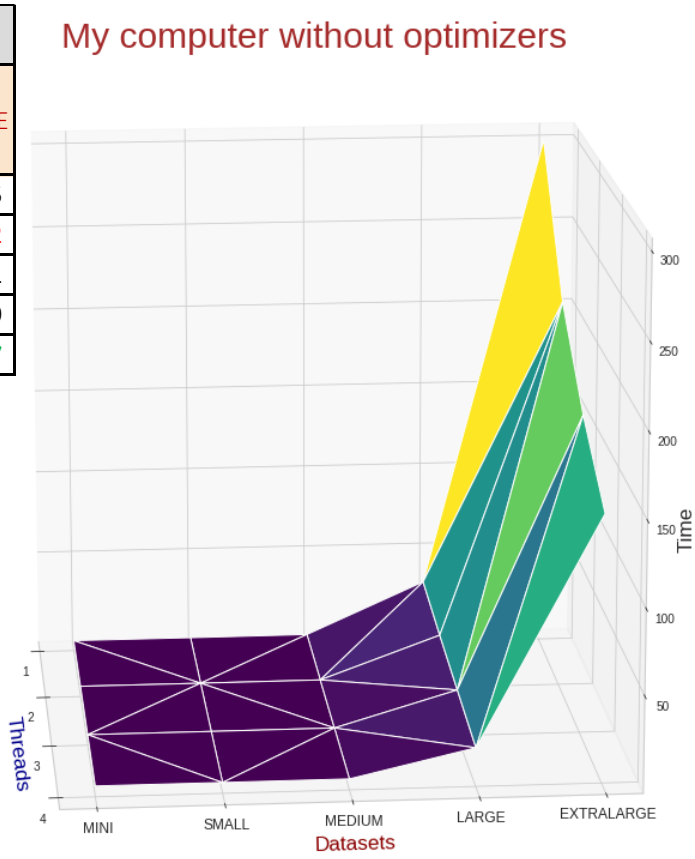


Из приведенных данных видно, что выполнение программы не всегда уменьшается с увеличением числа потоков, а оптимальное число потоков для EXTRALARGE DATASET равно 100 – далее время перестает расти, что можно объяснить тем, что в данном случае распараллеливался цикл по ***i***, в котором ***n*** = 200 витков. А значит при распараллеливании этого цикла с помощью ста потоков на каждую нить придется по 2 итерации. Если же число потоков будет 100 < ***threads*** < 200, то некоторые нити произведут 2 итерации, другие же всего 1 итерации и будут ожидать первых, поэтому время не улучшится, а скорее замедлиться в связи с накладными расходами на создание дополнительных нитей. Также из таблицы видно, что чем меньше датасет, тем меньше оптимальное число нитей – что снова связано с накладными расходами, связанными с нитями. Порой для маленьких объемов данных выгоднее применять последовательную версию алгоритма, что можно увидеть для случаев MINI и отчасти для случая SMALL.
График выглядит “слишком резко” в связи с тем, что так как матрицы трехмерные, то EXTRALARGE сильно превышает остальные датасеты (***n^3***) по объему данных.

Тестирование на персональном компьютере без оптимизации

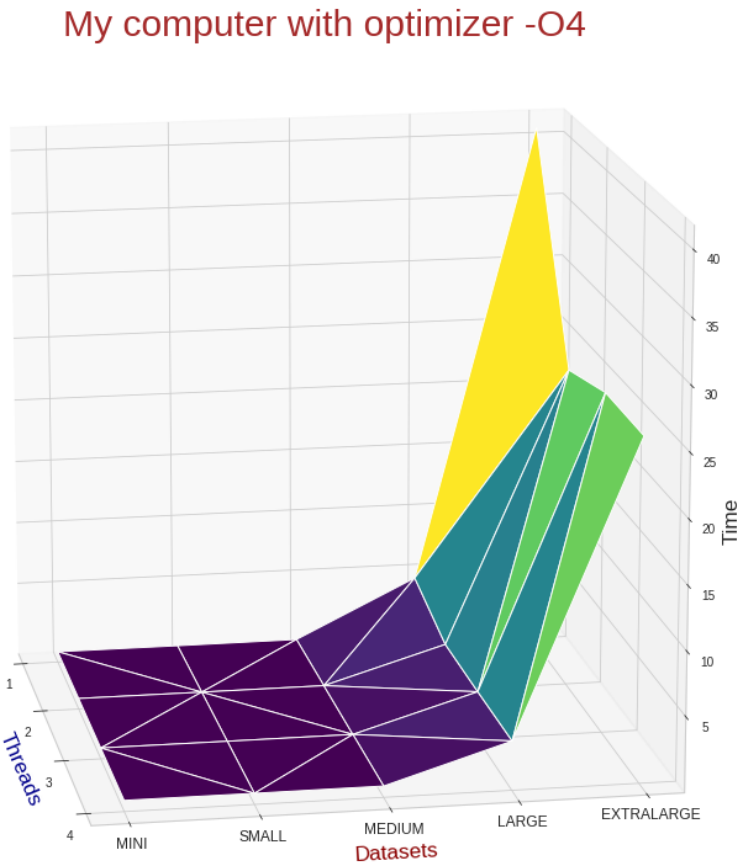
ПК без оптимизации					
Датасет / Число нитей	MINI	SMALL	MEDIUM	LARGE	EXTRALARGE
Sequental	0.000697	0.011113	0.211579	31.550799	299.652555
1	0.000608	0.011028	0.201193	31.487274	301.042312
2	0.000495	0.006707	0.143518	25.695053	224.622641
3	0.000338	0.004924	0.122168	20.895952	180.919949
4	0.000298	0.005453	0.101098	16.225581	148.871947

Из приведенных данных видно, что выполнение программы уменьшается с увеличением числа потоков, а оптимальное число потоков равно четырем.



Тестирование на персональном компьютере с оптимизацией -O4

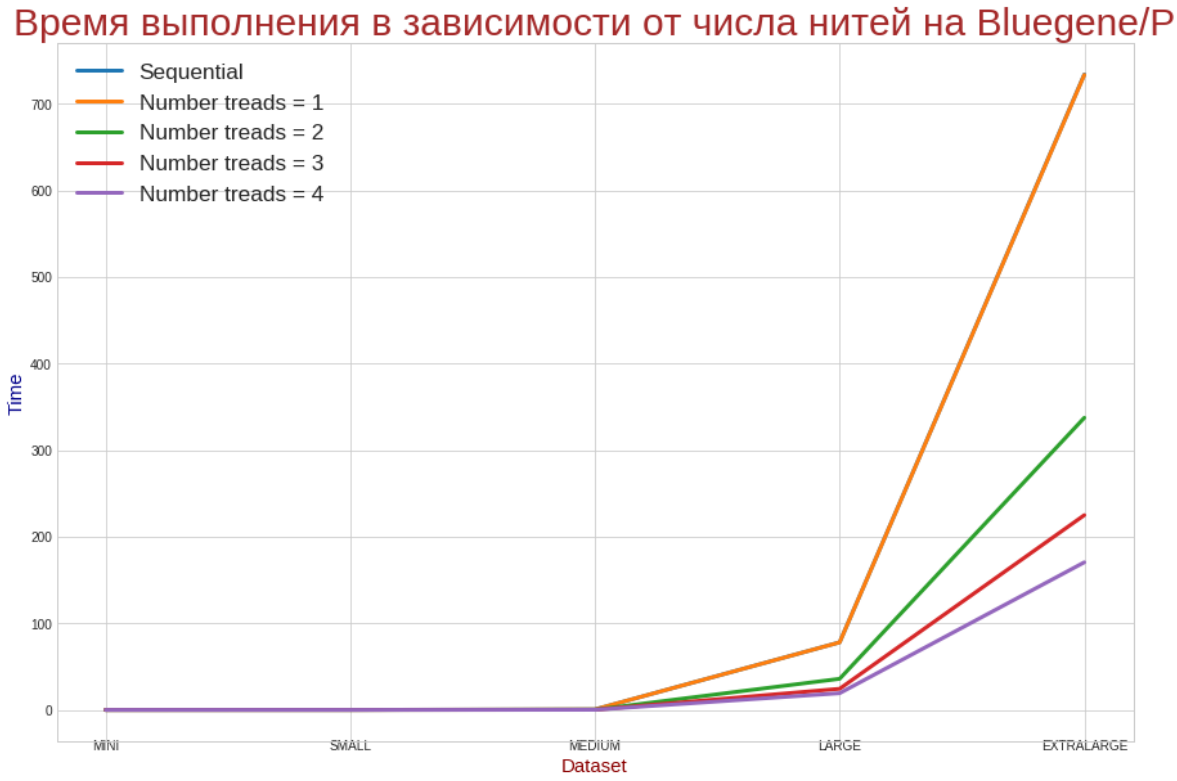
ПК с оптимизацией -O4					
Датасет / Число нитей	MINI	SMALL	MEDIUM	LARGE	EXTRALARGE
Sequental	0.000050	0.000759	0.018455	2.979332	29.423754
1	0.000099	0.001296	0.025618	4.656676	40.968206
2	0.000132	0.000610	0.015282	2.863539	24.437348
3	0.000134	0.000611	0.015180	2.873167	25.778034
4	0.003697	0.000551	0.017555	2.955054	25.674309

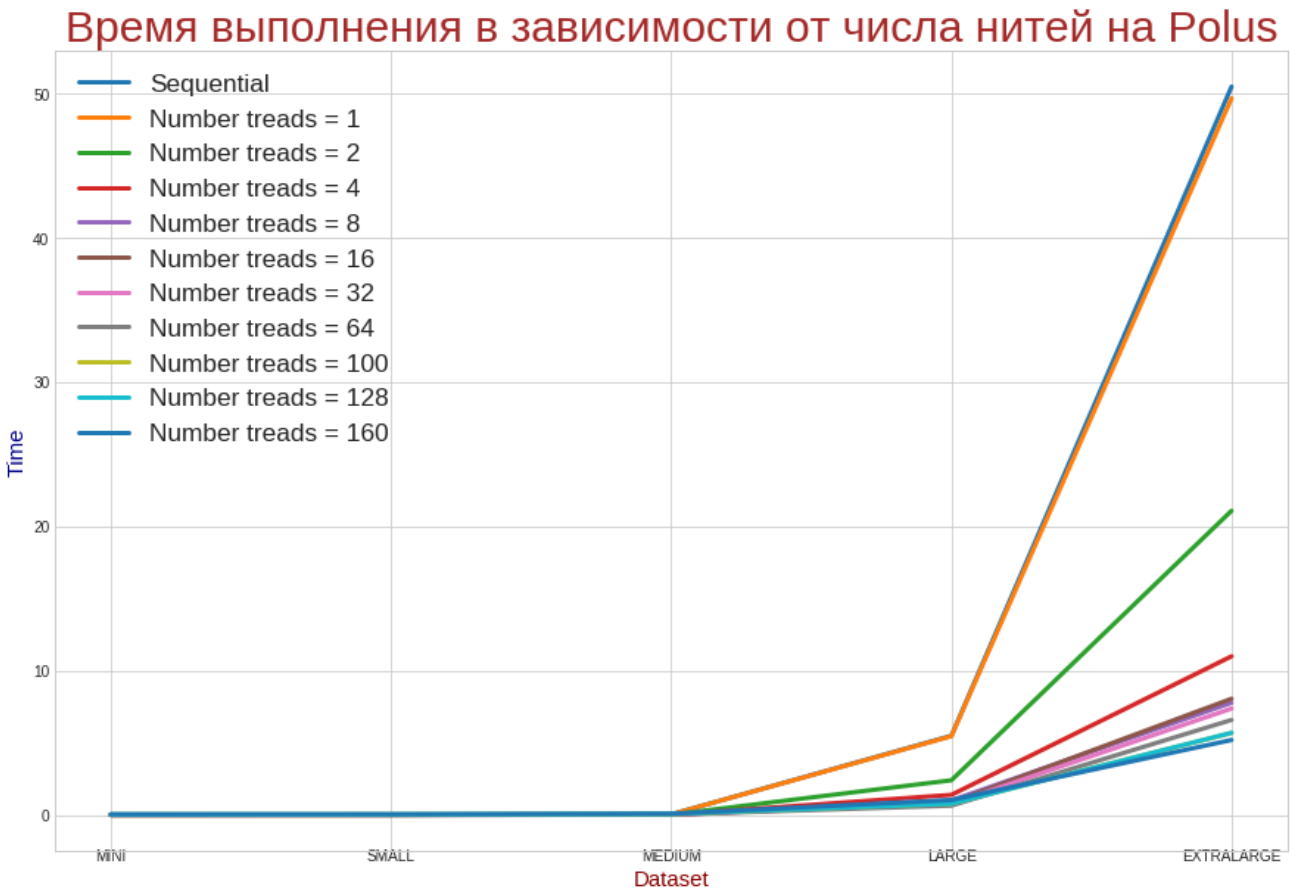


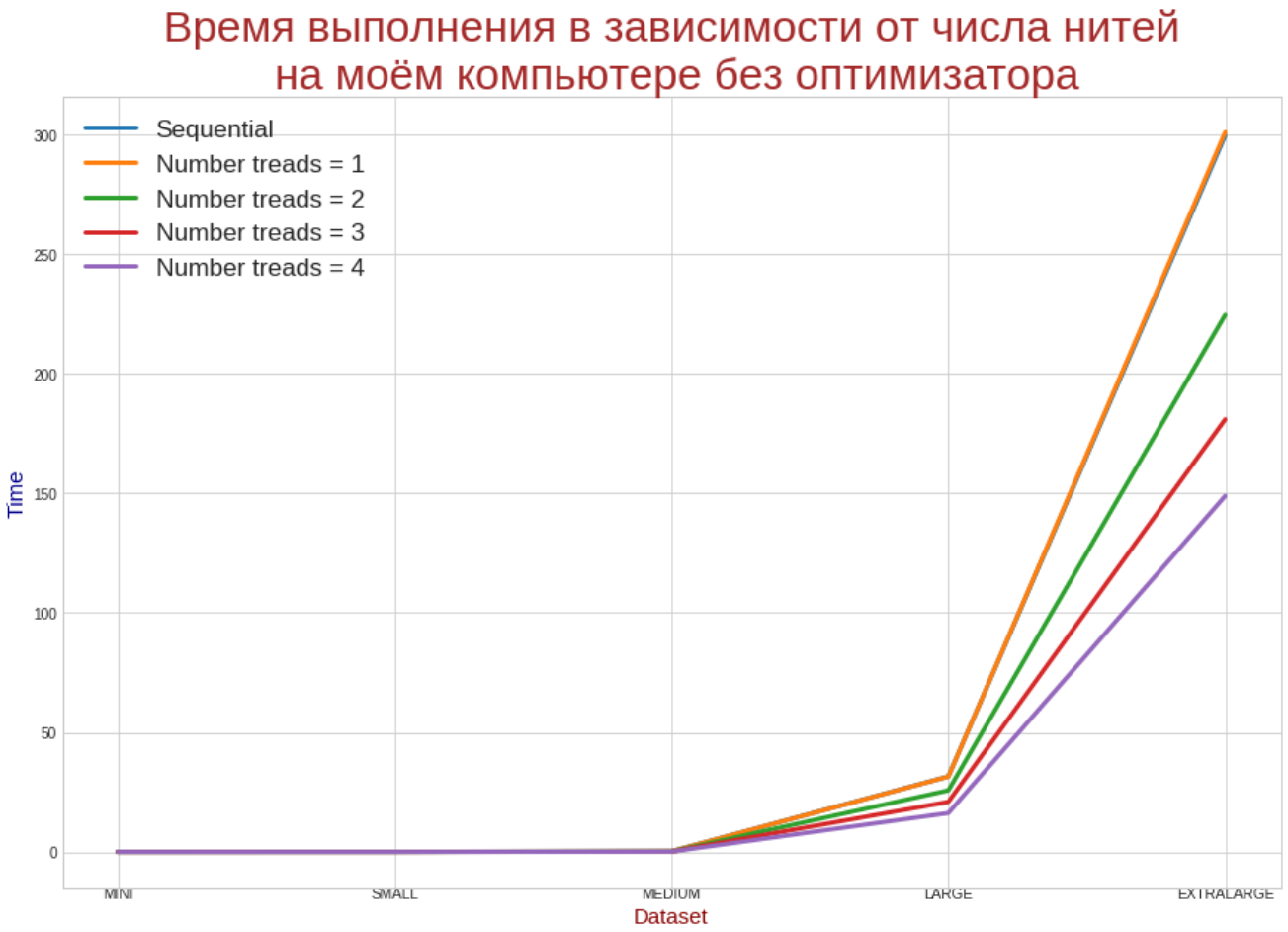
Из приведенных данных видно, что время выполнения программы уменьшается либо находится примерно на том же уровне, что и оптимальное с увеличением числа потоков, а оптимальное число потоков
равно 2-4.

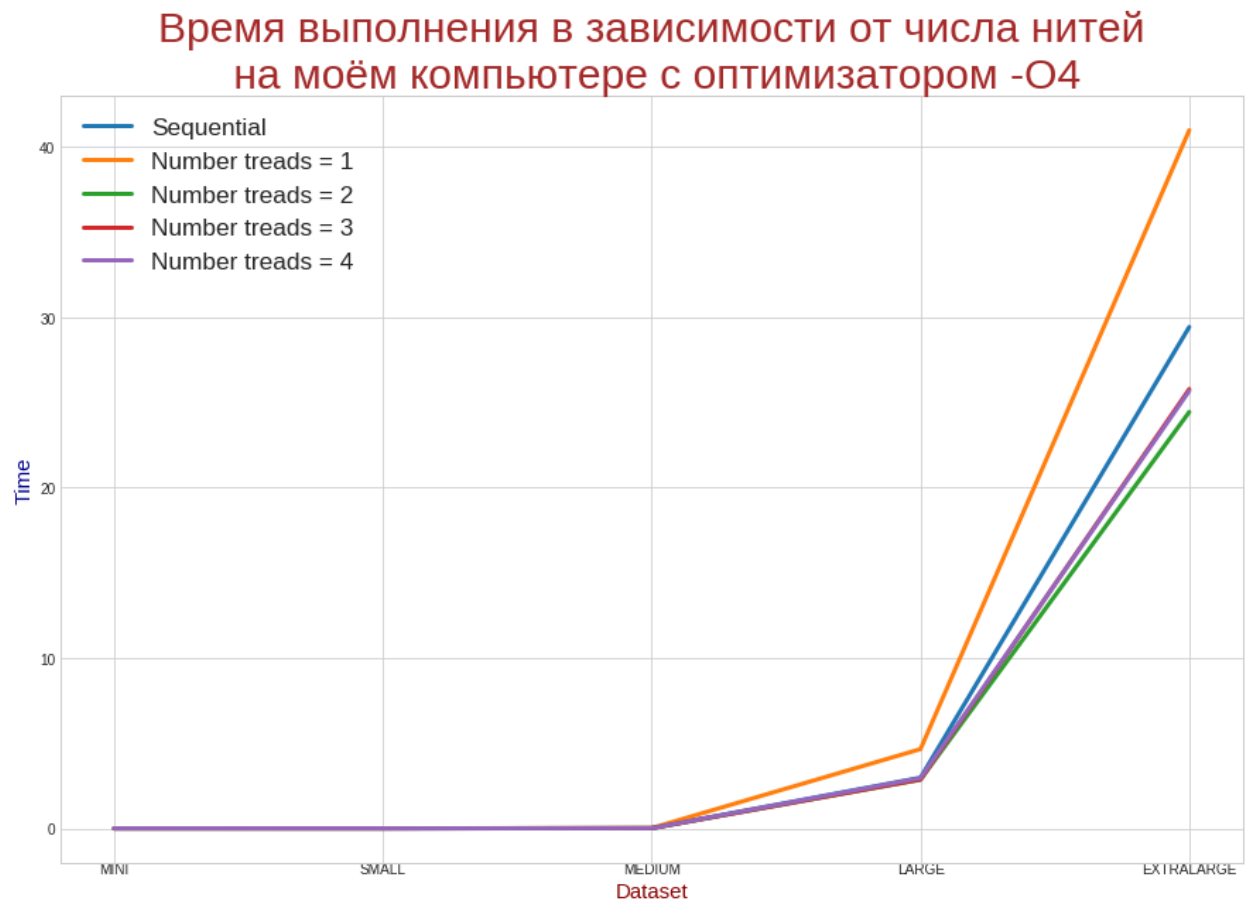
Анализ и сравнение результатов

Предоставим также 2D графики для каждой вычислительной машины, график сравнения вычислительных машин с друг другом, а также сравнение времени выполнения параллельной программы на личном ПК с и без оптимизатора *-O4*.

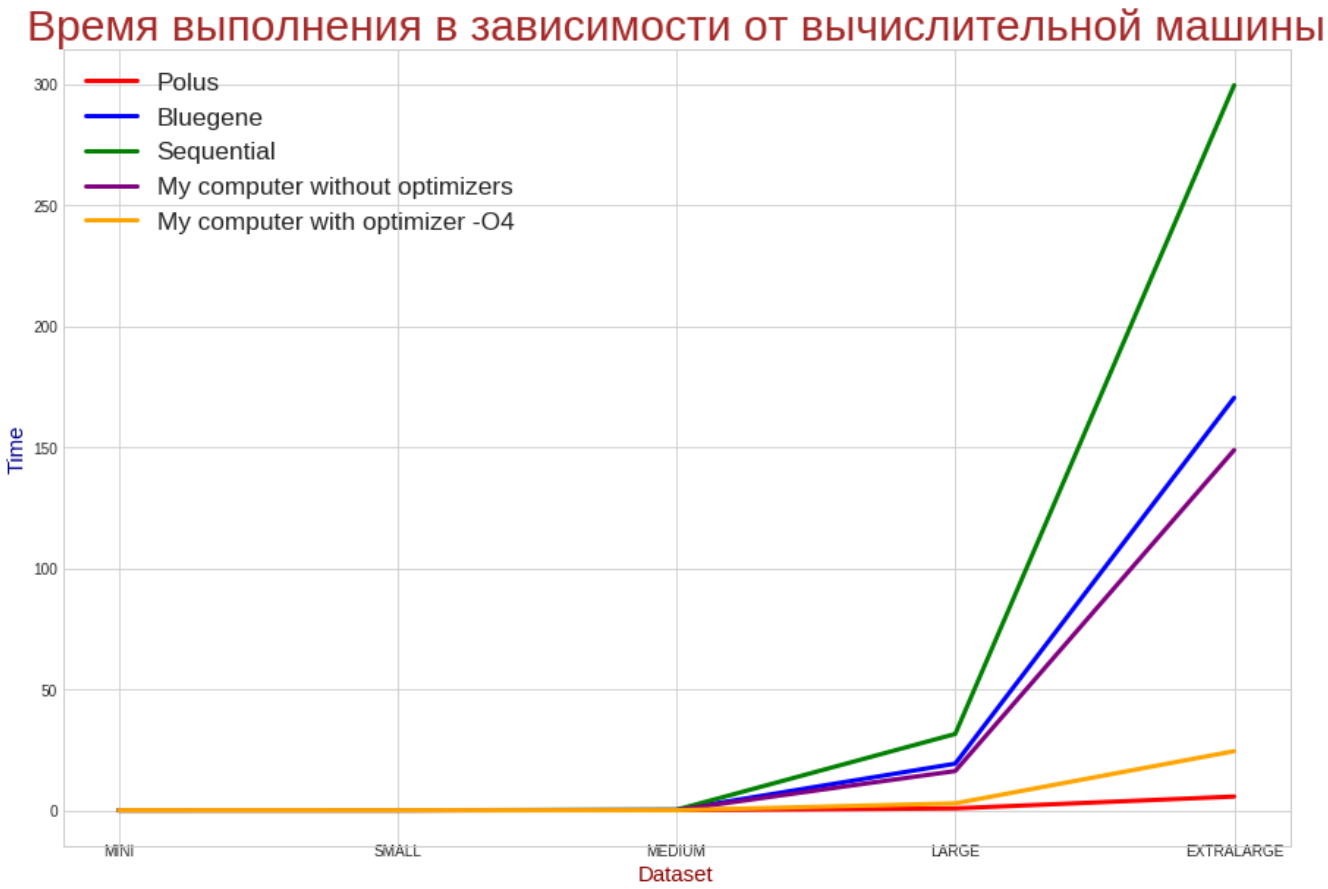



Время выполнения в зависимости от числа нитей на Polus
Sequential
Number treads = 1
Number treads = 2
Number treads = 4
Number treads = 8
Number treads = 16
Number treads = 32
Number treads = 64
Number treads = 100
Number treads = 128
Number treads = 160
Time
0
10
20
30
40
50
MINI
SMALL
MEDIUM
LARGE
EXTRALARGE
Dataset


Время выполнения в зависимости от числа нитей
на моём компьютере без оптимизатора
Sequential
Number treads = 1
Number treads = 2
Number treads = 3
Number treads = 4
Time
0
50
100
150
200
250
300
MINI
SMALL
MEDIUM
LARGE
EXTRALARGE
Dataset



И наконец, график сравнения вычислительных машин друг с другом:



В последнем графике для каждой вычислительной машины представлены данные для оптимального для них числа потоков – вычисленного ранее. Из этого графика видно, что *Polus* на 100 потоках показывает наилучший результат (почему именно на 100 было объяснено ранее). Следующим за *Polus* идёт мой ПК с наличием и отсутствием оптимизатора *-O4*. Примечательно, что оптимизатор *-O4* дал прирост примерно в 7 раз. Однако личный ПК даже без оптимизатора показал результаты лучше, чем один узел *Bluegene*. Тем не менее, все из представленных вычислительных машин в несколько раз сократили время выполнения по сравнению с временем выполнения последовательной программы на соответствующей системе.

По итогу, *Bluegene* и личный ПК показали линейный рост скорости выполнения программы при увеличении используемых потоков. 100 потоков – оптимальное кол-во для *Polus*, далее при увеличении их числа не происходит увеличения скорости выполнения, что объясняется выбранным параллельным алгоритмом (более подробно см. выше).

В некоторых случаях наблюдалось стагнация или даже увеличение времени выполнения программы с ростом числа потоков (чаще всего на небольших датасетах), что можно объяснить тем, что накладные расходы на создание дополнительных потоков перевешивали “доходы” от использования потоков. Это одна из основных причин недостаточной масштабируемости программы.

Блочный алгоритм показал хорошие результаты при проведении экспериментов. Также было выявлено, что порой полезно писать свою версию параллельной программы для каждого числа нитей (по крайней мере учитывать количество потоков).

В случае, если заранее предсказать оптимальное кол-во нитей для алгоритма не удаётся можно провести тестирование подобно представленному здесь и, исходя их полученных таблиц и графиков сделать соответствующий вывод. То есть с одной стороны можно строить алгоритм опираясь на предоставленное число потоков для распараллеливания, а с другой стороны для каждого из

построенных алгоритмов эмпирическим путем можно находить оптимальное число нитей.

Выводы

В данной практической работе я разобрался с технологией *OpenMP*, а также разработал с её помощью распараллеленные версии программы для задачи "Уравнение теплопроводности в трехмерном пространстве" ("Heat equation over 3D data domain").

Была исследована эффективность и масштабируемость полученной программы.

Результаты тестов показывают, что данная задача хорошо поддаётся распараллеливанию.

Были построены графики зависимости времени исполнения от числа ядер/процессоров для различного объёма входных данных.

Для каждого набора входных данных было найдено количество ядер/процессоров, при котором время выполнения задачи перестаёт уменьшаться, а также определенны основные причины недостаточной масштабируемости программы при максимальном числе используемых ядер/процессоров.

В итоге можно сделать вывод, что с помощью распараллеливание программы можно существенно уменьшить время ее выполнения. Особенно это заметно на больших объемах данных. В случае небольших объемах данных прирост наблюдается небольшой, поэтому порой даже выгоднее производить вычисления с помощью последовательной версии алгоритма.

OpenMP показала себя как технология, позволяющая довольно быстро и качественно получить значительный прирост скорости выполнения программ.

Используя лишь один узел суперкомпьютера, мы увеличили производительность программы с 300 секунд выполнения последовательной версии на личном ПК до 5 секунд выполнения параллельной версии на Polus с

100 нитями. Полученный результат вдохновляет и демонстрирует силу суперкомпьютеров и технологий параллельной обработки данных.

Приложение

Вместе с отчетом прикладываю файлы:

- Изначальная, последовательная программа: *heat-3d.c*, *heat-3d.h*
- Параллельная версия программы: *heat-3d_parallel.c*
- Ноутбук с построением графиков: *graph.ipynb*
- Файлы с результатами тестирования программы: *result_blugene.txt*, *result_polus.txt*, *result_my_computer.txt*