

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 Ломута 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).

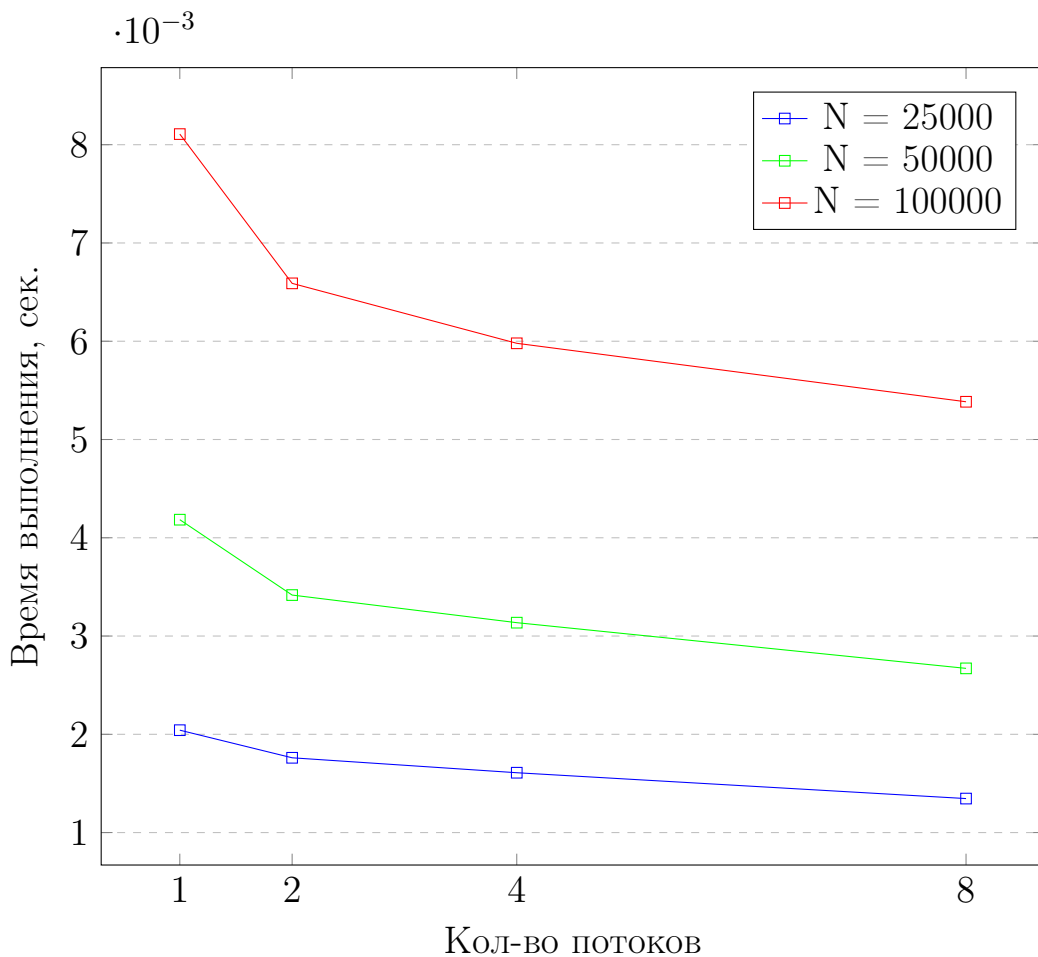


Рис. 1: Зависимость времени выполнения сортировки от количества процессов для массивов различного размера

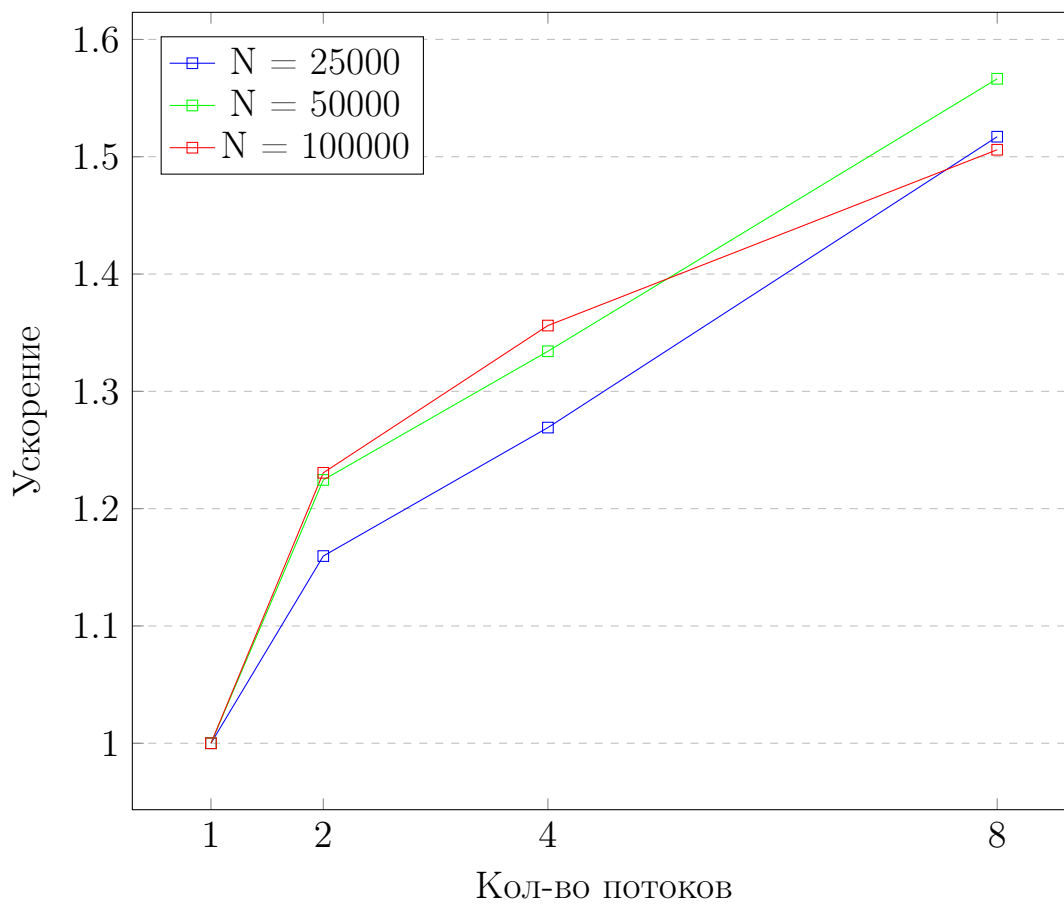


Рис. 2: Зависимость ускорения сортировки от количества процессов для массивов различного размера

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).

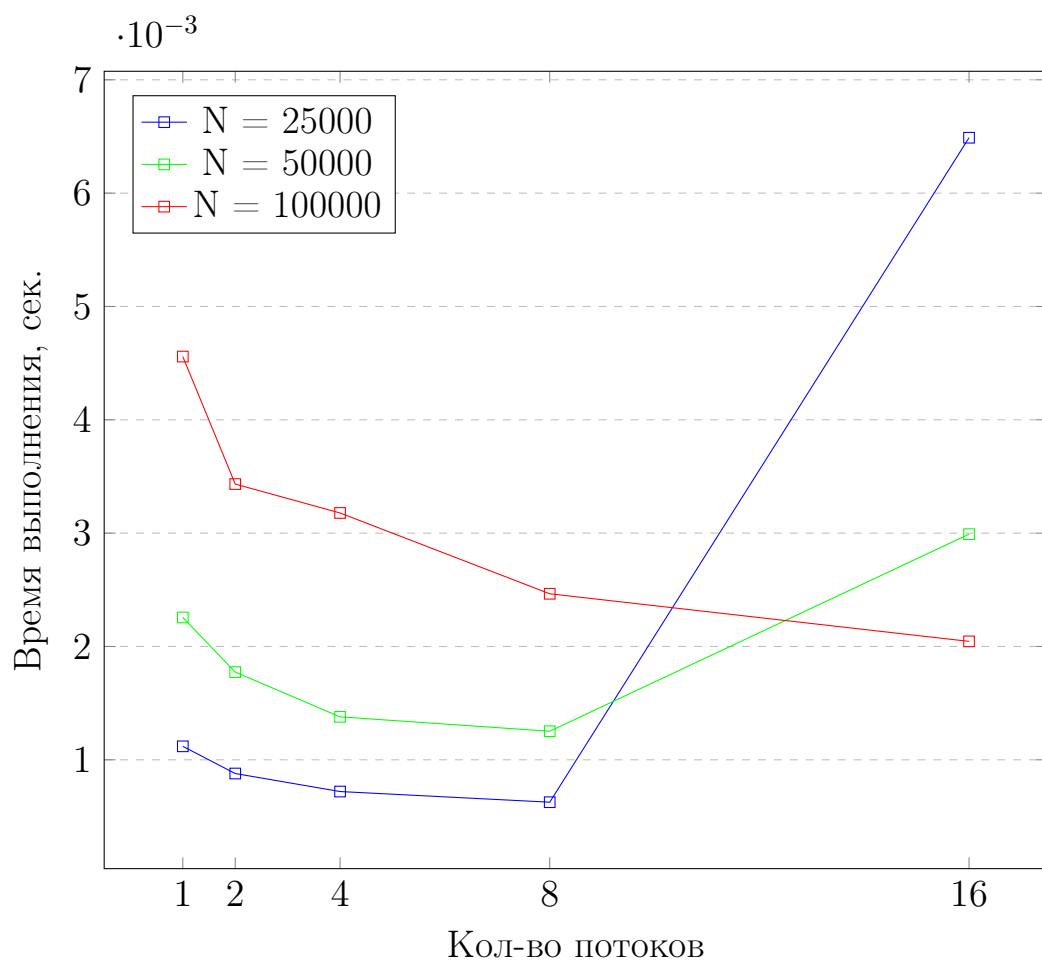


Рис. 3: Зависимость времени выполнения сортировки от количества процессов для массивов различного размера на Харизме

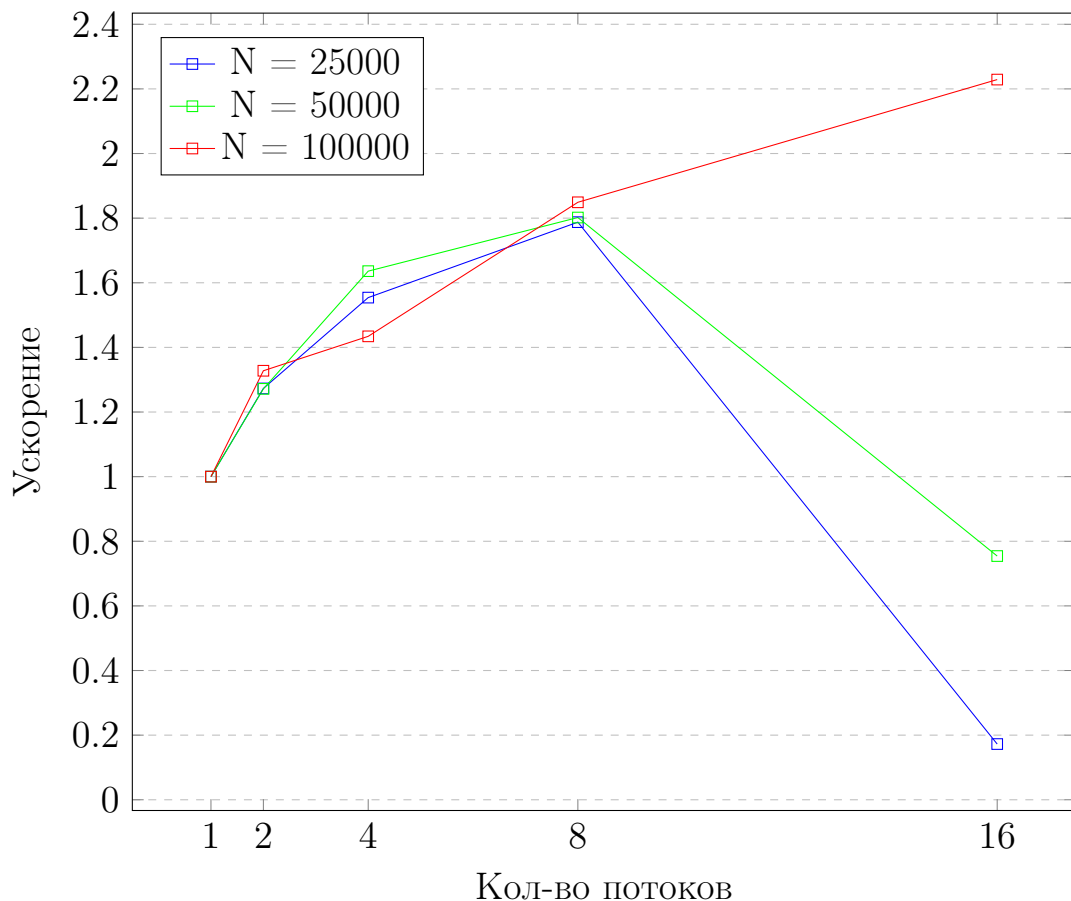


Рис. 4: Зависимость ускорения сортировки от количества процессов для массивов различного размера на Харизме

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 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.orenpnr 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 графиках время выполнения, а на следующих двух ускорение.

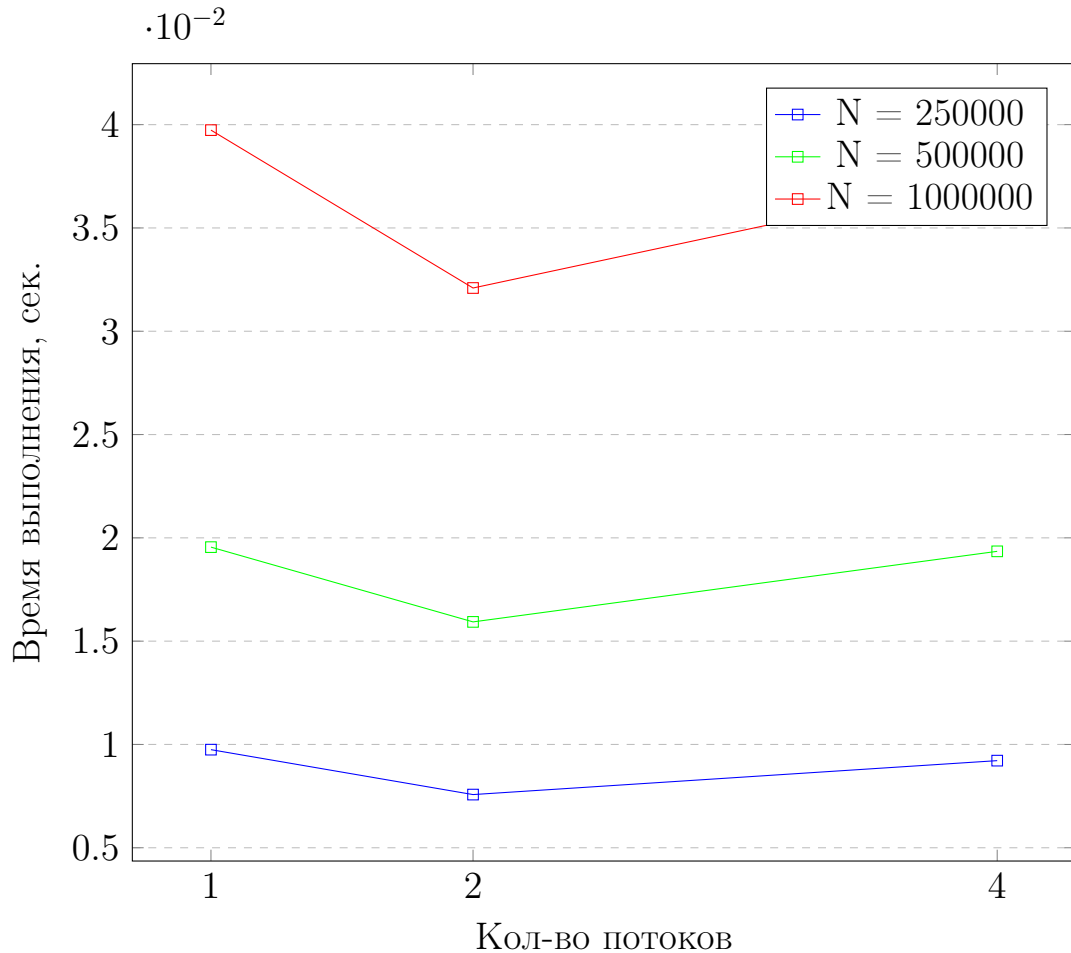


Рис. 5: Зависимость времени выполнения сортировки от количества процессов для массивов различного размера с использованием numba

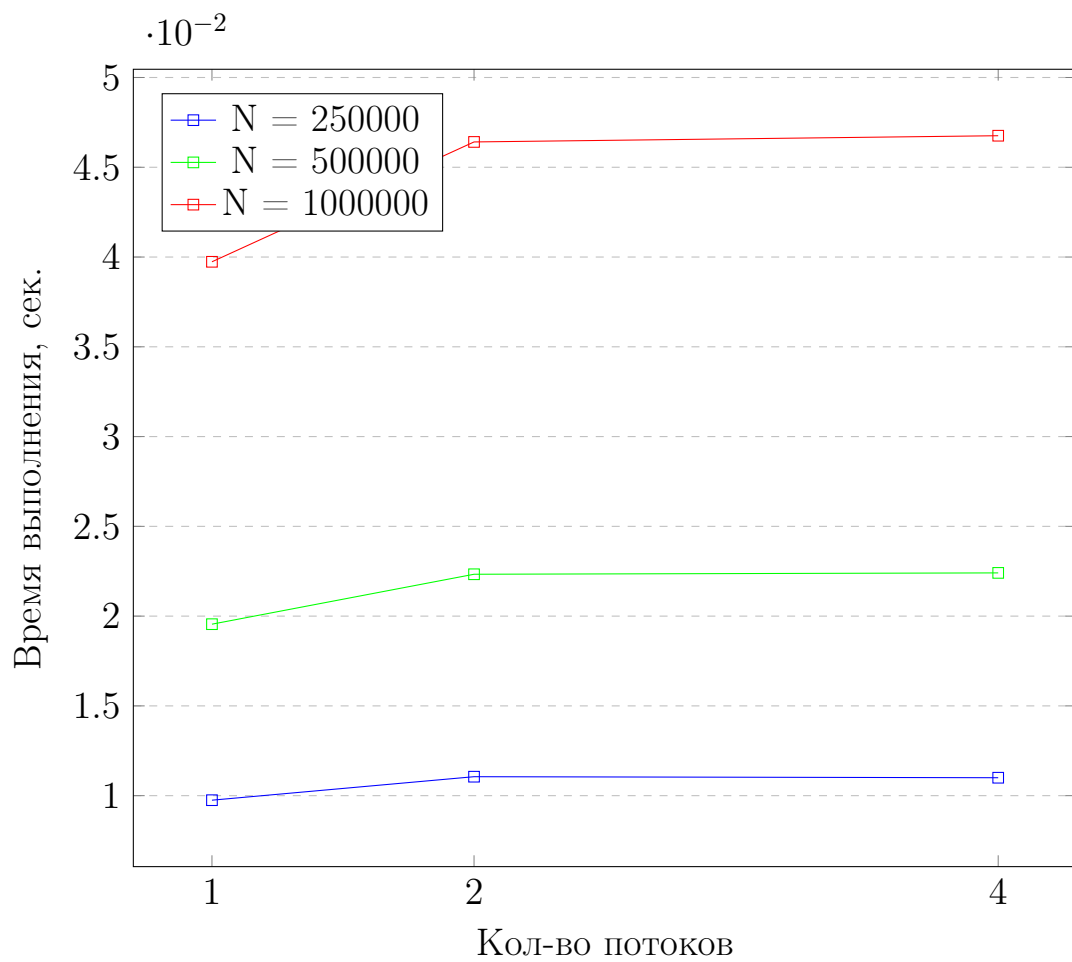


Рис. 6: Зависимость времени выполнения сортировки от количества процессов для массивов различного размера с использованием `numba.openmp`

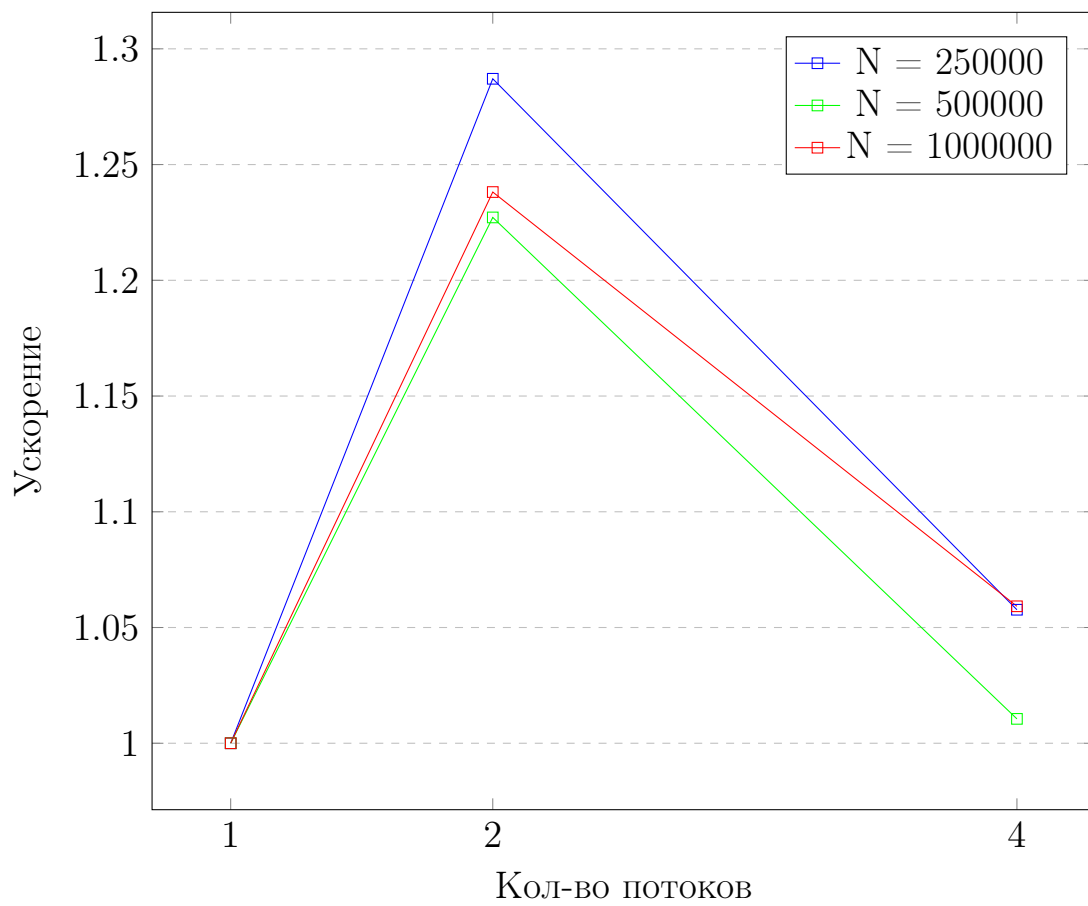


Рис. 7: Зависимость ускорения сортировки от количества процессов для массивов различного размера с использованием numba

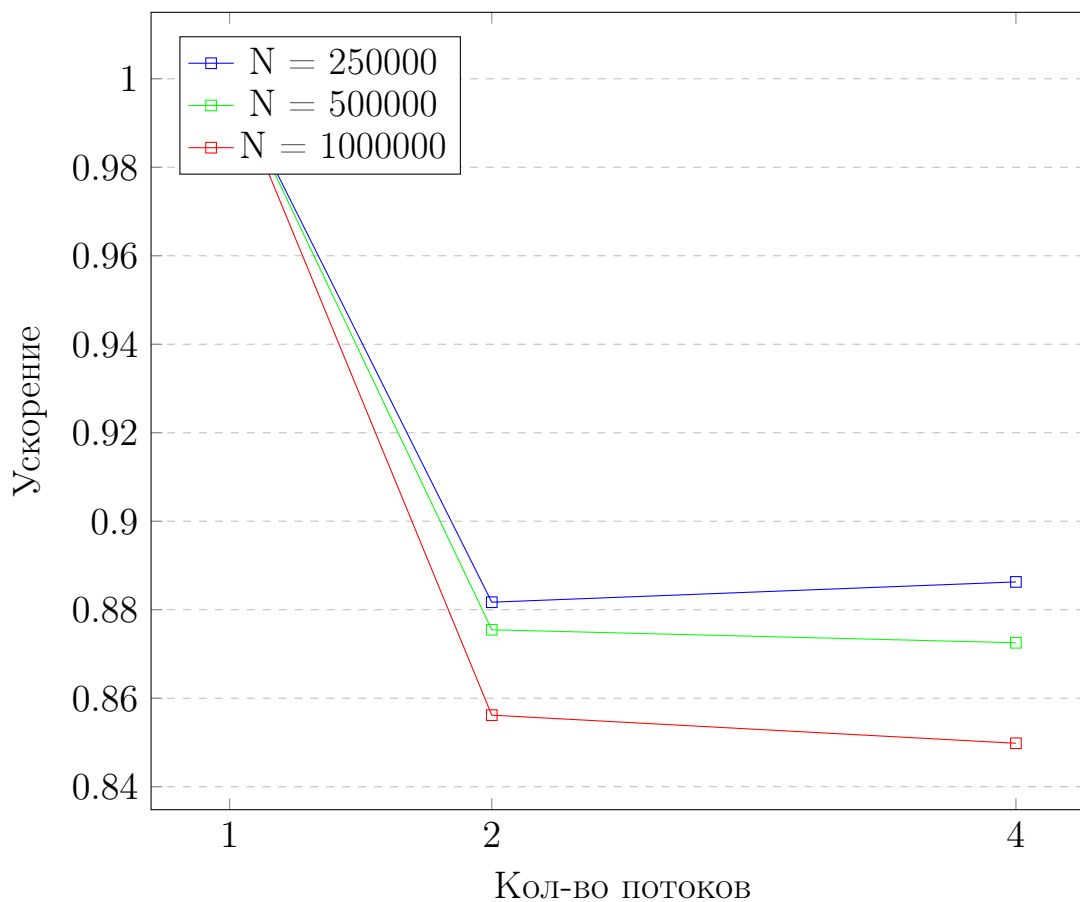


Рис. 8: Зависимость ускорения сортировки от количества процессов для массивов различного размера с использованием `numba.openmp`

По графику 5 и 7 видно, что заметное ускорение происходит только при двух потоках. А в случае использования `numba.openmp`, ускорения вовсе не наблюдается, более того данная реализация показывает себя медленнее.

Сравнение C и Python

Т.к. реализация на Python с использованием `numba` показала себя лучше, будем использовать ее для сравнения. Для начала приведем графики для массивов одинакового размера ($N = 100000$) на обоих языках (9-10)

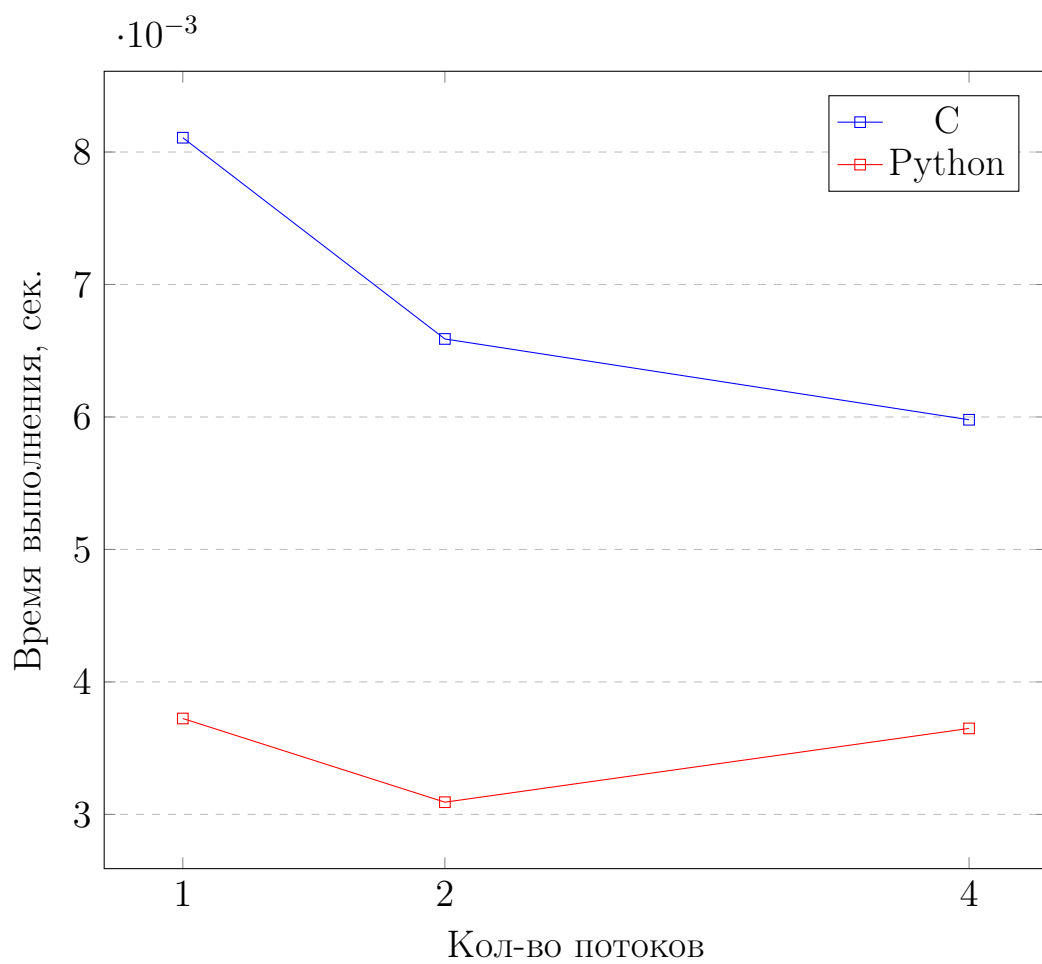


Рис. 9: Зависимость времени выполнения сортировки от количества процессов для

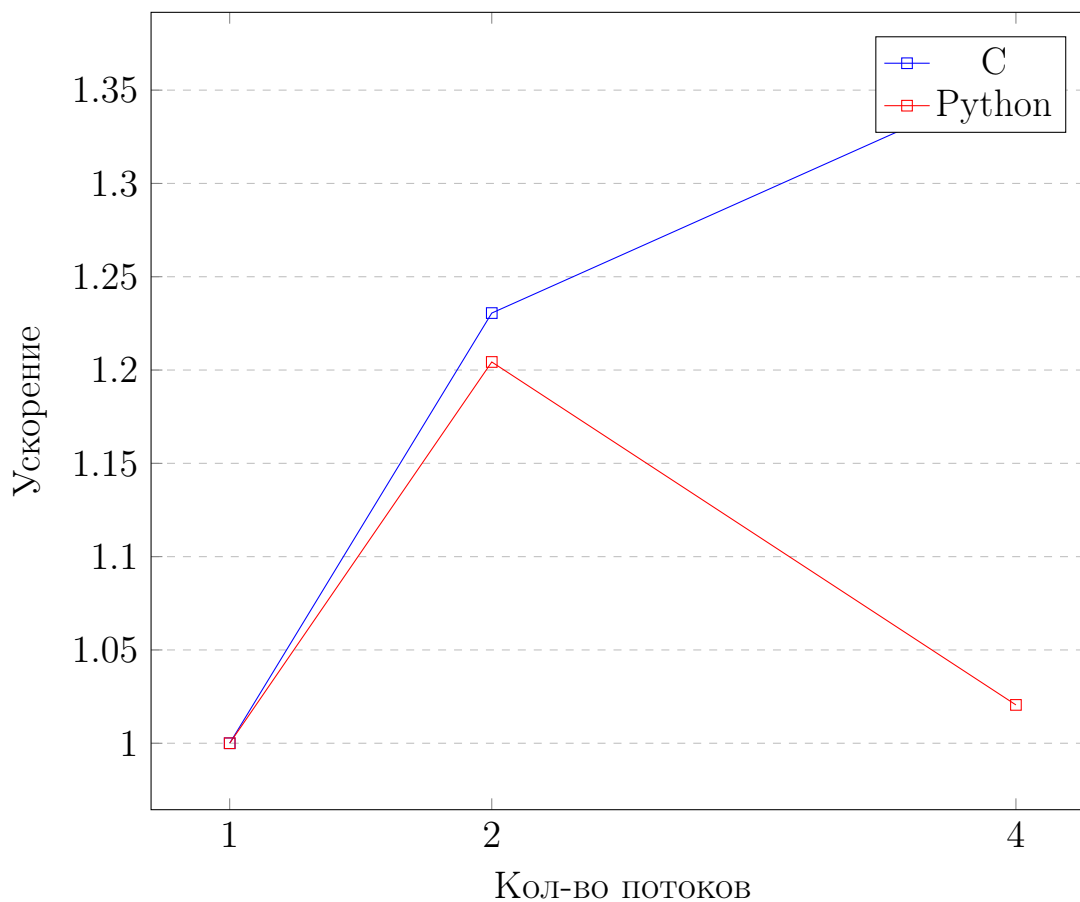


Рис. 10: Ускорение сортировки от количества процессов для

Из графиков 9-10 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 увеличивается до 8 потоков, а на Python только до 2.

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 играет оптимизация кода.

Листинги

Листинг 1: Последовательный алгоритм 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 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

```
1 import numpy as np
2 from numba import njit
3 import time
4
5 def quickSortSeq(arr, low, high):
6     if (low < high):
7         # partitionHoar
8         i = low
9         j = high
10        pivot = arr[(i + j) // 2]
11        while (1):
12            while (arr[i] < pivot):
13                i = i + 1
14            while (arr[j] > pivot):
15                j = j - 1
16
17            if (i >= j):
18                break
19
20            arr[i], arr[j] = arr[j], arr[i]
21            i = i + 1
22            j = j - 1
23        pi = j
24        # recursive call
25        quickSortSeq(arr, low, pi)
26        quickSortSeq(arr, pi + 1, high)
27
28 @njit
29 def quickSortSeqNjit(arr, low, high):
30     if (low < high):
31         # partitionHoar
32         i = low
33         j = high
34         pivot = arr[(i + j) // 2]
35         while (1):
36             while (arr[i] < pivot):
37                 i = i + 1
38             while (arr[j] > pivot):
39                 j = j - 1
40
41             if (i >= j):
42                 break
43
44             arr[i], arr[j] = arr[j], arr[i]
45             i = i + 1
46             j = j - 1
47         pi = j
48         # recursive call
49         quickSortSeqNjit(arr, low, pi)
50         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

```
1 import numpy as np
2 from numba import config, njit, prange
3 import time
4 import numba
5
6 @njit(parallel=True)
7 def quickSortPar(arr, low, high):
8     if low < high:
9         i = low
10        j = high
11        pivot = arr[(i + j) // 2]
12        while True:
13            while arr[i] < pivot:
14                i = i + 1
15            while arr[j] > pivot:
16                j = j - 1
17
18            if i >= j:
19                break
20
21            arr[i], arr[j] = arr[j], arr[i]
22            i = i + 1
23            j = j - 1
24        pi = j
25
26        for id in prange(2):
27            if (id == 0):
28                quickSortSeq(arr, low, pi)
29            else:
30                quickSortSeq(arr, pi + 1, high)
```

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

```
1 from numba import njit
2 from numba.openmp import openmp_context as openmp
3 from numba.openmp import omp_get_thread_num, omp_get_num_threads, omp_set_num_threads,
   omp_get_max_threads, omp_get_wtime
4 import numpy as np
5 from numba import njit
6 from numba.openmp import openmp_context as openmp
7 import random
8
9 @njit
10 def quickSortPar(arr, low, high, max_d, d = 0):
11     if (low < high):
12         # partitionHoar
13         i = low
14         j = high
15         pivot = arr[(i + j) // 2]
16         while (1):
17             while (arr[i] < pivot):
18                 i = i + 1
19             while (arr[j] > pivot):
20                 j = j - 1
21
22             if (i >= j):
23                 break
24
25             arr[i], arr[j] = arr[j], arr[i]
26             i = i + 1
27             j = j - 1
28         pi = j
29
30         # recursive call
31         if (d < max_d):
32             with openmp("task_shared(arr, low, pi, max_d, d)":
33                 quickSortPar(arr, low, pi, max_d, d + 1)
34             with openmp("task_shared(arr, pi, high, max_d, d)":
35                 quickSortPar(arr, pi + 1, high, max_d, d + 1)
36         else:
37             quickSortPar(arr, low, pi, max_d, d + 1)
38             quickSortPar(arr, pi + 1, high, max_d, d + 1)
```