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"

ОТЧЕТ

о выполненном задании
студента 325 учебной группы факультета ВМК МГУ

Жигалова Никиты Сергеевича

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

Москва, 2024 г

Постановка задачи

1. Реализовать две параллельных версии программы для задачи "Уравнение теплопроводности в трехмерном пространстве" с помощью технологий параллельного программирования *OpenMP*.
    a. Вариант параллельной программы с распределением витков циклов при помощи директивы **for**.
    b. Вариант параллельной программы с использованием механизма задач (директива **task**).
2. Убедиться в корректности разработанных версий программ и в их эффективности.
3. Подобрать начальные параметры так, чтобы:
    a. Задача помещалась в оперативную память одного узла кластера.
    b. Время решения задачи было в примерном диапазоне 5 сек.-15 минут.
4. Исследовать эффективность полученных программ на суперкомпьютере Polus.
    a. Также сравнить с исходной версией программы
    b. Исследовать влияние опций оптимизации, поддерживаемых компиляторами (-O2, -O3, -fast...)
5. Исследовать масштабируемость полученной программы.
6. Построить графики зависимости времени исполнения от числа ядер/процессоров для различного объёма входных данных для каждого из разработанных вариантов программы.
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() //считаем время старта
   void bench_timer_stop() // считаем время финиша
   static double rtclock() // функция подсчета времени
   void bench_timer_print() // вывод времени выполнения программы
   static void init_array (int n, float A[n][n][n], float B[n][n][n])
   //инициализируем массивы A и B
   static void print_array(int n, float A[n][n][n]) - печать массива
   static void kernel_heat_3d(int tsteps, int n, float A[n][n][n], float
   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:

```
64  static
65  void kernel_heat_3d(int tsteps,
66          int n,
67          float A[ n][n][n],
68          float B[ n][n][n])
69  {
70      int t, i, j, k;
71
72      for (t = 1; t <= TSTEPS; t++) {
73          for (i = 1; i < n-1; i++) {
74              for (j = 1; j < n-1; j++) {
75                  for (k = 1; k < n-1; k++) {
76                      B[i][j][k] = 0.125f * (A[i+1][j][k] - 2.0f * A[i][j][k] + A[i-1][j][k])
77                                  + 0.125f * (A[i][j+1][k] - 2.0f * A[i][j][k] + A[i][j-1][k])
78                                  + 0.125f * (A[i][j][k+1] - 2.0f * A[i][j][k] + A[i][j][k-1])
79                                  + A[i][j][k];
80                  }
81              }
82          }
83          for (i = 1; i < n-1; i++) {
84              for (j = 1; j < n-1; j++) {
85                  for (k = 1; k < n-1; k++) {
86                      A[i][j][k] = 0.125f * (B[i+1][j][k] - 2.0f * B[i][j][k] + B[i-1][j][k])
87                                  + 0.125f * (B[i][j+1][k] - 2.0f * B[i][j][k] + B[i][j-1][k])
88                                  + 0.125f * (B[i][j][k+1] - 2.0f * B[i][j][k] + B[i][j][k-1])
89                                  + B[i][j][k];
90                  }
91              }
92          }
93      }
94  }
```

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 float. 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:

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

Предпринятые модификации к последовательной программе

В исходной программе были некоторые проблемы и неточности, вот некоторые исправления:

1) В основной функции программы - `static void kernel_heat_3d(int tsteps, int n, float A[n][n][n], float B[n][n][n])` в фрагменте `for (t = 1; t <= TSTEPS; t++) {` … вместо TSTEPS будем использовать tsteps, иначе теряется смысл передачи этого параметра в аргументы функции.
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()`. Сами функции в итоговой программе будут отсутствовать.

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

Было принято решение задавать датасеты в цикле, перебирая работу программы на всех, после чего результат усредняется по пяти запускам

4) Также лишний фрагмент с выводом массива будет удален (функция print_array) (нам не важно содержимое массива, мы больше хотим замерить скорость работы)

Построение параллельной версии программы с помощью директивы parallel for

1) В функции `static void kernel_heat_3d(int tsteps, int n, float A[n][n][n], float B[n][n][n])` производится множество операций при релаксации массива A и B. Заметим, что вычисления типа `2.0f * A[i][j][k]`, умножения на `0.125f` и другие выполняются несколько раз. Приведем в них 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] = 0.125f * (A[i + 1][j][k] - 2.0f * A[i][j][k] + A[i-1][j][k])
           + 0.125f * (A[i][j + 1][k] - 2.0f * A[i][j][k] + A[i][j-1][k])
           + 0.125f * (A[i][j][k + 1] - 2.0f * A[i][j][k] + A[i][j][k-1])
           + A[i][j][k];
```

```
B[i][j][k] = A[i][j][k] * 0.25f;
B[i][j][k] += 0.125f * (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]);
```

В итоге вместо 15 операций в цикле мы выполняем 8.

2) В функции `static void kernel_heat_3d(int tsteps, int n, float A[n][n][n], float 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:

```
79  static void
80  kernel_heat_3d_parallel_base(int tsteps, int n, float A[n][n][n],
81                               float B[n][n][n]) // основная функция работы с массивами
82  {
83      int t, i, j, k;
84
85      for (t = 1; t <= tsteps; t++) {
86          #pragma omp parallel for private(i, j, k)
87          for (i = 1; i < n - 1; i++) {
88              for (j = 1; j < n - 1; j++) {
89                  for (k = 1; k < n - 1; k++) {
90                      B[i][j][k] = 0.25f * A[i][j][k];
91                      B[i][j][k] += 0.125f * (A[i + 1][j][k] + A[i - 1][j][k] + A[i][j + 1][k] + A[i][j - 1][k] +
92                                              A[i][j][k + 1] + A[i][j][k - 1]);
93                  }
94              }
95          }
96          for (i = 1; i < n - 1; i++) {
97              for (j = 1; j < n - 1; j++) {
98                  for (k = 1; k < n - 1; k++) {
99                      A[i][j][k] = 0.25f * B[i][j][k];
100                     A[i][j][k] += 0.125f * (B[i + 1][j][k] + B[i - 1][j][k] + B[i][j + 1][k] + B[i][j - 1][k] +
101                                             B[i][j][k + 1] + B[i][j][k - 1]);
102                 }
103             }
104         }
105     }
106 }
```

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 к сближению обращений к ячейкам памяти, при этом производительность повышается за счет кэширования. На практике при тестировании эта манипуляция действительно

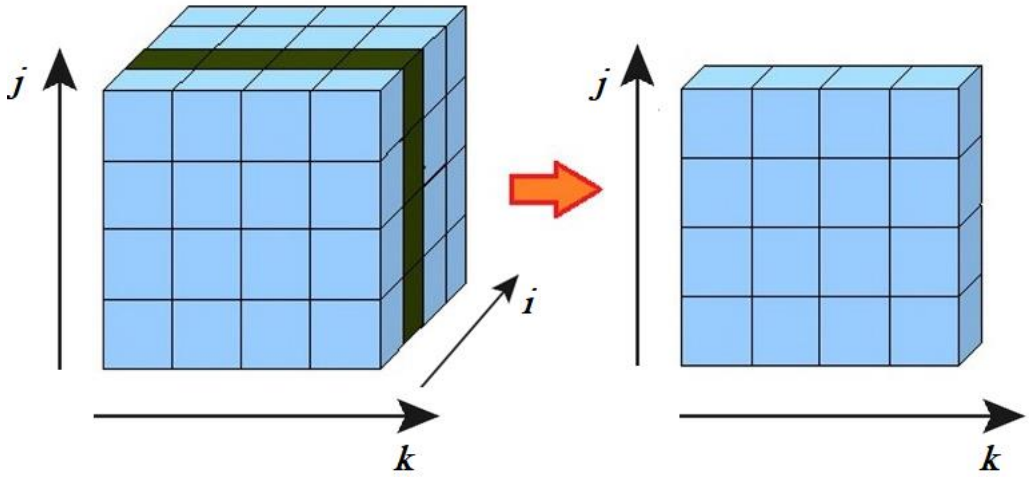
увеличивает скорость программы по сравнению с базовым параллельным алгоритмом. Получим функцию ***kernel_heat_3d_parallel_mini***:

Улучшенная версия параллельной программы для небольших датасетов:

```
static void
kernel_heat_3d_parallel_mini(int tsteps, int n, float A[n][n][n],
                             float B[n][n][n]) // основная функция работы с массивами
{
    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) {
                    B[i][j][k] = 0.25f * A[i][j][k];
                    B[i][j][k] += 0.125f * (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]);

                    B[i][j][k + 1] = 0.25f * A[i][j][k + 1];
                    B[i][j][k + 1] += 0.125f * (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]);

                    B[i][j][k + 2] = 0.25f * A[i][j][k + 2];
                    B[i][j][k + 2] += 0.125f * (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]);

                    B[i][j][k + 3] = 0.25f * A[i][j][k + 3];
                    B[i][j][k + 3] += 0.125f * (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]);
                }
            }
        }
        #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++) {
                    A[i][j][k] = 0.25f * B[i][j][k];
                    A[i][j][k] += 0.125f * (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]);

                    A[i][j][k + 1] = 0.25f * B[i][j][k + 1];
                    A[i][j][k + 1] += 0.125f * (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]);

                    A[i][j][k + 2] = 0.25f * B[i][j][k + 2];
                    A[i][j][k + 2] += 0.125f * (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]);

                    A[i][j][k + 3] = 0.25f * B[i][j][k + 3];
                    A[i][j][k + 3] += 0.125f * (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]);
                }
            }
        }
```

5) Для больших датасетов, ***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 ***i*** задаёт плоскости, то в данном алгоритме будем разбивать

каждую плоскость с измерениями ***j***, ***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:

```
162  static void
163  kernel_heat_3d_parallel_norm(int tsteps, int n, float A[n][n][n], float B[n][n][n], int block_size)
164  {
165      int t, i, j, k, kk, jj;
166      for (t = 1; t <= tsteps; t++) {
167          #pragma omp parallel for private(i, j, k, kk, jj)
168          for (jj = 1; jj < n - 1; jj += block_size) {
169              for (i = 1; i < n - 1; i++) {
170                  for (kk = 1; kk < n - 1; kk += block_size) {
171                      for (j = jj; j < MIN(n - 1, jj + block_size); j++) {
172                          for (k = kk; k < MIN(n - 1, kk + block_size); k++) {
173                              B[i][j][k] = A[i][j][k] * 0.25f;
174                              B[i][j][k] = 0.125f * (A[i + 1][j][k] + A[i - 1][j][k] + A[i][j + 1][k] + A[i][j - 1][k] +
175                                                     A[i][j][k + 1] + A[i][j][k - 1]);
176                          }
177                      }
178                  }
179              }
180          }
181
182          #pragma omp parallel for private(i, j, k, kk, jj)
183          for (jj = 1; jj < n - 1; jj += block_size) {
184              for (i = 1; i < n - 1; i++) {
185                  for (kk = 1; kk < n - 1; kk += block_size) {
186                      for (j = jj; j < MIN(n - 1, jj + block_size); j++) {
187                          for (k = kk; k < MIN(n - 1, kk + block_size); k++) {
188                              A[i][j][k] = B[i][j][k] * 0.25f;
189                              A[i][j][k] = 0.125f * (B[i + 1][j][k] + B[i - 1][j][k] + B[i][j + 1][k] + B[i][j - 1][k] +
190                                                     B[i][j][k + 1] + B[i][j][k - 1]);
191                          }
192                      }
193                  }
194              }
195          }
196      }
197  }
```

Параллельная для больших датасетов при сравнительно большом числе нитей:

```
199  static void
200  kernel_heat_3d_parallel_big(int tsteps, int n, float A[n][n][n], float B[n][n][n], int block_size)
201  {
202      int t, i, j, k, kk, jj;
203      for (t = 1; t <= tsteps; t++) {
204  #pragma omp parallel for private(i, j, k, kk, jj)
205          for (i = 1; i < n - 1; i++) { // тут нитей больше значит можем параллелить внений цикл
206              for (jj = 1; jj < n - 1; jj += block_size) {
207                  for (kk = 1; kk < n - 1; kk += block_size) {
208                      for (j = jj; j < MIN(n - 1, jj + block_size); j++) {
209                          for (k = kk; k < MIN(n - 1, kk + block_size); k++) {
210                              B[i][j][k] = A[i][j][k] * 0.25f;
211                              B[i][j][k] = 0.125f * (A[i + 1][j][k] + A[i - 1][j][k] + A[i][j + 1][k] + A[i][j - 1][k] +
212                                                     A[i][j][k + 1] + A[i][j][k - 1]);
213                          }
214                      }
215                  }
216              }
217          }
218
219  #pragma omp parallel for private(i, j, k, kk, jj)
220          for (i = 1; i < n - 1; i++) {
221              for (jj = 1; jj < n - 1; jj += block_size) {
222                  for (kk = 1; kk < n - 1; kk += block_size) {
223                      for (j = jj; j < MIN(n - 1, jj + block_size); j++) {
224                          for (k = kk; k < MIN(n - 1, kk + block_size); k++) {
225                              A[i][j][k] = B[i][j][k] * 0.25f;
226                              A[i][j][k] = 0.125f * (B[i + 1][j][k] + B[i - 1][j][k] + B[i][j + 1][k] + B[i][j - 1][k] +
227                                                     B[i][j][k + 1] + B[i][j][k - 1]);
228                          }
229                      }
230                  }
231              }
232          }
233      }
234  }
```

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 = 200. 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 в зависимости от размера датасета и числа нитей в распоряжении вычислительной машины вызывает соответствующую параллельную функцию.

Построение параллельной программы с использованием директивы task

Далее организовываем работу через директиву task. Здесь было принято решение использовать одну функцию, аналогичную base.

Для этого внешние циклы по **t** и по **i** заключаем в директивы `#pragma omp parallel` и `#pragma omp single`, так как внешний цикл по **t** у нас нельзя распараллелить (как было сказано выше он отвечает за итерации вычисления массивов, а их нельзя смешивать), а внутри цикла по **i** мы уже будем вызывать процессы обрабатывающие параллельно вложенные циклы.

Внутри самих task мы используем общее переменные A, B, n, а остальные мы записываем через private и firstprivate.

Как будет показано далее это даст значительный прирост в скорости выполнения, относительно исходной программы, для больших размеров массивов (особенно на ExtraLarge датасете)

```
58  static void
59  kernel_heat_3d_parallel_task(int tsteps, int n, float A[n][n][n],
60                               float B[n][n][n]) // основная функция работы с массивами
61  {
62
63      int t, i, j, k;
64  #pragma omp parallel
65      {
66  #pragma omp single
67          {
68              for (t = 1; t <= tsteps; t++) {
69                  for (i = 1; i < n - 1; i++) {
70  #pragma omp task shared(A, B, n) firstprivate(t, i) private(j, k)
71                      for (j = 1; j < n - 1; j++) {
72                          for (k = 1; k < n - 1; k++) {
73                              B[i][j][k] = 0.25f * A[i][j][k];
74                              B[i][j][k] += 0.125f * (A[i + 1][j][k] + A[i - 1][j][k] + A[i][j + 1][k] + A[i][j - 1][k] +
75                                                     A[i][j][k + 1] + A[i][j][k - 1]);
76                          }
77                      }
78                  }
79  #pragma omp taskwait
80
81                  for (i = 1; i < n - 1; i++) {
82  #pragma omp task shared(A, B, n) firstprivate(t, i) private(j, k)
83                      for (j = 1; j < n - 1; j++) {
84                          for (k = 1; k < n - 1; k++) {
85                              A[i][j][k] = 0.25f * B[i][j][k];
86                              A[i][j][k] += 0.125f * (B[i + 1][j][k] + B[i - 1][j][k] + B[i][j + 1][k] + B[i][j - 1][k] +
87                                                     B[i][j][k + 1] + B[i][j][k - 1]);
88                          }
89                      }
90                  }
91  #pragma omp taskwait
92              }
93          }
94      }
95  }
```

Также на выходе цикла по **`i`** устанавливаем `#pragma omp taskwait`, чтобы разграничить выполнение двух циклов, а также чтобы не возникало Segmentation Error.

Построение параллельной программы с помощью MPI (Message Passing Interface)

Далее производилось построение параллельной версии программы с помощью MPI. Для этого было принято решение в основной функции произвести разбиение цикла по **i** и далее с помощью основного процесса (с rank == 0) пересылать остальным полосы из массивов A и B ширины **i,** с помощью комманд Send и Recv.

```
static void
kernel_heat_3d_MPI(int tsteps, int n, float A[n][n][n],
                   float B[n][n][n], int rank, int block_size, int size)
{
    int t, i, j, k;
    int last_block_size = n - block_size * (size-2);
    float* line = (float *)calloc((block_size+1) * n * n, sizeof(float)); // полоса из массива
    for (int t = 1; t <= tsteps; t++) {
        if (!rank) {
            for (int num = 1; num < size-1; ++num) {
                // отправляем процессу с номером num соответствующую полосу из A
                MPI_Send(A[MAX((num-1) * block_size - 1, 0)], (block_size + 1) * n * n, MPI_FLOAT, num, 42, MPI_COMM_WORLD);
            }
            MPI_Send(A[(size-2) * block_size], (last_block_size) * n * n, MPI_FLOAT, size-1, 41, MPI_COMM_WORLD);
        } else if (rank == size - 1) {
            MPI_Recv(line, (last_block_size) * n * n, MPI_FLOAT, 0, 41, MPI_COMM_WORLD, NULL); // полчаем полосу
        } else {
            MPI_Recv(line, (block_size+1) * n * n, MPI_FLOAT, 0, 42, MPI_COMM_WORLD, NULL);
        }
        MPI_Barrier(MPI_COMM_WORLD);

        if (rank != 0) {
            int block_start = MAX((rank - 1) * block_size, 1);
            int block_end = MIN(block_start + block_size, n - 1);
            float* block = (float *)calloc((block_end - block_start) * n * n, sizeof(float));
            for (i = block_start; i < block_end; i++) {
                for (j = 1; j < n - 1; j++) {
                    for (k = 1; k < n - 1; k++) {
                        block[n * n * (i - block_start) + n * j + k] = 0.25f * line[IDX(i-block_start, j, k, n)];
                        block[n * n * (i - block_start) + n * j + k] += 0.125f * (line[IDX(i-block_start+1, j, k, n)] + line[IDX(i-block_start-1, j, k, n)]
                                                                                + line[IDX(i-block_start, j+1, k, n)] + line[IDX(i-block_start, j-1, k, n)] +
                                                                                line[IDX(i-block_start, j, k+1, n)] + line[IDX(i-block_start, j, k-1, n)]);
                    }
                }
            }
            if (rank < size-1) {
                MPI_Send(block, (block_end - block_start) * n * n, MPI_FLOAT, 0, 3, MPI_COMM_WORLD);
            } else {
                MPI_Send(block, (block_end - block_start) * n * n, MPI_FLOAT, 0, 4, MPI_COMM_WORLD);
            }
            free(block);
```

```
        } else {
            for (int num = 1; num < size-1; ++num) {
                MPI_Recv(B[block_size*(num-1)], block_size * n * n, MPI_FLOAT, num, 3, MPI_COMM_WORLD, NULL);
            }
            // тут учитываем, что последний процесс может иметь не полный размер
            MPI_Recv(B[block_size*(size-2)], (last_block_size) * n * n, MPI_FLOAT, size-1, 4, MPI_COMM_WORLD, NULL);
        }
        MPI_Barrier(MPI_COMM_WORLD);
```

После идет полностью аналогичный блок, в котором мы уже получаем данные из B и обновляем A.

Подробнее, суть моего метода заключается в том, что в начале каждой итерации цикла по **t** основной процесс производил разбиение цикла по **i**, поровну деля число вычислений данного цикла между остальными процессами, которые производили вычисления. Массив отдельным процессам передавался через нулевой с помощью MPI_Send и после вычислений обновлялся через MPI_Recv.

Отдельно учитывалось, что последний процесс может выполнять меньшее число операций, чем block_size, поэтому последний процесс везде обрабатывается отдельно, а сообщения ему передаются с отдельным тегом, чтобы не возникало коллизий.

`MPI_Barrier(MPI_COMM_WORLD);` использовался для синхронизации процессов, а также чтобы каждая отдельная итерация вычисления матриц **t** выполнялась изолированно от остальных.

Также было принято решение отказаться от тестирований на датасетах MINI и SMALL, в силу не информативности результатов, и в силу того, что достаточно быстро число процессов превысит N, после чего лишние процессы никак не будут участвовать в выполнении программы.

Тестирование

Информация о тестировании

Было произведено тестирование изначальной последовательной программы и полученной параллельной программы. Тестирование производилось на суперкомпьютерные вычислительные ресурсы факультета ВМК – *Polus*.

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.

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

Polus – 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 40, 60, 80, 100, 120, 140, 160 потоков..

Замеры времени производились с помощью функции omp_get_wtime().

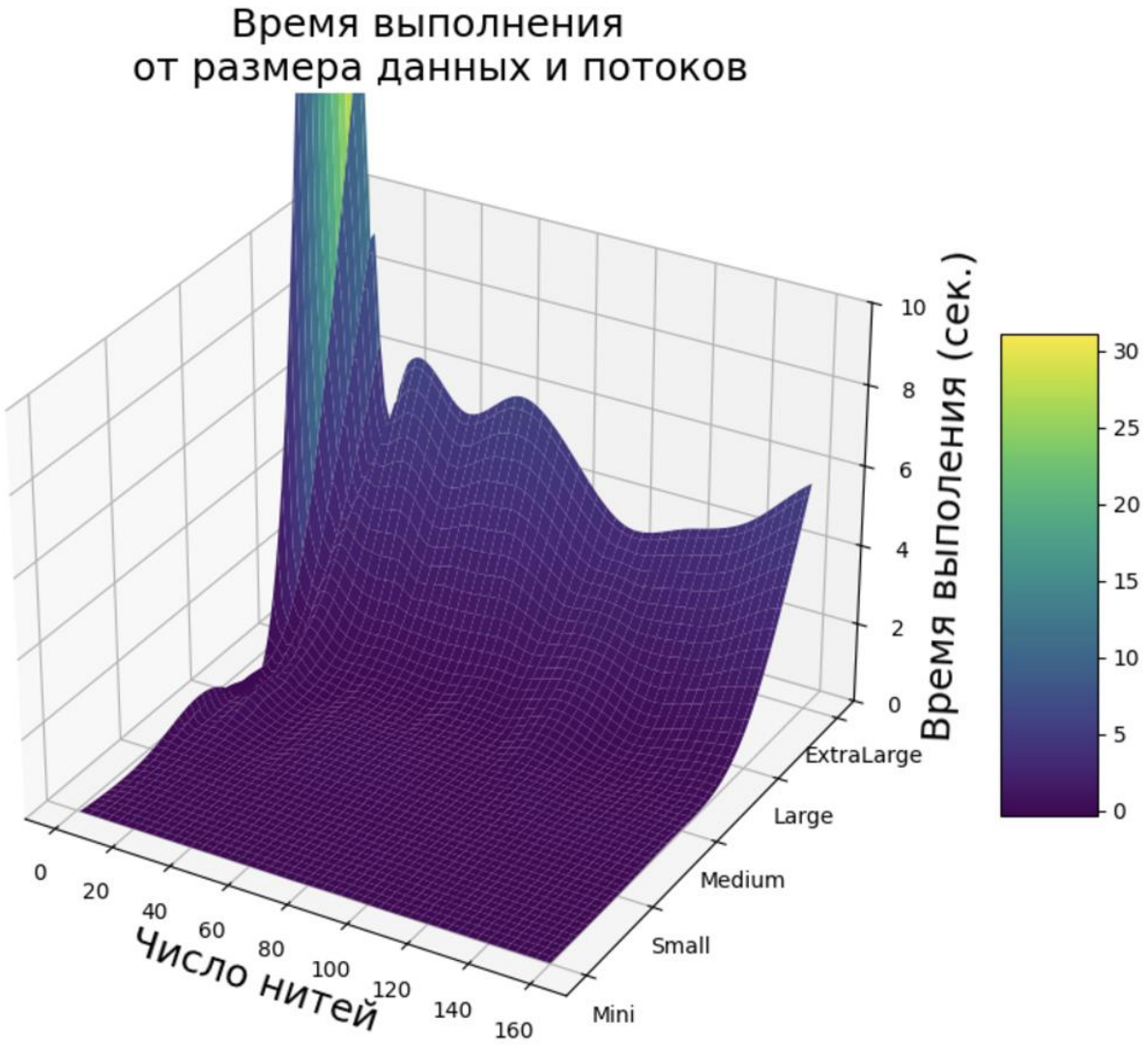
Для каждого набора входных данных (MINI, SMALL, MEDIUM, LARGE, EXTRALARGE) происходило 5 итераций подсчёта с последующим усреднением времени.

Время выполнения исходной программы					
Датасет	Mini	Small	Medium	Large	ExtraLarge
Время	0.000115	0.002398	0.054699	7.588201	75.482243

Для начала тесты проводились для прагмы parallel for, вот соответствующая сводная таблица:

Число нитей	Mini	Small	Medium	Large	ExtraLarge
1	0.000173	0.003117	0.072995	6.360267	57.765595
2	0.000167	0.001670	0.036864	3.237761	29.311453
3	0.000180	0.001171	0.025459	2.143014	20.693173
4	0.000188	0.001000	0.197410	1.617002	14.873810
5	0.000282	0.001294	0.026422	1.659031	14.112389
6	0.000205	0.000789	0.016137	1.107664	12.119614
7	0.000195	0.000794	0.012346	1.103486	9.167287
8	0.000195	0.000803	0.012291	0.848487	9.228575
9	0.000211	0.000716	0.013824	0.856777	9.110180
10	0.000272	0.000682	0.010609	1.017550	7.334358
20	0.000438	0.000704	0.009669	1.006881	5.932552
40	0.000640	0.000961	0.009750	0.663500	5.177348
60	0.000978	0.001296	0.012496	0.595216	5.936165
80	0.001186	0.001360	0.013426	0.677454	4.703110
100	0.001697	0.001806	0.024165	0.952710	3.336006
120	0.005492	0.001992	0.027607	0.460411	3.774789
140	0.009555	0.003376	0.031262	0.591688	4.381302
160	0.014271	0.003546	0.037491	0.923266	5.737227

Также вот более наглядный график по результатам работы, показывающий, что при малом числе нитей и больших объемах данных программа выполнялась значительно дольше.

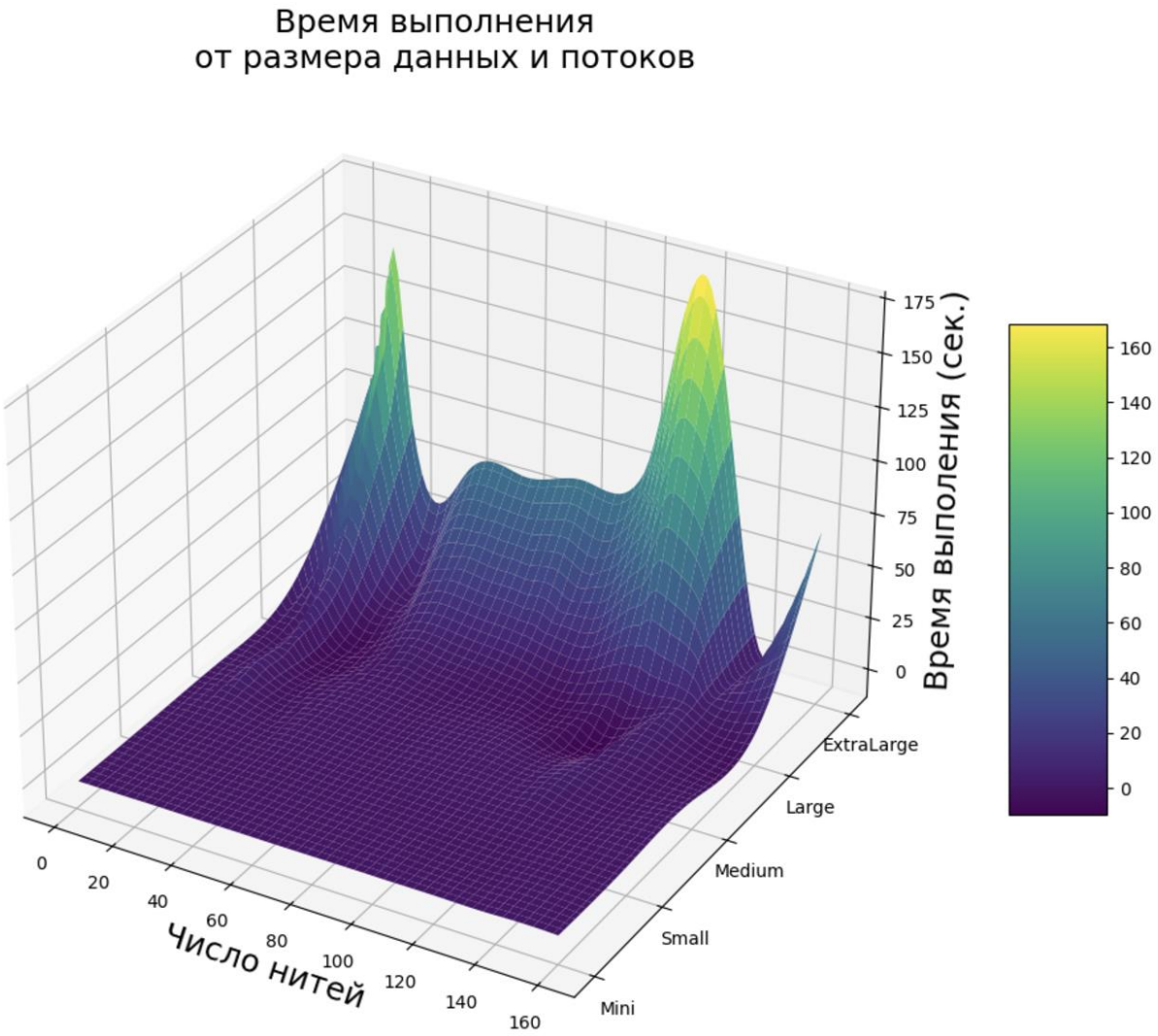


Из приведенных данных видно, что выполнение программы не всегда уменьшается с увеличением числа потоков, а оптимальное число потоков для EXTRALARGE DATASET равно 100 – далее время перестает расти, что можно объяснить тем, что в данном случае распараллеливался цикл по ***i***, в котором ***n*** = 200 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.

Теперь для механизма задач:



Видим, что график очень неравномерный – наблюдается взрывной рост на нитях – на 8 и на 120, несмотря на усреднение по 5 итерациям, хотя все же возможно, это произошло из-за нагруженности кластера, сбоев или т.п.

Таблица распределения времени выполнения:

Число нитей	Mini	Small	Medium	Large	ExtraLarge
1	0.000149	0.002904	0.070436	8.038869	61.422909
2	0.000262	0.002011	0.029170	3.138006	27.870125
3	0.000250	0.002057	0.021344	3.120079	26.229170
4	0.000463	0.002346	0.029545	4.717738	46.752775
5	0.000238	0.001925	0.028801	4.137136	38.897969
6	0.000408	0.001958	0.020458	2.591000	24.530065
7	0.000425	0.001938	0.020607	2.805597	39.332868
8	0.000744	0.004089	0.047585	9.337623	139.013408
9	0.000512	0.002122	0.035006	5.612301	54.123020
10	0.000555	0.002860	0.041952	6.578411	74.320142
20	0.000604	0.002766	0.026081	4.441720	69.831966
40	0.000849	0.002949	0.030216	4.220109	52.104172
60	0.001394	0.003824	0.036329	7.794712	56.611791
80	0.001927	0.004211	0.079822	4.912104	62.782140
100	0.002692	0.005869	0.136750	0.985115	72.632057
120	0.020284	0.016106	0.063368	7.860014	173.136570
140	0.030506	0.015082	0.052952	4.598796	7.890473
160	0.026584	0.005726	0.040315	2.195765	69.775297

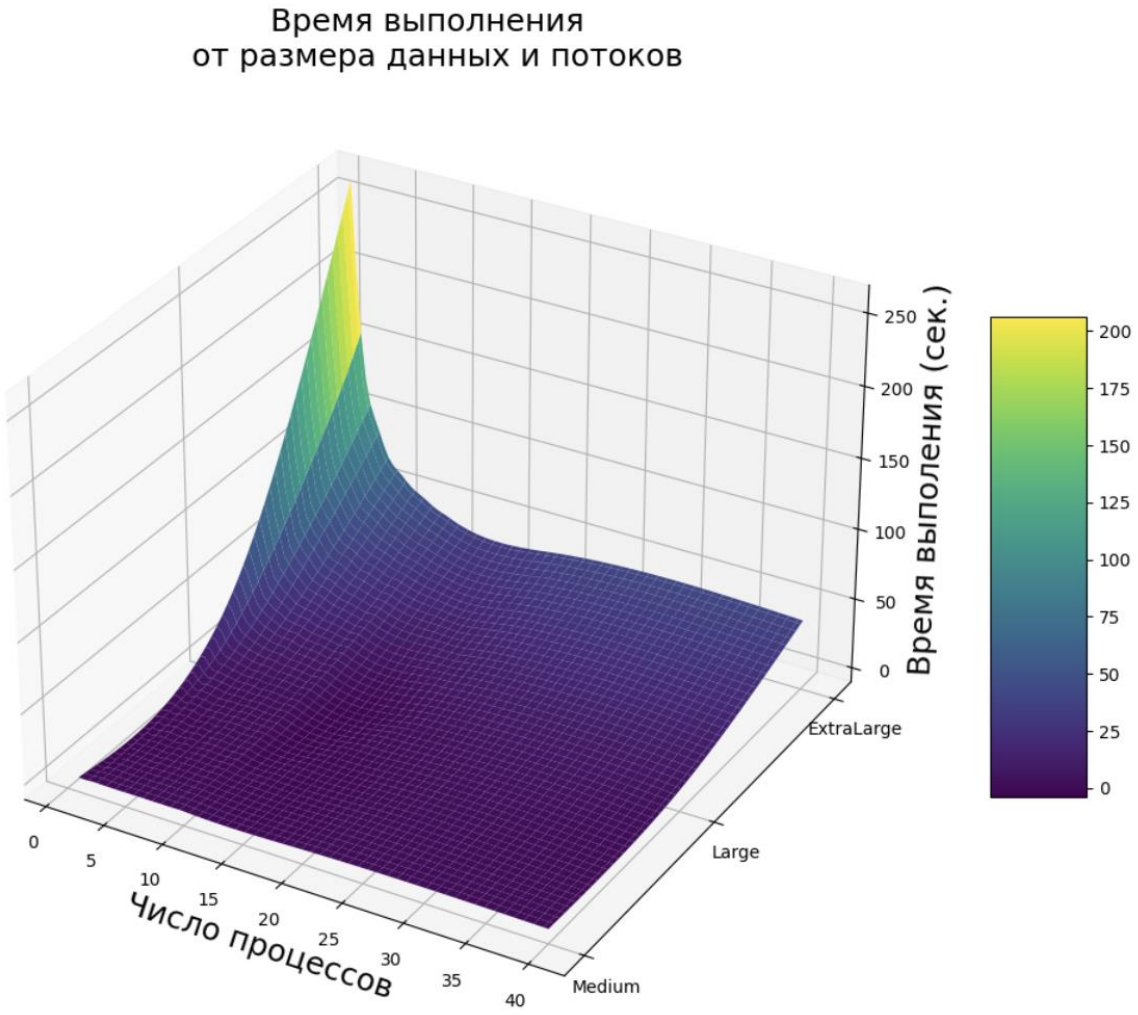
Отсюда и из графика видим, что хоть механизм задач и не дает стабильного прироста скорости, но при небольшом числе нитей (2-7) мы получаем достаточно значительное улучшение скорости работы относительно исходного кода (примерно в три раза быстрее)

MPI

Для MPI было принято решение тестировать только для датасетов размера 40, 120 и 200, т.е. только MEDIUM, LARGE, EXTRALARGE, в силу того, что бесполезно производить вычисления, если остаются свободные процессы, время выполнения будет точно расти, потому, что время выполнения самого

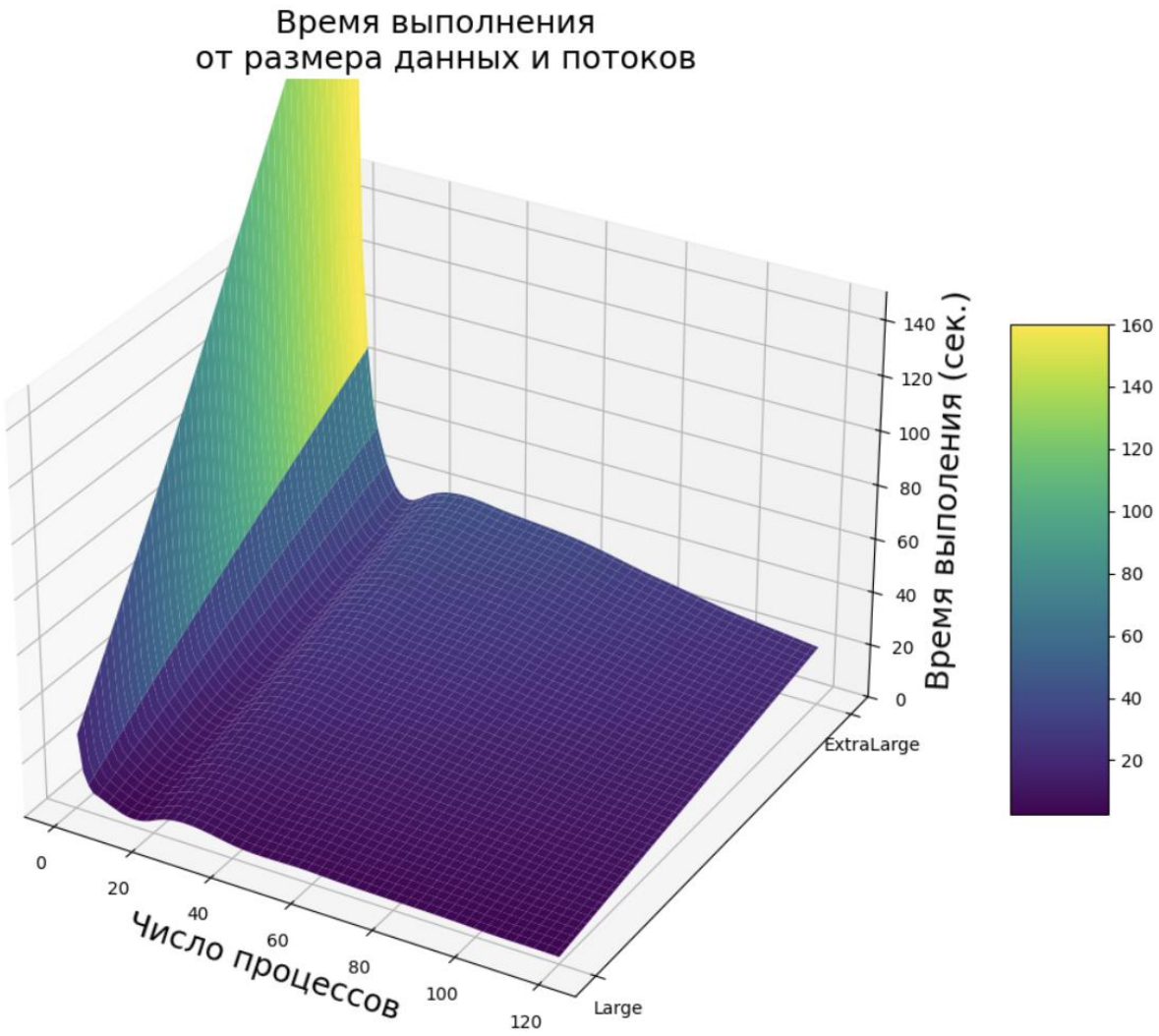
алгоритма не поменяется, а вот дополнительное время на создание и синхронизацию процессов будет вносить свой вклад.

Результаты для датасетов MEDIUM, LARGE, EXTRALARGE, при числе процессов 1-40:



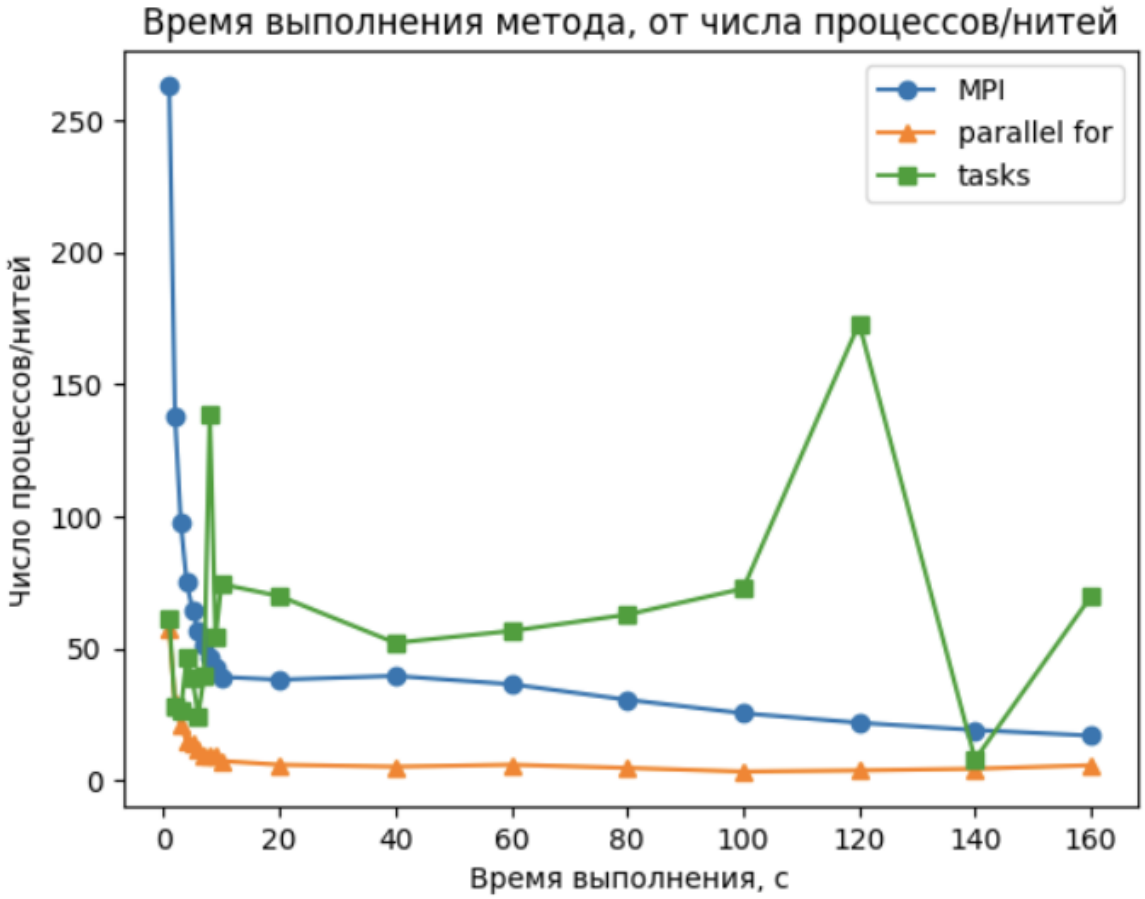
Число процессов	Medium	Large	ExtraLarge
1	0.203366	28.097266	263.352222
2	0.129331	15.292906	138.061106
3	0.103017	11.143740	97.632123
4	0.079740	8.739041	75.274362
5	0.082819	7.703834	64.623795
6	0.075843	6.795627	56.815157
7	0.073294	6.355657	51.464350
8	0.076511	5.792682	46.392596
9	0.065882	4.933087	42.601655
10	0.060450	4.525320	39.172380
20	0.078074	4.657109	38.139866
40	0.100999	5.017628	39.578404

Результаты для датасетов LARGE, EXTRALARGE, при числе процессов 1-120 (здесь уже с 40 отсутствуют запуски на MEDIUM, так как там N=40):



Число процессов	Large	ExtraLarge
1	28.097266	263.352222
2	15.292906	138.061106
3	11.143740	97.632123
4	8.739041	75.274362
5	7.703834	64.623795
6	6.795627	56.815157
7	6.355657	51.464350
8	5.792682	46.392596
9	4.933087	42.601655
10	4.525320	39.172380
20	4.657109	38.139866
40	5.017628	39.578404
60	4.351240	36.425080
80	3.621490	30.538230
100	3.017330	25.426450
120	2.592440	21.832600

Итоговый график для 1-160 процессов на датасете EXTRALARGE, с сравнением между всеми описываемыми выше методами



Как можно сделать вывод, лучше всего подошло использование прагмы parallel for, скорее всего это связано с тем, что этот метод наиболее хорошо подходит для распараллеливания четверного цикла, что мы имеем в задаче, task и MPI же имеют слишком много накладных расходов с созданием процессов/задач и их синхронизацией, из-за чего результаты для них в среднем хуже.