Домашнее задание

Задание:

1. Реализовать один из алгоритмов сортировки (сортировка Хоара) последовательно и параллельно, с использованием технологии OpenMP на языках C и Python.

2. Провести исследование реализованных алгоритмов по времени и ускорению в зависимости от числа потоков и размера массива.

Технические характеристики

Ниже приведены технические характеристики устройства, на котором были проведены эксперименты при помощи разработанного ПО:

- операционная система: Windows 10 (64-разрядная);
- оперативная память: 32 GB;
- процессор: Intel(R) Core(TM) i7-7700K CPU @ 4.20GHz;
- количество ядер: 4;
- количество потоков: 8.

Постановка экспериментов

Для измерения времени выполнения реализованных алгоритмов использовалась функция omp_get_wtime(), т.к. параллелизация производилась средствами технологии openMP. Для более точной оценки замеры проводились 100-1000 итераций в зависимости от размера массивов, а результат усреднялся.

Реализация на C

Сначала был реализован последовательный алгоритм сортировки Хоара (листинг 1). В познавательных целях были опробованы разбиение Ломуто и схема

Хоара. После этого был реализован параллельный алгоритм Хоара с использованием технологии openMP (листинг 2). Была использована стратегия "Разделяй и властвуй которая отлично ложится на алгоритм быстрой сортировки - при каждом новом рекурсивном вызове процесс разделяется на два и так, до тех пор, пока не будут использовано максимально возможно число потоков. Были опробованы варианты с omp parallel sections, omp task с условием if и просто omp task. Лучше всего показала себя последняя реализация, поэтому для сравнения с последовательным алгоритм использовалась именно она.

При первоначальной тестировании последовательного и параллельного алгоритмов получались неудовлетворительные результаты: параллельная реализация показывала себя хуже последовательной и не ускорялась с увеличением числа потоков. Но после включения оптимизации (флагом /O2 в Visual Studio) получились более корректные данные (рис. 1-2).

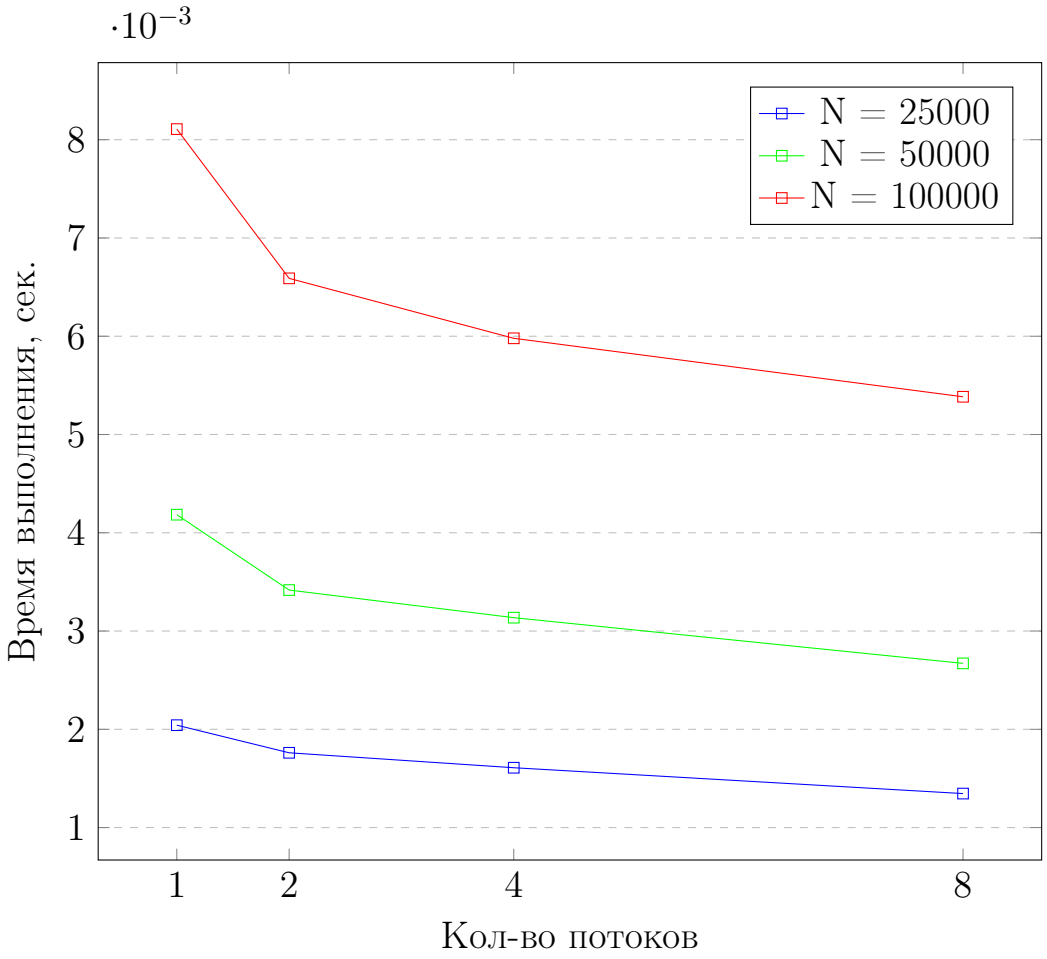

Рис. 1: Зависимость времени выполнения сортировки от количества процессов для массивов различного размера

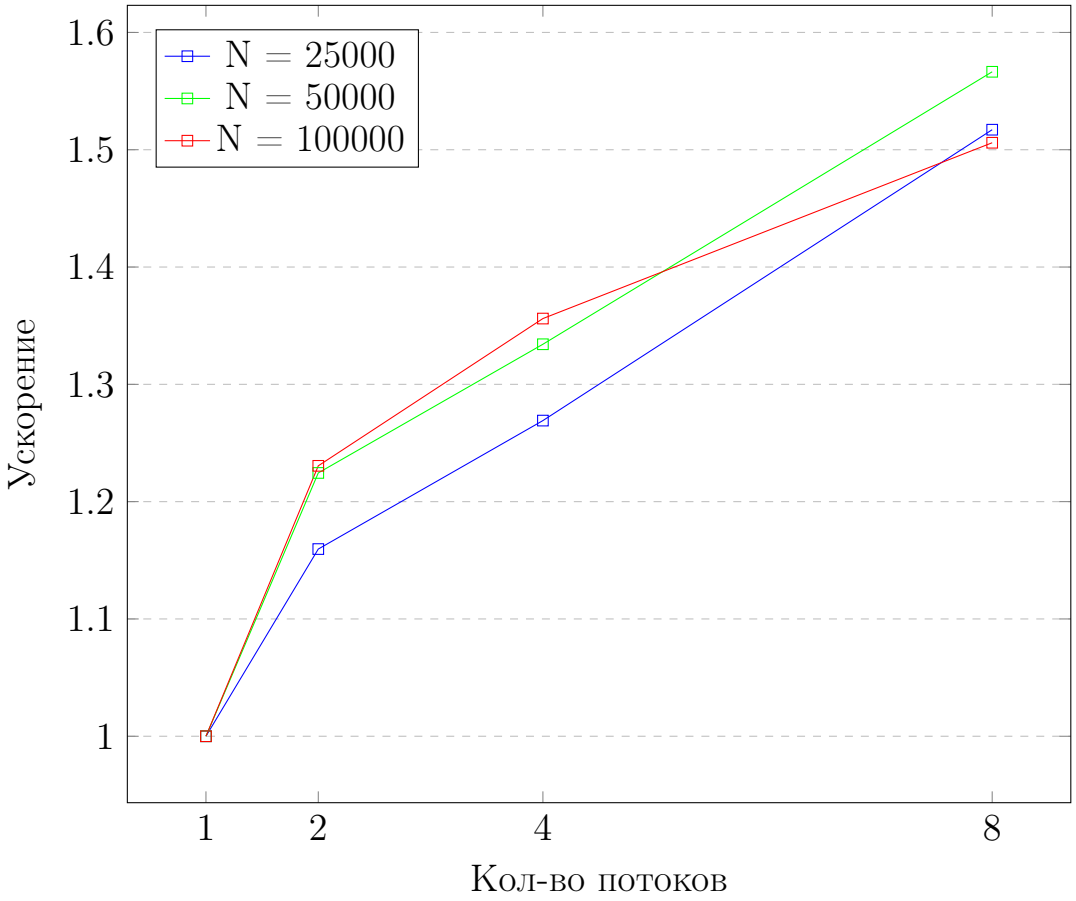

Рис. 2: Зависимость ускорения сортировки от количества процессов для массивов различного размера

Как видно из графиков 1-2 параллельная реализация работает быстрее последовательной, а ускорение для массивов различного размера примерно одинаковое, но не превышает 1.6.

Дополнительно два данных алгоритма были протестированы на суперкомпьютере Харизма (рис. 3-4).

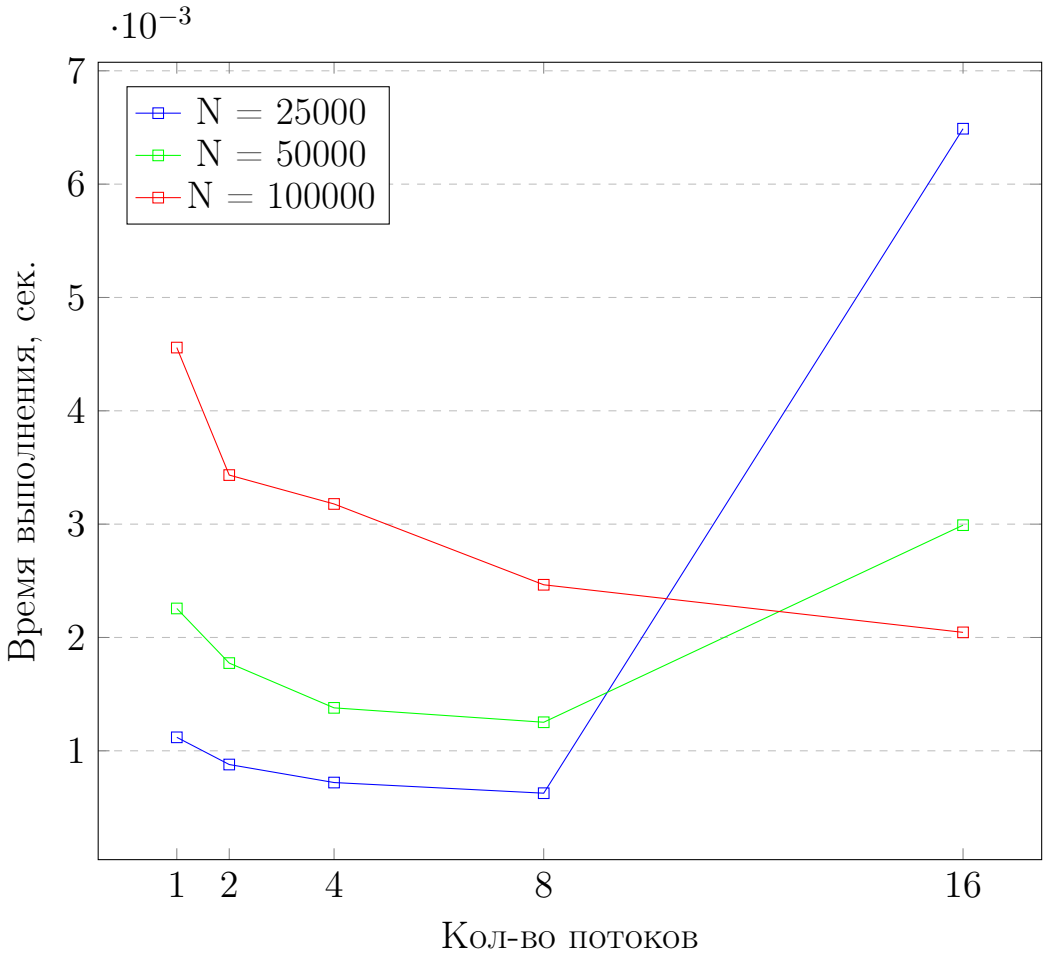

Рис. 3: Зависимость времени выполнения сортировки от количества процессов для массивов различного размера на Харизме

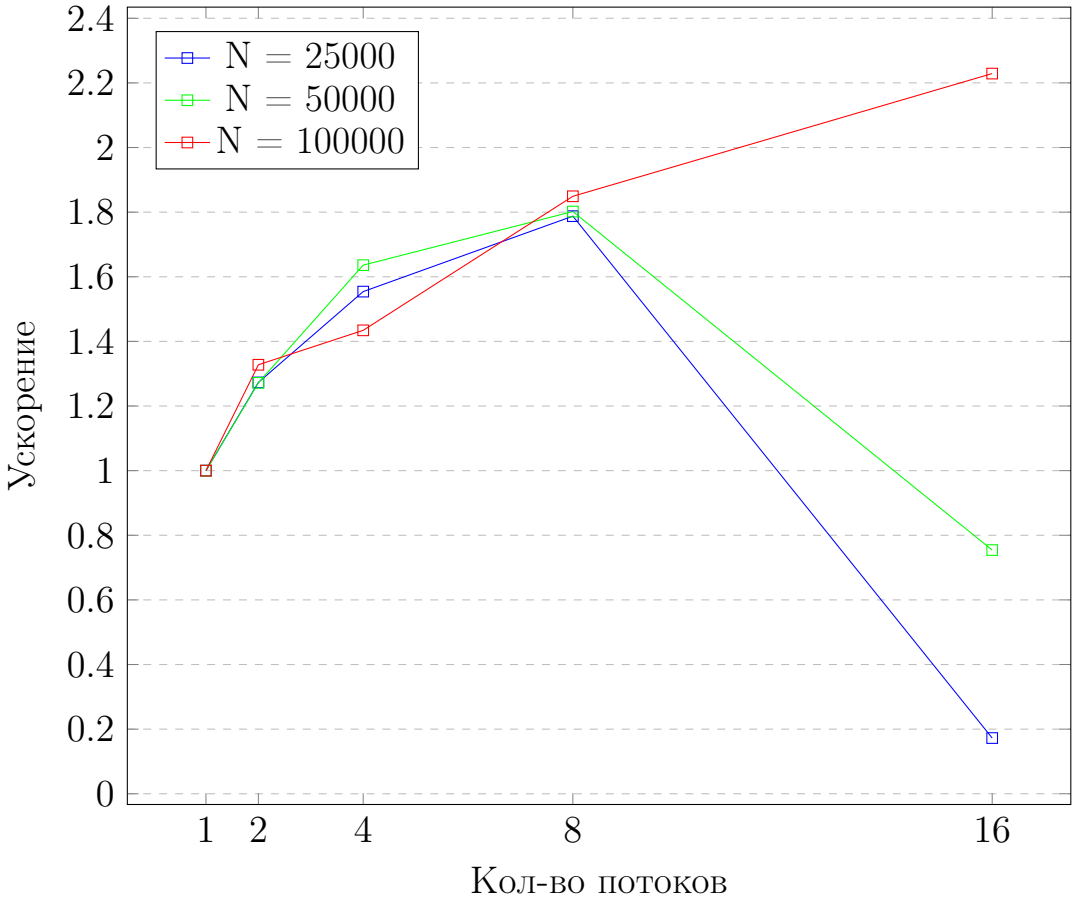

Рис. 4: Зависимость ускорения сортировки от количества процессов для массивов различного размера на Харизме

По графикам 3-4 видно, что результат аналогичен, но быстрее чем на предыдущей машине и при меньших размерах массивов с 16 потоками наблюдается ухудшение результатов по сравнению с последовательным вычислением. Также стоит заметить, что для наибольшего массива ускорение на 16 потоках особенно заметно.

Реализация на python

На питоне аналогично были реализованы последовательные (листинг 3) и параллельные (листинги 4-5) алгоритмы сортировки Хоара.

Первая последовательная реализация не использует декоратор @njit, а поэтому является самой медленной и показала результат хуже чем аналогичный алгоритм на C примерно в 5 раз. Вторая и третья реализации используют декоратор @njit, но одна из них рекурсивная, а другая итерационная и использует стек и поэтому медленней и для сравнения с параллельными алгоритмами использовалась рекурсивная реализация с декоратором @njit.

Одна параллельная реализация сделана при помощи библиотеки numba и

декоратора @njit(parallel=True), а вторая при помощи numba.openmp. Библиотеку numba.openmp возможно использовать только на Linux подобных системах, поэтому она была установлена в docker контейнер, но в нем было доступно использование только четырех потоков. Для чистоты эксперимента, при сравнении алгоритмов на Python, все вычисления производились через docker контейнер.

На графиках 5-8 представлены результаты оценки времени работы данных алгоритмов. На первых двух графиков время выполнения, а на следующих двух ускорение.

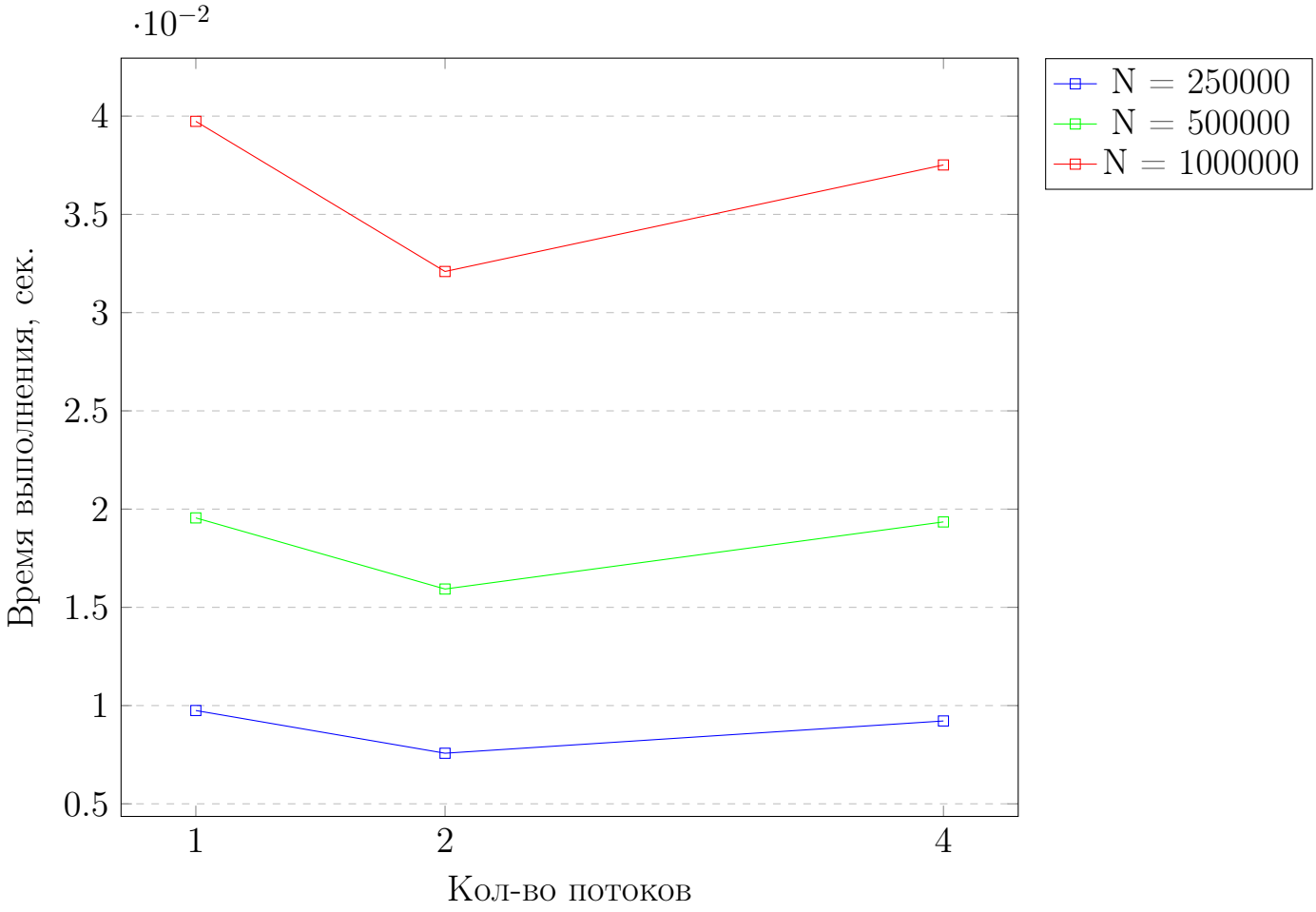

Рис. 5: Зависимость времени выполнения сортировки от количества процессов для массивов различного размера с использованием numba

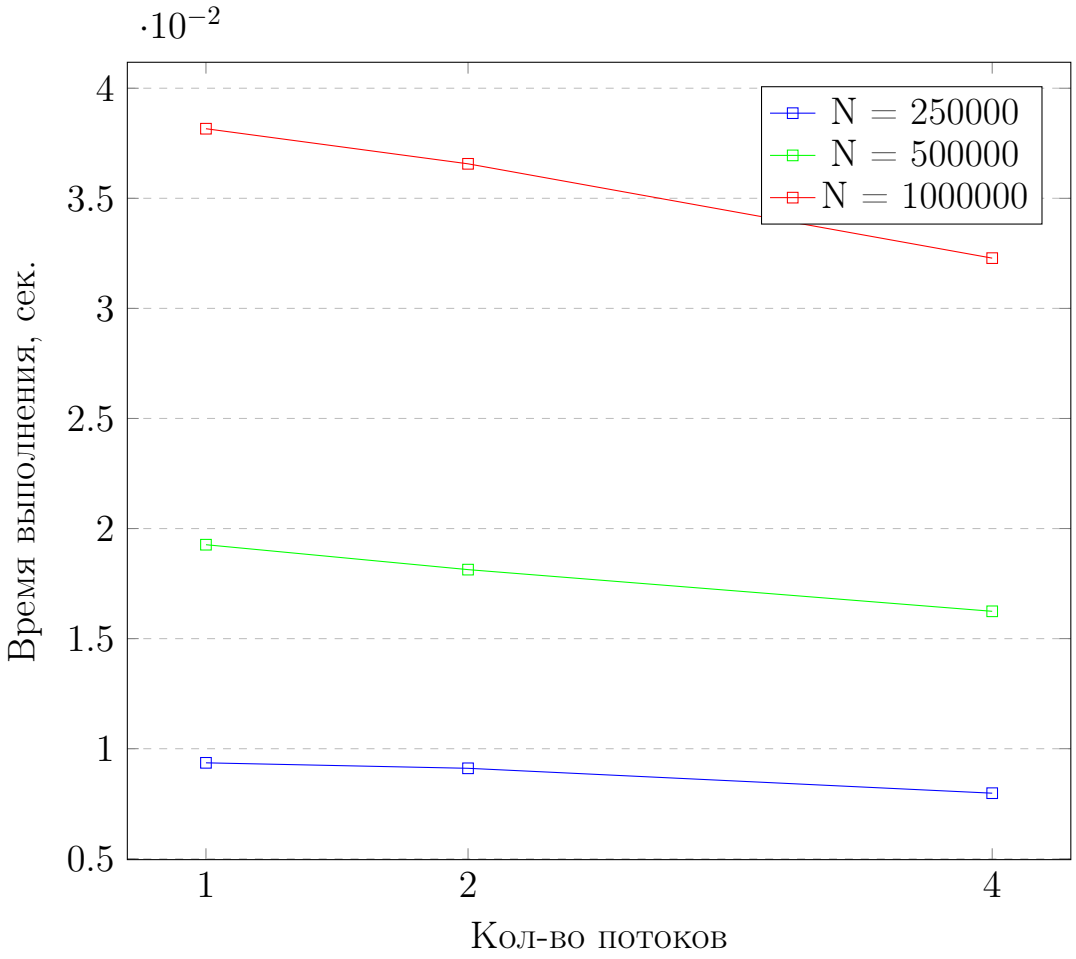

Рис. 6: Зависимость времени выполнения сортировки от количества процессов для массивов различного размера с использованием numba.openmp

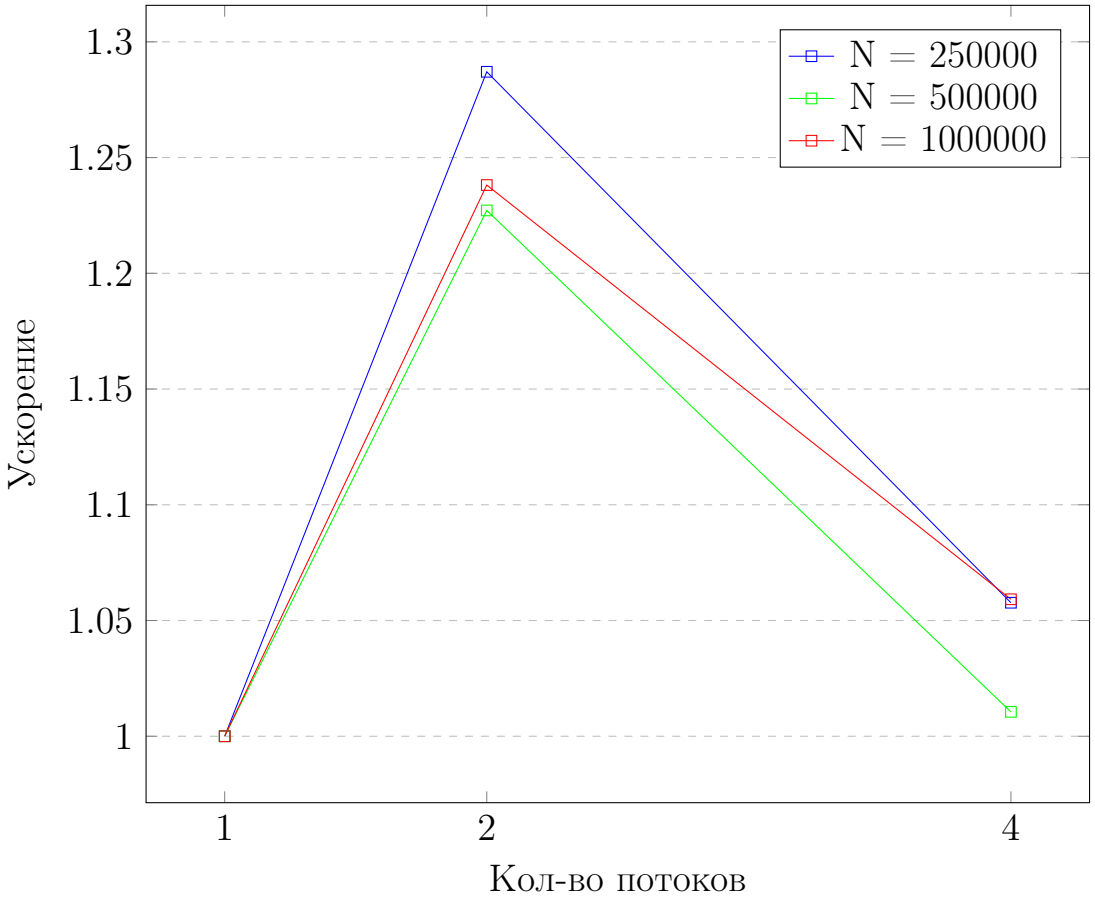

Рис. 7: Зависимость ускорения сортировки от количества процессов для массивов различного размера с использованием numba

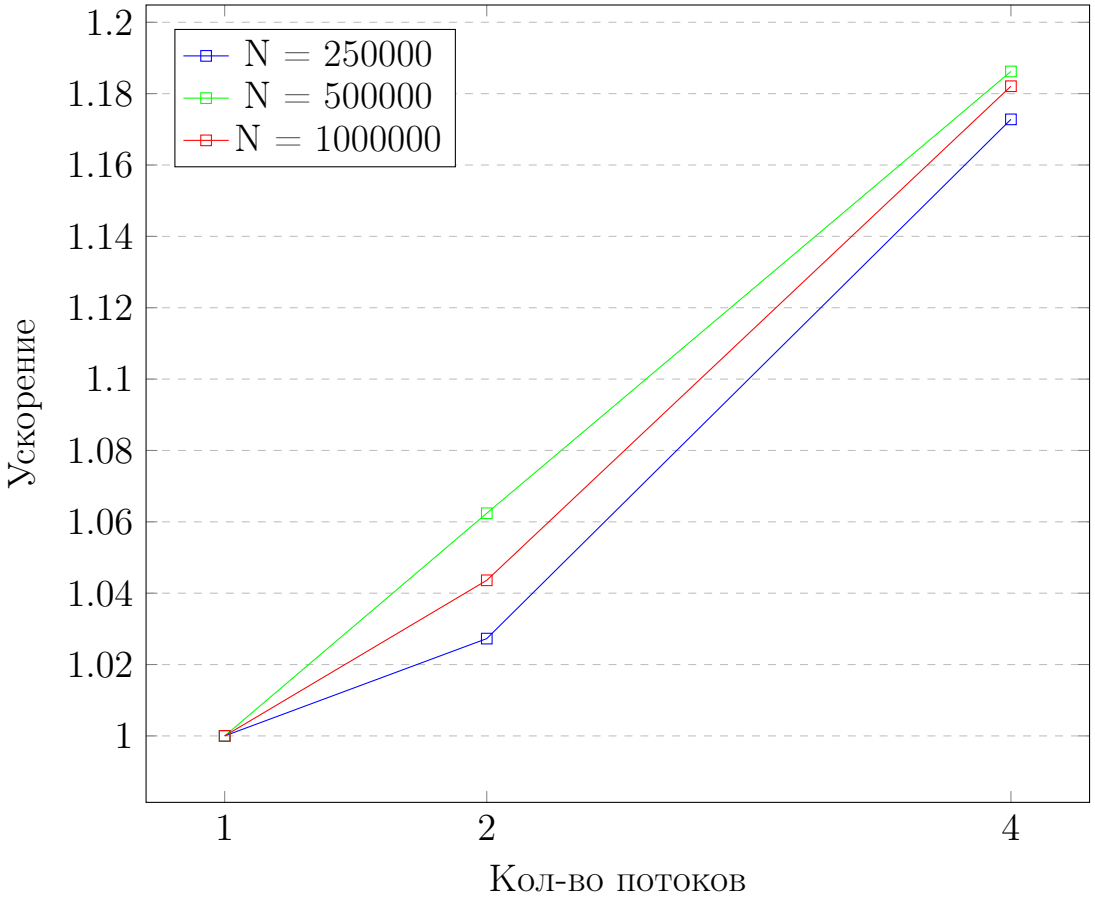

Рис. 8: Зависимость ускорения сортировки от количества процессов для массивов различного размера с использованием numba.openmp

По графику 5 и 7 видно, что заметное ускорение происходит только при двух потоках. А в случае использования numba.openmp ускорение наблюдается и при четырех потоках.

Попробуем сравнить последовательную реализацию и параллельную с numba.openmp на суперкомпьютере Харизма (рис. 9-10):

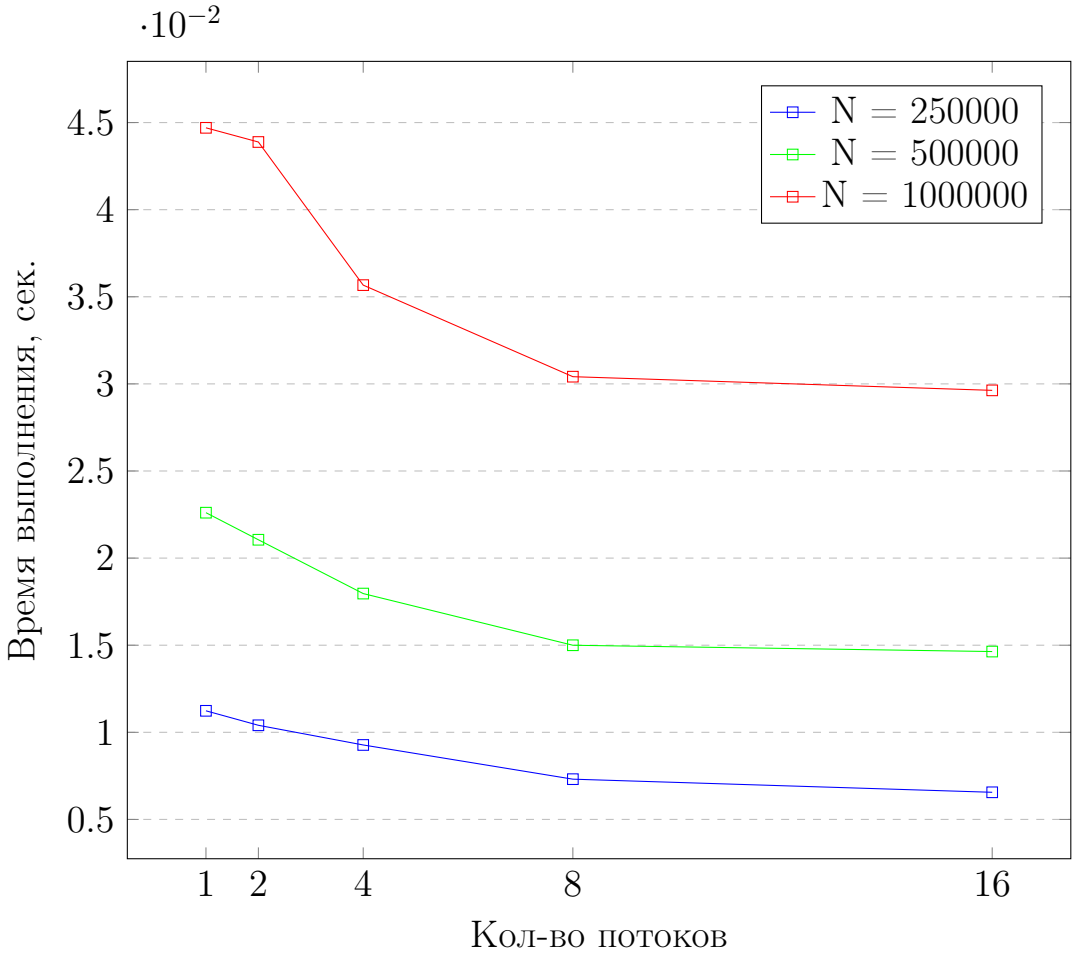

Рис. 9: Зависимость времени выполнения сортировки от количества процессов для массивов различного размера с использованием numba.openmp на Харизме

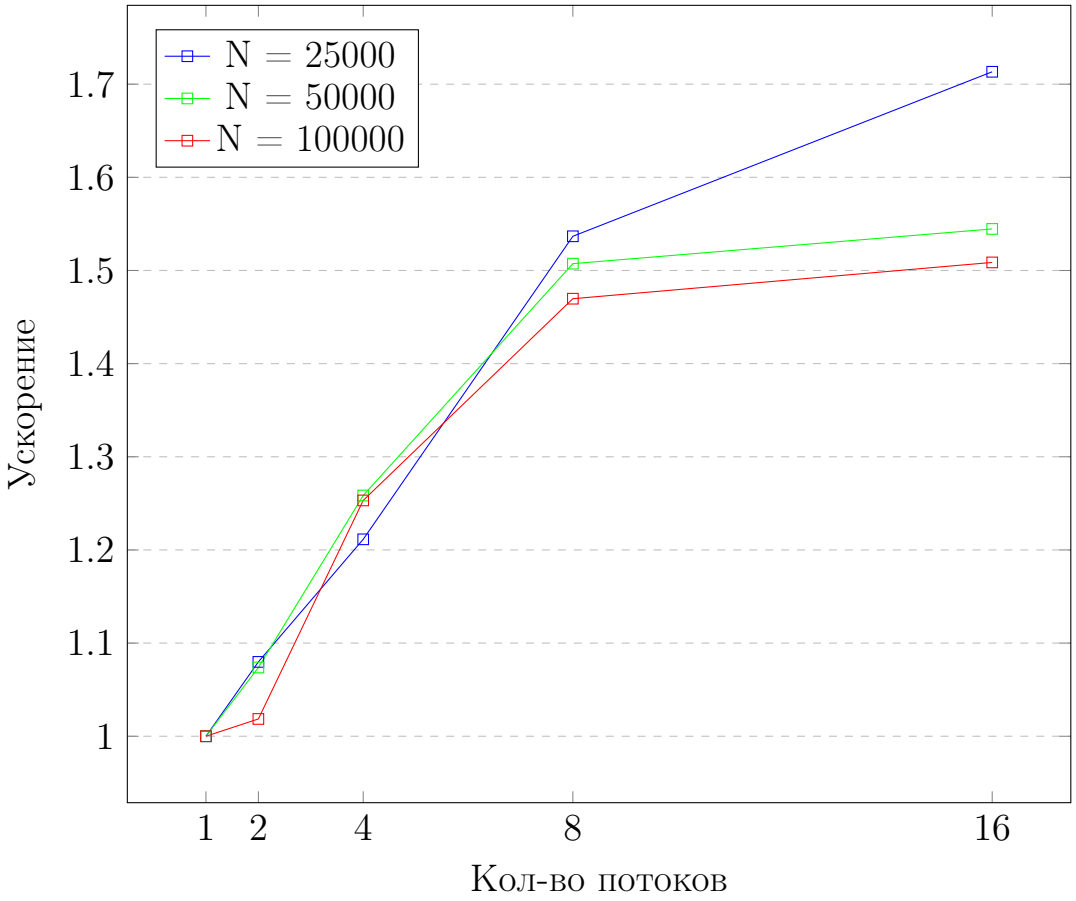

Рис. 10: Зависимость ускорения сортировки от количества процессов для массивов различного с использованием numba.openmp размера на Харизме

Сравнение C и Python

Т.к. реализация на Python с использованием numba.openmp показала себя лучше, будем использовать ее для сравнения. Для начала приведем графики для массивов одинакового размера (N = 100000) на обоих языках (рис. 11-12):

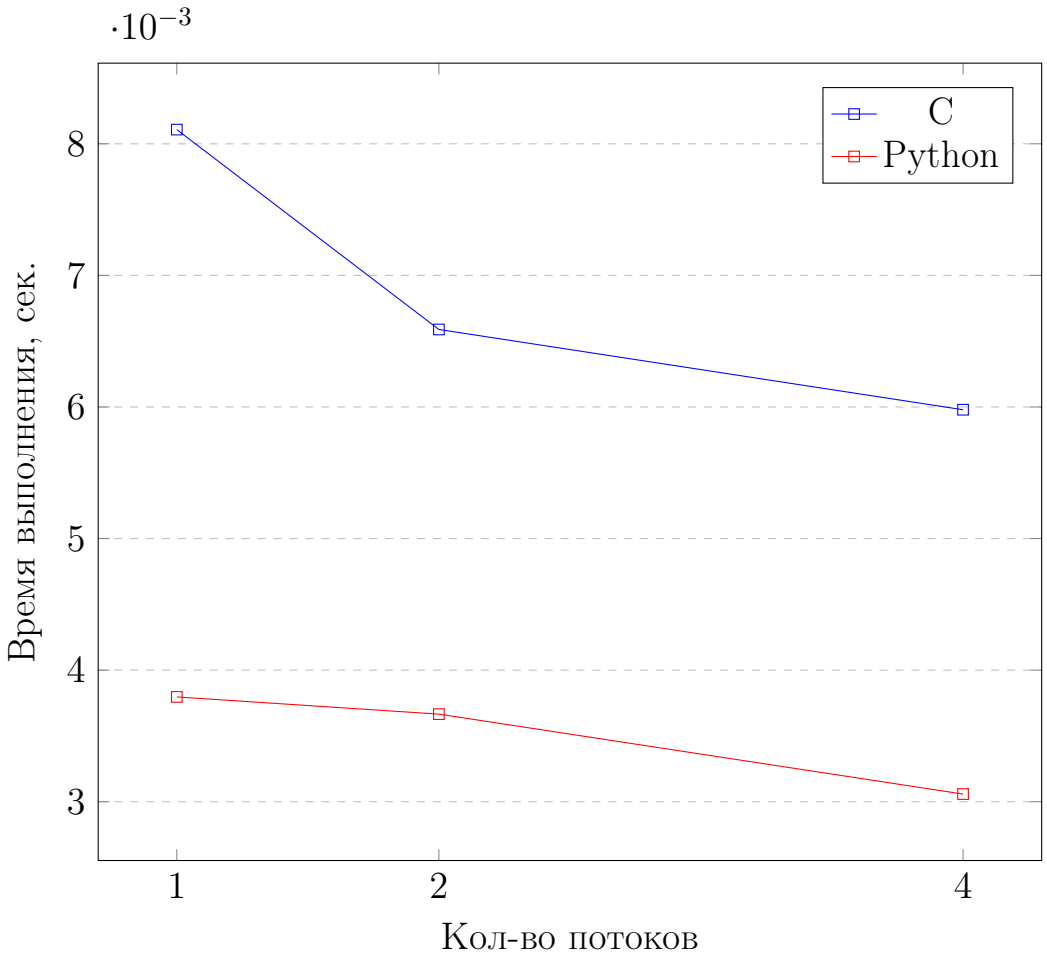

Рис. 11: Зависимость времени выполнения сортировок на разных языках от количества процессов для массивов различного размера

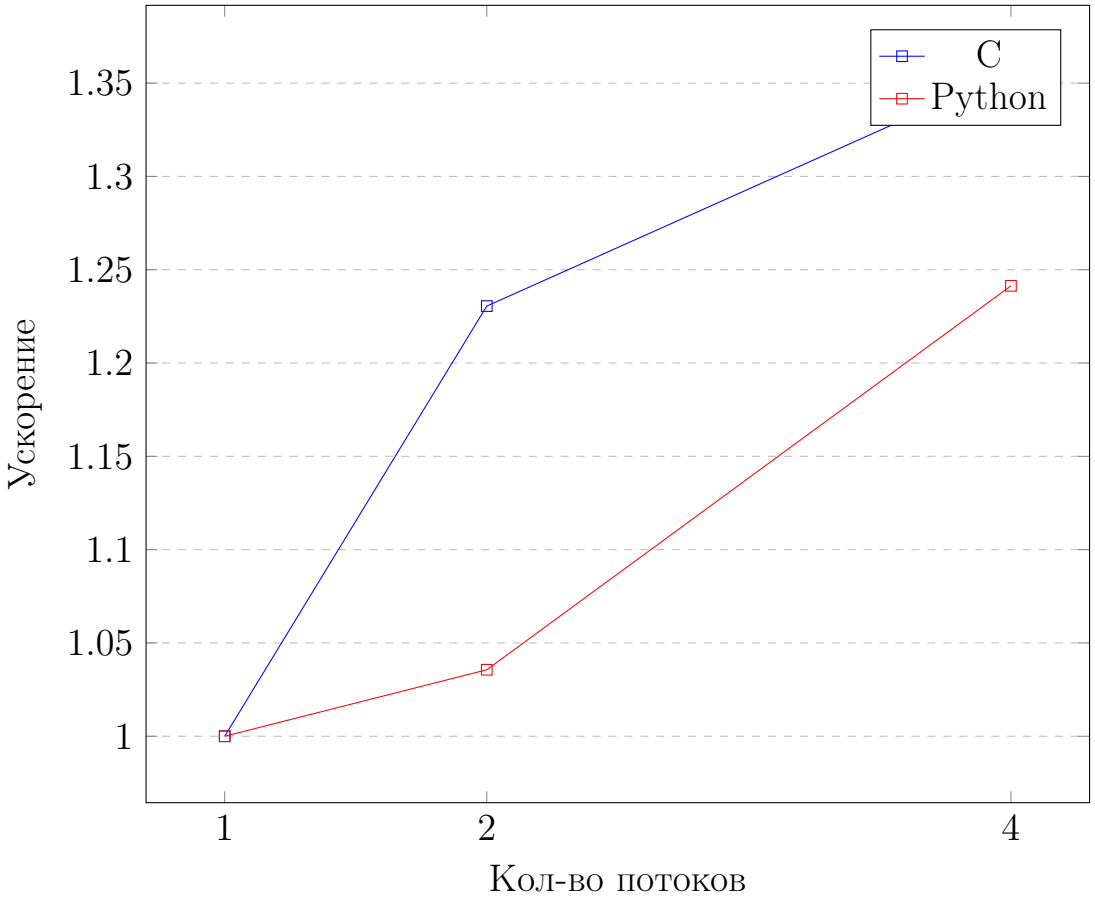

Рис. 12: Ускорение сортировок на разных языках от количества процессов для массивов различного размер

Из графиков 11-12 можно сделать вывод, что реализация на python быстрее из-за оптимизации numba при помощи декоратора @njit, т.к. без него алгоритм работает медленнее чем на C. Но при этом ускорение на языке C больше чем на Python.

Попробуем сделать аналогичное сравнение на суперкомпьютере Харизма, но на массивах больше в 10 раз (N = 1000000) (рис. 13-14):

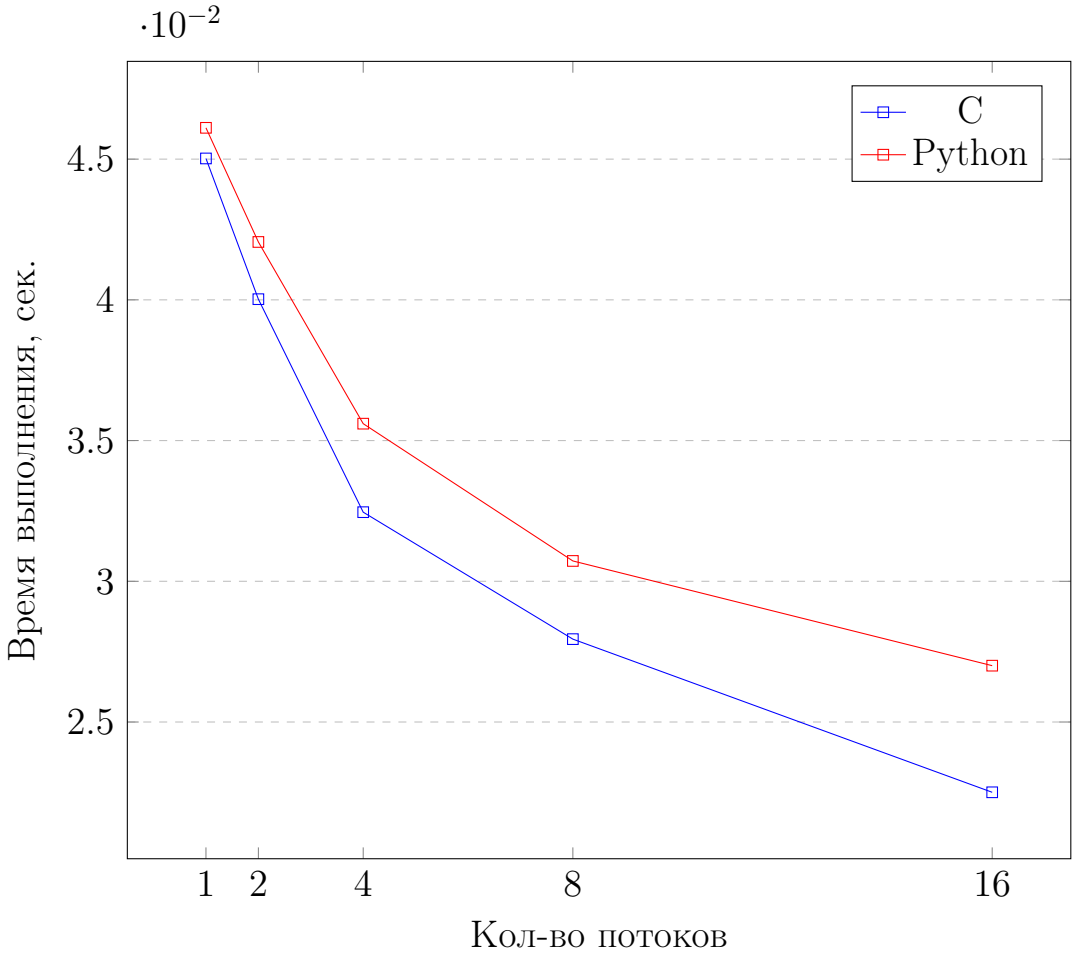

Рис. 13: Зависимость времени выполнения сортировок на разных языках от количества процессов для массивов различного размера на Харизме

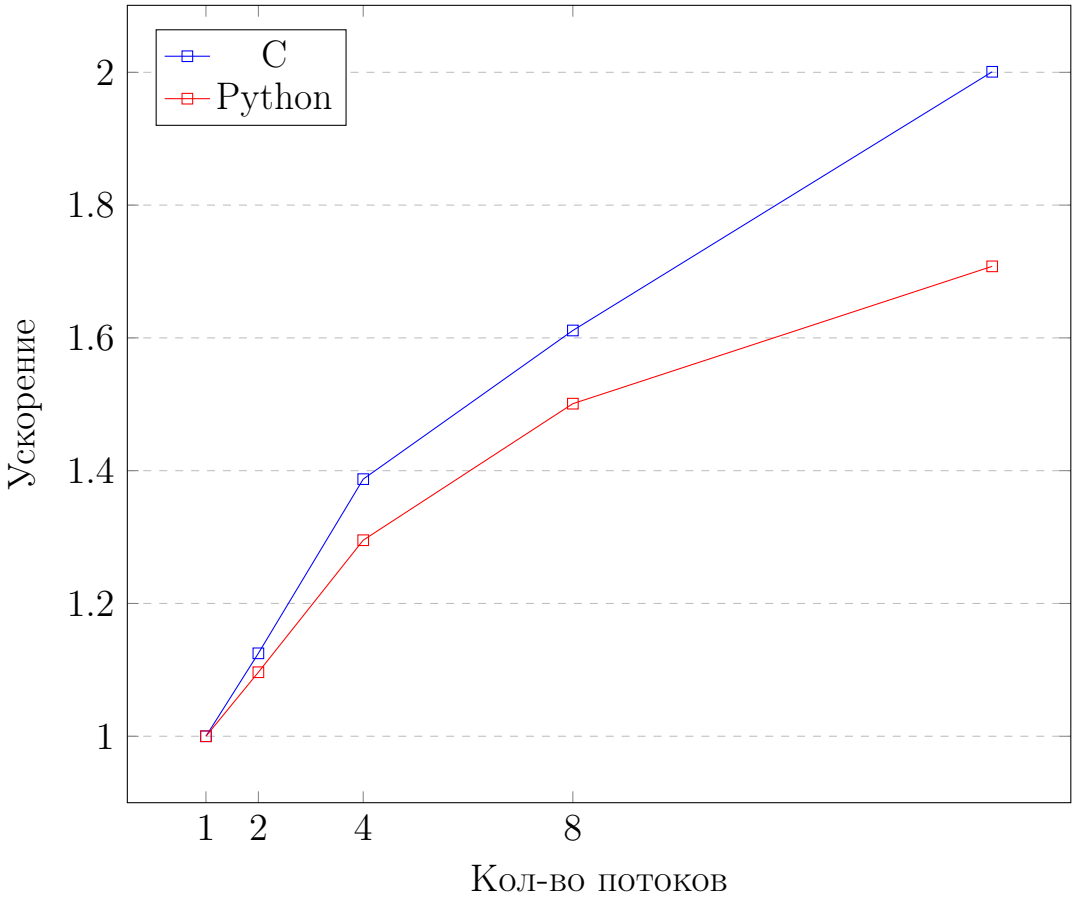

Рис. 14: Ускорение сортировок на разных языках от количества процессов для массивов различного размер на Харизме

Как видно по графиками 13-14, на Харизме реализация на C показала себя лучше, чем на Python. Вероятно это из-за того что на суперкомпьютере использовался компилятор g++ с флагом оптимизации -O3, а на предыдущей машине MSVC с флагом /O2.

Выводы

По итогу проделанной работы были реализованы последовательные и параллельные алгоритмы сортировки Хоара на языках C и Python с использованием технологии OpenMP, а также проведено их исследование.

Из экспериментов можно сделать вывод, что очень важно оптимально распараллеливать алгоритм для получения выигрыша по времени, а также большую роль в этом играют: оптимизация кода, технические параметры, выбор компилятора и т.п.

Листинги

Листинг 1: Последовательный алгоритм сортировки Хоара на языке C

```
void swap(int* a, int* b)
{
  if (*a == *b) return;
  *a = *a + *b;
  *b = *a - *b;
  *a = *a - *b;
}

int partitionLomuto(int* arr, int low, int high)
{
  int pivot = arr[high];
  int i = (low - 1);

  for (int j = low; j < high; j++)
    if (arr[j] <= pivot)
    {
      i++;
      swap(&arr[i], &arr[j]);
    }

  swap(&arr[i + 1], &arr[high]);
  return (i + 1);
}

int partitionHoar(int* arr, int low, int high)
{
  size_t i = low;
  size_t j = high;
  int pivot = arr[(i + j) / 2];
  while (1)
  {
    while (arr[i] < pivot)
      ++i;
    while (arr[j] > pivot)
      --j;

    if (i >= j)
      break;

    swap(&arr[i++], &arr[j--]);
  }
  return j;
}

```

```
45 void quickSortSeq(int* arr, int low, int high)
46 {
47   if (low < high)
48   {
49     int pi = partitionHoar(arr, low, high);
50     // Рекурсивно сортируем элементы, находящиеся до пивота
51     quickSortSeq(arr, low, pi);
52     // Рекурсивно сортируем элементы, находящиеся после пивота
53     quickSortSeq(arr, pi + 1, high);
54   }
55 }
```

Листинг 2: Параллельные алгоритмы сортировки Хоара на языке C

```
1  void swap(int* a, int* b)
2  {
3    if (*a == *b) return;
4    *a = *a + *b;
5    *b = *a - *b;
6    *a = *a - *b;
7  }
8
9  int partitionLomuto(int* arr, int low, int high)
10 {
11   int pivot = arr[high];
12   int i = (low - 1);
13
14   for (int j = low; j < high; j++)
15     if (arr[j] <= pivot)
16     {
17       i++;
18       swap(&arr[i], &arr[j]);
19     }
20
21   swap(&arr[i + 1], &arr[high]);
22   return (i + 1);
23 }
24
25 int partitionHoar(int* arr, int low, int high)
26 {
27   size_t i = low;
28   size_t j = high;
29   int pivot = arr[(i + j) / 2];
30   while (1)
31   {
32     while (arr[i] < pivot)
33       ++i;
34     while (arr[j] > pivot)
35       --j;
36
37     if (i >= j)
38       break;
```

```
39
40      swap(&arr[i++], &arr[j--]);
41    }
42    return j;
43  }
44
45  //Working only with 2 threads - with more WRONG
46  void quickSortParOld(int* arr, int low, int high, int max_d, int d = 0)
47  {
48    if (low < high)
49    {
50      int pi = partitionHoar(arr, low, high);
51
52      if (d < max_d)
53      {
54        #pragma omp parallel sections
55        {
56          #pragma omp section
57          {
58            //printf("omp_get_thread_num() = %d\n", omp_get_thread_num());
59            // Рекурсивно сортируем элементы, находящиеся до пивота
60            quickSortParOld(arr, low, pi, max_d, d + 1);
61          }
62          #pragma omp section
63          {
64            //printf("omp_get_thread_num() = %d\n", omp_get_thread_num());
65            // Рекурсивно сортируем элементы, находящиеся после пивота
66            quickSortParOld(arr, pi + 1, high, max_d, d + 1);
67          }
68        }
69        //#pragma omp taskwait
70      }
71      else
72      {
73        {
74          //printf("omp_get_thread_num() = %d\n", omp_get_thread_num());
75          // Рекурсивно сортируем элементы, находящиеся до пивота
76          quickSortParOld(arr, low, pi, max_d, d + 1);
77          // Рекурсивно сортируем элементы, находящиеся после пивота
78          quickSortParOld(arr, pi + 1, high, max_d, d + 1);
79        }
80      }
81    }
82  }
83
84  void quickSortParOld2(int* arr, int low, int high, int max_d, int d = 0)
85  {
86    if (low < high)
87    {
88      int pi = partitionHoar(arr, low, high);
89
```

```
90      #pragma omp task if (d < max_d)
91      {
92        //printf("omp_get_thread_num() = %d\n", omp_get_thread_num());
93        quickSortParOld2(arr, low, pi, max_d, d + 1);
94      }
95      #pragma omp task if (d < max_d)
96      {
97        //printf("omp_get_thread_num() = %d\n", omp_get_thread_num());
98        quickSortParOld2(arr, pi + 1, high, max_d, d + 1);
99      }
100   }
101 }
102
103 void quickSortPar(int* arr, int low, int high, int max_d, int d = 0)
104 {
105   if (low < high)
106   {
107     int pi = partitionHoar(arr, low, high);
108
109     if (d < max_d)
110     {
111       #pragma omp task
112       {
113         //printf("omp_get_thread_num() = %d\n", omp_get_thread_num());
114         // Рекурсивно сортируем элементы, находящиеся до пивота
115         quickSortPar(arr, low, pi, max_d, d + 1);
116       }
117       #pragma omp task
118       {
119         //printf("omp_get_thread_num() = %d\n", omp_get_thread_num());
120         // Рекурсивно сортируем элементы, находящиеся после пивота
121         quickSortPar(arr, pi + 1, high, max_d, d + 1);
122       }
123     }
124     else
125     {
126       {
127         //printf("omp_get_thread_num() = %d\n", omp_get_thread_num());
128         // Рекурсивно сортируем элементы, находящиеся до пивота
129         quickSortPar(arr, low, pi, max_d, d + 1);
130         // Рекурсивно сортируем элементы, находящиеся после пивота
131         quickSortPar(arr, pi + 1, high, max_d, d + 1);
132       }
133     }
134   }
135 }
```

Листинг 3: Последовательные алгоритм сортировки Хоара на языке Python

```
import numpy as np
from numba import njit
import time

def quickSortSeq(arr, low, high):
    if (low < high):
        # partitionHoar
        i = low
        j = high
        pivot = arr[(i + j) // 2]
        while (1):
            while (arr[i] < pivot):
                i = i + 1
            while (arr[j] > pivot):
                j = j - 1

            if (i >= j):
                break

            arr[i], arr[j] = arr[j], arr[i]
            i = i + 1
            j = j - 1
        pi = j
        # recurcive call
        quickSortSeq(arr, low, pi)
        quickSortSeq(arr, pi + 1, high)

@njit
def quickSortSeqNjit(arr, low, high):
    if (low < high):
        # partitionHoar
        i = low
        j = high
        pivot = arr[(i + j) // 2]
        while (1):
            while (arr[i] < pivot):
                i = i + 1
            while (arr[j] > pivot):
                j = j - 1

            if (i >= j):
                break

            arr[i], arr[j] = arr[j], arr[i]
            i = i + 1
            j = j - 1
        pi = j
        # recurcive call
        quickSortSeqNjit(arr, low, pi)
        quickSortSeqNjit(arr, pi + 1, high)
```

```
51
52 @njit
53 def quickSortIterative(arr):
54     stack = []
55     stack.append((0, len(arr) - 1))
56
57     while stack:
58         low, high = stack.pop()
59
60         if low < high:
61             # partitionHoar
62             i = low
63             j = high
64             pivot = arr[(i + j) // 2]
65             while (1):
66                 while (arr[i] < pivot):
67                     i = i + 1
68                 while (arr[j] > pivot):
69                     j = j - 1
70
71                 if (i >= j):
72                     break
73
74                 arr[i], arr[j] = arr[j], arr[i]
75                 i = i + 1
76                 j = j - 1
77             pivot = j
78             # using stack
79             stack.append((low, pivot))
80             stack.append((pivot + 1, high))
```

Листинг 4: Параллельный алгоритм сортировки Хоара на языке Python с использованием numba

```
import numpy as np
from numba import config, njit, prange
import time
import numba

@njit(parallel=True)
def quickSortPar(arr, low, high):
    if low < high:
        i = low
        j = high
        pivot = arr[(i + j) // 2]
        while True:
            while arr[i] < pivot:
                i = i + 1
            while arr[j] > pivot:
                j = j - 1

            if i >= j:
                break

            arr[i], arr[j] = arr[j], arr[i]
            i = i + 1
            j = j - 1
        pi = j

        for id in prange(2):
            if (id == 0):
                quickSortSeq(arr, low, pi)
            else:
                quickSortSeq(arr, pi + 1, high)
```

Листинг 5: Параллельный алгоритм сортировки Хоара на языке Python с использованием numba.openmp

```
from numba import njit
from numba.openmp import openmp_context as openmp
from numba.openmp import omp_get_thread_num, omp_get_num_threads, omp_set_num_threads,
    omp_get_max_threads, omp_get_wtime
import numpy as np
from numba import njit
from numba.openmp import openmp_context as openmp
import random

@njit
def quickSortPar(arr, low, high, max_d, d = 0):
    if (low < high):
        # partitionHoar
        i = low
        j = high
        pivot = arr[(i + j) // 2]
        while (1):
            while (arr[i] < pivot):
                i = i + 1
            while (arr[j] > pivot):
                j = j - 1

            if (i >= j):
                break

            arr[i], arr[j] = arr[j], arr[i]
            i = i + 1
            j = j - 1
        pi = j

        if (d < max_d):
            with openmp("task␣shared(arr)"):
                quickSortPar(arr, low, pi, max_d, d + 1)
            with openmp("task␣shared(arr)"):
                quickSortPar(arr, pi + 1, high, max_d, d + 1)
            with openmp("taskwait"):
                return
        else:
            quickSortPar(arr, low, pi, max_d, d + 1)
            quickSortPar(arr, pi + 1, high, max_d, d + 1)

@njit
def quickSortParHelp(arr, max_d):
    with openmp("parallel␣shared(arr)"):
        with openmp("single"):
            quickSortPar(arr, 0, len(arr)-1, max_d)
```