Отчёт о выполнении задания

Орловский Ростислав Владиславович

Описание реализованного алгоритма

1. Инициализация (`init`)

- Заполняются массивы A и B, где граничные элементы устанавливаются в нули, а внутренние элементы заполняются значениями $1.0 + i + j$.
- Распараллеливание осуществляется с помощью директивы `#pragma omp for schedule(static)`, чтобы разделить работу между потоками.

2. Основной цикл

- Выполняется максимум $itmax$ итераций.
- На каждой итерации:
  - Используется **чередование массивов A и B**:
    - На нечётных итерациях обновления выполняются из массива B в массив A.
    - На чётных итерациях — наоборот.
  - Это решение исключает необходимость копирования данных между массивами, что экономит ресурсы.
  - Функция `relax` обновляет значения массива с помощью параллельных вычислений.
  - Выводится текущий номер итерации и значение eps, отражающее максимальное изменение температуры за итерацию.
  - Если eps $<$ maxeps, цикл завершается досрочно.

3. Функция `relax`

- Основной цикл по массиву разбивается на части для параллельной обработки потоками.
- Для каждого элемента массива вычисляется среднее значение его соседей, и это значение записывается в другой массив.
- Максимальная разница между старым и новым значениями температуры аккумулируется в eps с помощью директивы `#pragma omp reduction(max:eps)`.

4. Верификация (`verify`)

- Вычисляется сумма всех значений массива S, умноженных на индексы строк и столбцов.
- Используется параллельный цикл с директивой `#pragma omp parallel for reduction(+:s)` для безопасного суммирования значений.
- Чередование массивов A и B позволяет использовать их поочерёдно как источник данных и как место для записи результатов.
- На каждой итерации определяется, какой из массивов будет обновляться, в зависимости от чётности итерации:
  - Нечётные: данные из B, результат в A.
  - Чётные: данные из A, результат в B.

5. Работа `relax`

- Каждый поток берёт часть массива *src* и для каждого элемента вычисляет среднее значение его соседей.
- Результаты записываются в массив *dest*.
- Потоки обновляют глобальную переменную eps для сохранения максимального изменения температуры в процессе итерации.

Изменения в модернизированной версии

1. Добавление параллелизма с использованием OpenMP

- Основной цикл по итерациям, а также операции внутри функций `init`, `relax` и `verify` были распараллелены.
- Используются директивы OpenMP:
  - `#pragma omp parallel`: Создаёт параллельную область, где работа разделяется между потоками.
  - `#pragma omp for`: Разделяет итерации циклов между потоками, обеспечивая равномерное распределение работы.
  - `#pragma omp reduction`: Используется для безопасного объединения результатов, таких как максимумы (`max`) или суммы (`+`).
  - `#pragma omp barrier`: Обеспечивает синхронизацию между потоками, гарантируя завершение всех потоков перед выполнением следующих операций.
  - `#pragma omp master` и `#pragma omp single`: Гарантируют, что определённые действия (например, обновление eps или вывод данных) выполняются только одним потоком.

2. Оптимизация скорости работы программы

- Введена чередующаяся схема работы с матрицами: на нечётных итерациях данные читаются из B, а записываются в A, и наоборот. Это уменьшает количество операций копирования.

Оптимизация с использованием OpenMP

Основные улучшения

- **Распараллеливание вычислений**: Внутренние циклы в `relax` и `verify` распараллелены с использованием `#pragma omp for`, что позволяет каждому потоку обрабатывать свою часть массива.
- **Распределение работы между потоками**: Используется директива `schedule(static)`, чтобы каждый поток обрабатывал равные блоки итераций, улучшая балансировку нагрузки.
- **Безопасное обновление глобальных переменных**: Для обновления переменной eps используется `reduction(max:eps)`, что предотвращает проблемы с одновременным доступом из разных потоков.
- **Минимизация синхронизаций**: Используются конструкции `#pragma omp barrier` и `#pragma omp master`, обеспечивающие синхронизацию только там, где это необходимо.

Ключевые примеры кода

Основной цикл:

```
#pragma omp parallel private(it)
{
    for (it = 1; it <= itmax; it++) {
        #pragma omp barrier
        #pragma omp master
            eps = 0.0;

        if (it % 2 == 1) {
            relax(B, A);
        } else {
            relax(A, B);
        }

        #pragma omp single
        {
            printf("it=%4d␣␣␣eps=%f\n", it, eps);
        }

        if (eps < maxeps) {
            break;
        }
    }
}
```

Функция `relax`:

```
void relax(double src[N][N], double dest[N][N]) {
    #pragma omp for reduction(max:eps) private(i, j) schedule(static)
    for (int i = 1; i <= N-2; i++) {
        for (int j = 1; j <= N-2; j++) {
            double new_val = (src[i-1][j] + src[i+1][j] +
                              src[i][j-1] + src[i][j+1]) / 4.0;

            double diff = fabs(src[i][j] - new_val);
            eps = Max(eps, diff);

            dest[i][j] = new_val;
        }
    }
}
```

Функция `verify`:

```
void verify(double src[N][N]) {
    double s = 0.0;

    #pragma omp parallel for reduction(+:s) private(i, j) schedule(static)
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            s += src[i][j] * (i + 1) * (j + 1);
        }
    }
    #pragma omp master
    {
        s /= (N * N);
        printf("␣␣S␣=␣%f\n", s);
    }
}
```

Итог

Достижения оптимизации

- **Скорость**: Благодаря распараллеливанию, алгоритм эффективно использует ресурсы многоядерных процессоров.
- **Масштабируемость**: Увеличение размера массива N не приводит к значительному увеличению времени выполнения за счёт параллельных вычислений.

Изменения с использованием OpenMP `task`

Описание работы алгоритма

1. Инициализация (`init`)

- Заполняются массивы A и B, где граничные элементы устанавливаются в нули, а внутренние элементы заполняются значениями $1.0 + i + j$.

- Данные инициализируются блочно для повышения эффективности.

2. Основной цикл

- Выполняется максимум $itmax$ итераций.
- На каждой итерации:
  - Используется **чередование массивов** A **и** B:
    - На нечётных итерациях обновления выполняются из массива B в массив A.
    - На чётных итерациях — наоборот.
  - Чередование исключает необходимость копирования данных между массивами после каждой итерации.
  - Функция `relax` обновляет значения массива с помощью блочной обработки данных.
  - Выводится текущий номер итерации и значение eps, отражающее максимальное изменение температуры за итерацию.
  - Если eps $<$ maxeps, цикл завершается досрочно.

3. Функция `relax`

- Массив разбивается на блоки размером TASK_SIZE $\times$ TASK_SIZE.
- Для каждого блока создаются задачи (`#pragma omp task`), которые выполняются потоками.
- Каждая задача обрабатывает свой блок данных:
  - Для каждого элемента блока вычисляется среднее значение его соседей.
  - Результат записывается в массив $dest$.
  - Максимальная разница между старым и новым значением аккумулируется в eps через критическую секцию (`#pragma omp critical`).

4. Верификация (`verify`)

- Вычисляется сумма всех значений массива S, умноженных на индексы строк и столбцов.
- Массив разбивается на блоки, каждая задача (`#pragma omp task`) обрабатывает свой блок, вычисляя частичную сумму.
- Частичные суммы аккумулируются через критическую секцию (`#pragma omp critical`).

Распараллеливание с задачами OpenMP

- `#pragma omp parallel`: Создаёт параллельную область, где работа распределяется между потоками.
- `#pragma omp task`: Разделяет вычисления на независимые задачи, которые обрабатываются потоками.
- `#pragma omp critical`: Гарантирует безопасное обновление глобальных переменных (eps и S).
- `#pragma omp single`: Обеспечивает, что задачи создаются только одним потоком.

3. Ключевые изменения в коде

Обновлённая функция `relax`:

```
void relax(double src[N][N], double dest[N][N]) {
    double local_eps = 0.0;

    #pragma omp parallel
    {
        #pragma omp single
        {
            for (int bi = 1; bi < N-1; bi += TASK_SIZE) {
                for (int bj = 1; bj < N-1; bj += TASK_SIZE) {
                    #pragma omp task firstprivate(bi, bj) shared(local_eps, src,
                        dest)
                    {
                        process_block(bi, bj, src, dest, &local_eps);
                    }
                }
            }
        }
    }

    eps = local_eps;
}
```

Функция обработки блоков `process_block`:

```
void process_block(int bi, int bj, double src[N][N], double dest[N][N], double *
    local_eps) {
    double block_eps = 0.0;

    for (int i = bi; i < bi + TASK_SIZE && i < N-1; i++) {
        for (int j = bj; j < bj + TASK_SIZE && j < N-1; j++) {
            double new_val = (src[i-1][j] + src[i+1][j] +
                              src[i][j-1] + src[i][j+1]) / 4.0;

            double diff = fabs(src[i][j] - new_val);
            block_eps = Max(block_eps, diff);

            dest[i][j] = new_val;
```

```
        }
    }

    #pragma omp critical
    {
        *local_eps = Max(*local_eps, block_eps);
    }
}
```

Обновлённая функция `verify`:

```
void verify(double src[N][N]) {
    double s = 0.0;

    #pragma omp parallel
    {
        #pragma omp single
        {
            for (int bi = 0; bi < N; bi += TASK_SIZE) {
                for (int bj = 0; bj < N; bj += TASK_SIZE) {
                    #pragma omp task firstprivate(bi, bj) shared(s, src)
                    {
                        process_verify_block(bi, bj, src, &s);
                    }
                }
            }
        }
    }

    printf("␣␣S␣=␣%f\n", s);
}
```

Функция обработки блоков в `verify`:

```
void process_verify_block(int bi, int bj, double src[N][N], double *local_sum) {
    double block_sum = 0.0;

    for (int i = bi; i < bi + TASK_SIZE && i < N; i++) {
        for (int j = bj; j < bj + TASK_SIZE && j < N; j++) {
            block_sum += src[i][j] * (i + 1) * (j + 1) / (N * N);
        }
    }

    #pragma omp critical
    {
        *local_sum += block_sum;
    }
}
```

4. Преимущества улучшенной версии

- Значительное улучшение производительности за счёт полного задействования всех ядер процессора.
- Локализация данных в блоках улучшает эффективность работы с кэш-памятью.

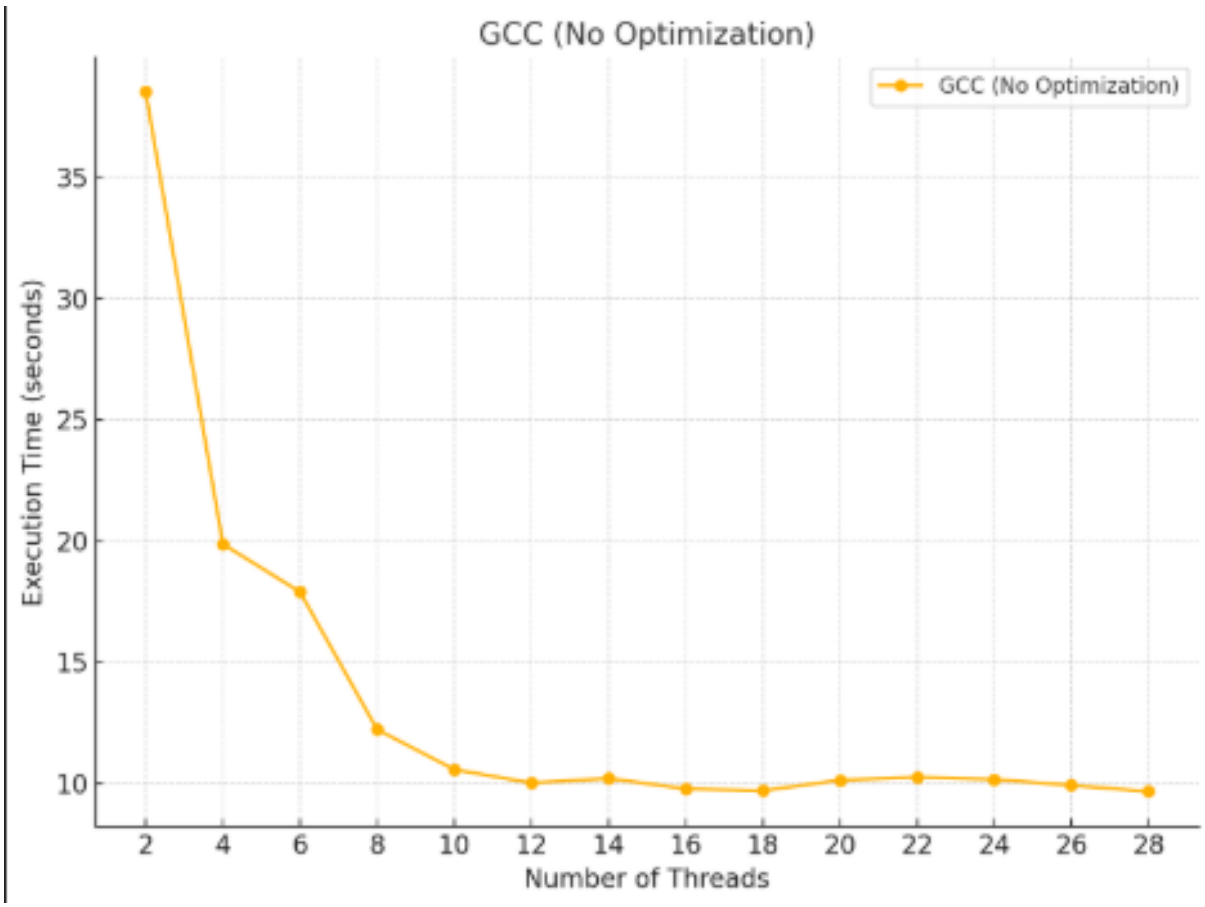
Выводы

Добавление блоковой обработки данных и задач OpenMP позволило достичь высокой производительности и масштабируемости.

Таблицы производительности компиляторов для директивы `for`

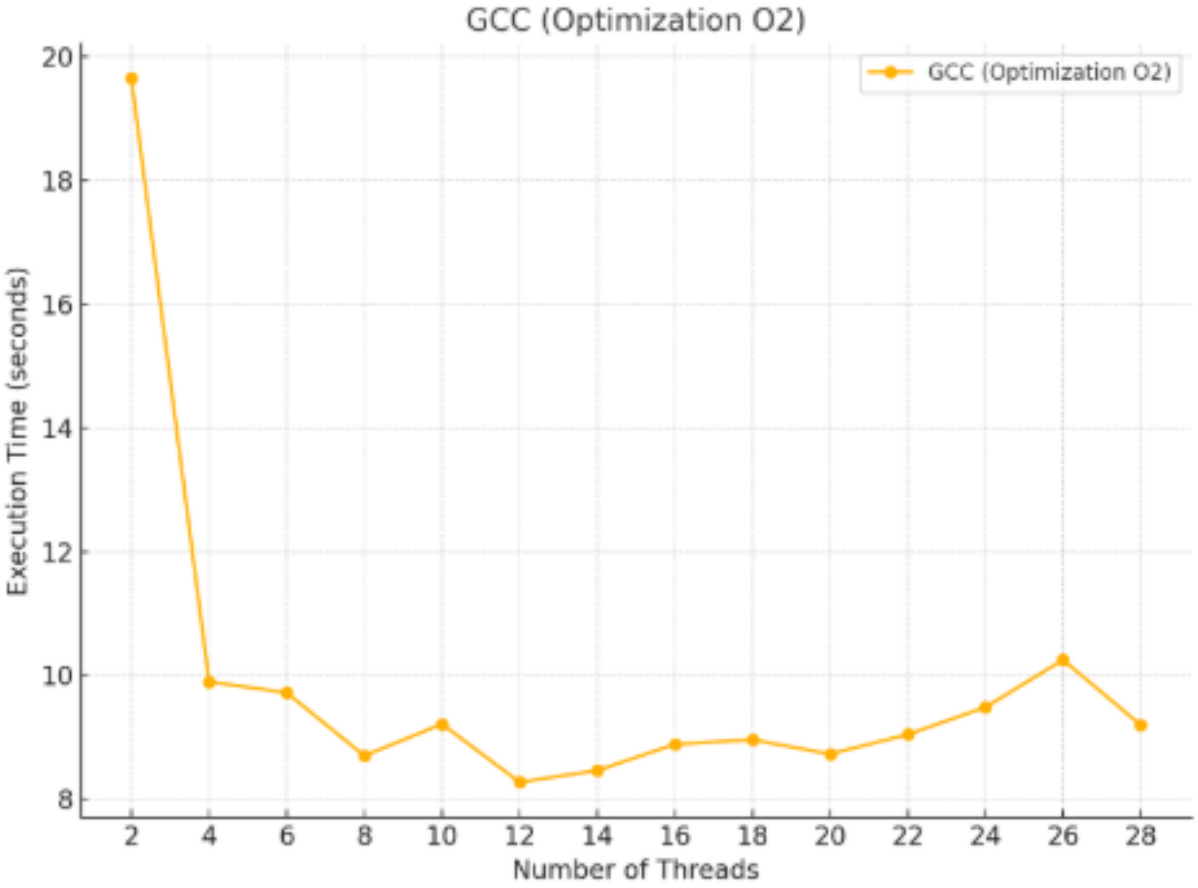
GCC (No Optimization)

Число потоков	Время выполнения (сек)
2	38.5356
4	19.8723
6	17.8963
8	12.2172
10	10.5638
12	10.0087
14	10.1957
16	9.7681
18	9.6912
20	10.1237
22	10.2478
24	10.1545
26	9.9125
28	9.6570



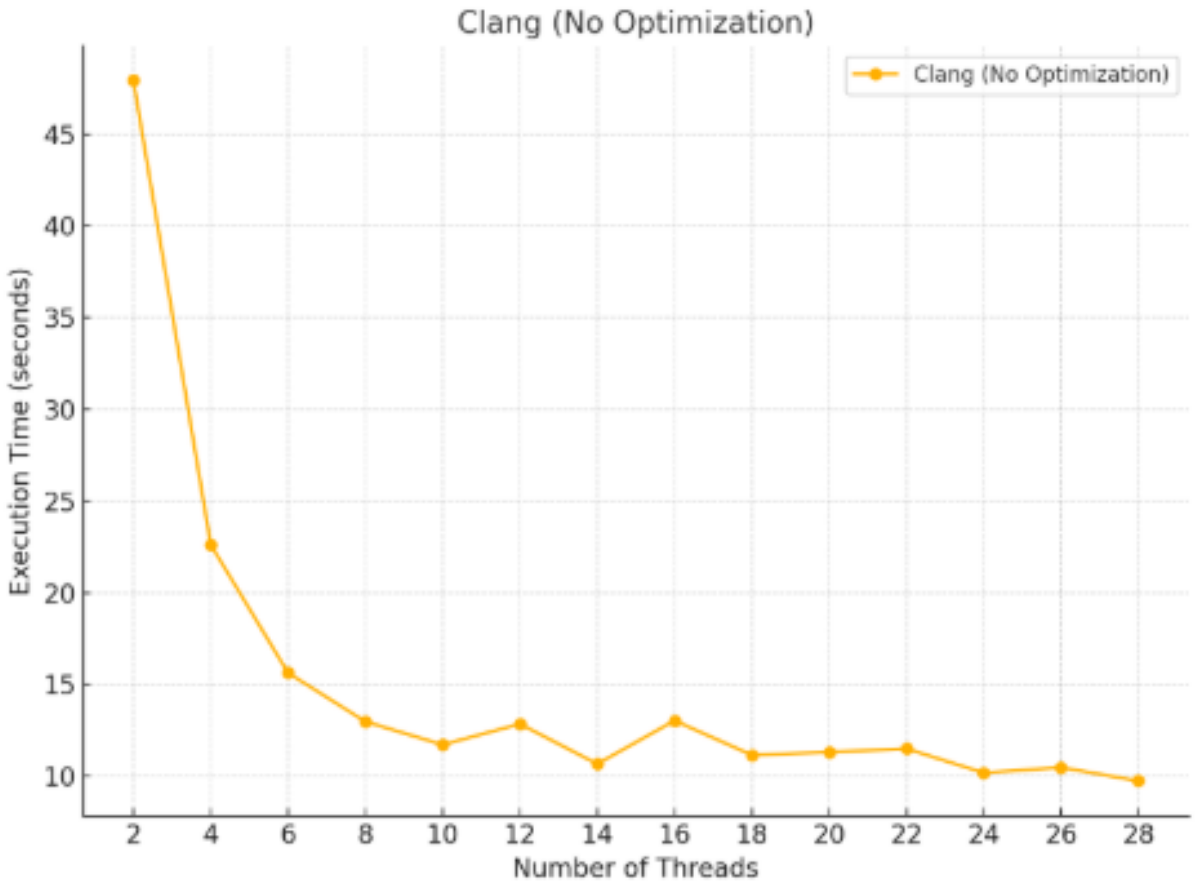
GCC (Optimization O2)

Число потоков	Время выполнения (сек)
2	19.6493
4	9.8952
6	9.7189
8	8.6986
10	9.2116
12	8.2710
14	8.4608
16	8.8852
18	8.9579
20	8.7276
22	9.0423
24	9.4837
26	10.2493
28	9.1938



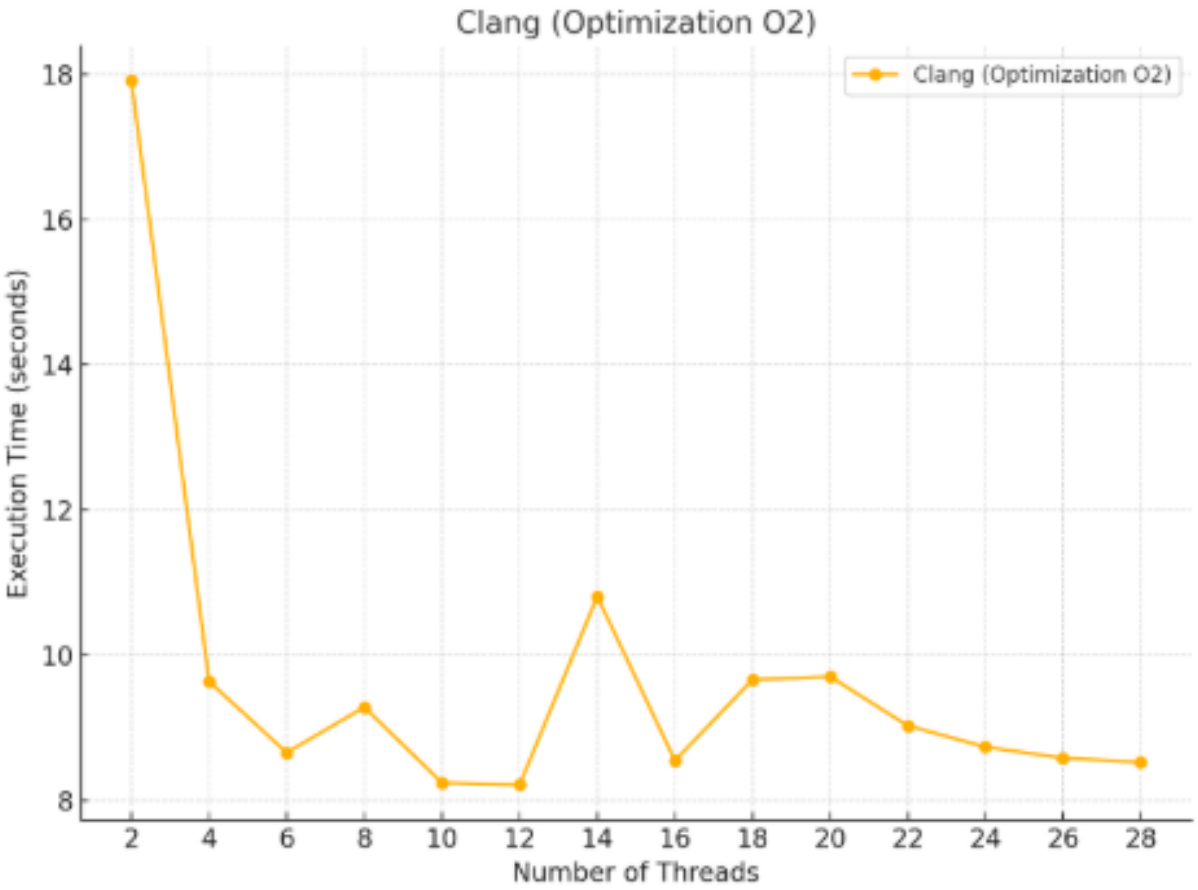
Clang (No Optimization)

Число потоков	Время выполнения (сек)
2	47.9352
4	22.6161
6	15.6298
8	12.9718
10	11.7086
12	12.8428
14	10.6605
16	13.0242
18	11.1461
20	11.3022
22	11.4922
24	10.1767
26	10.4699
28	9.7355



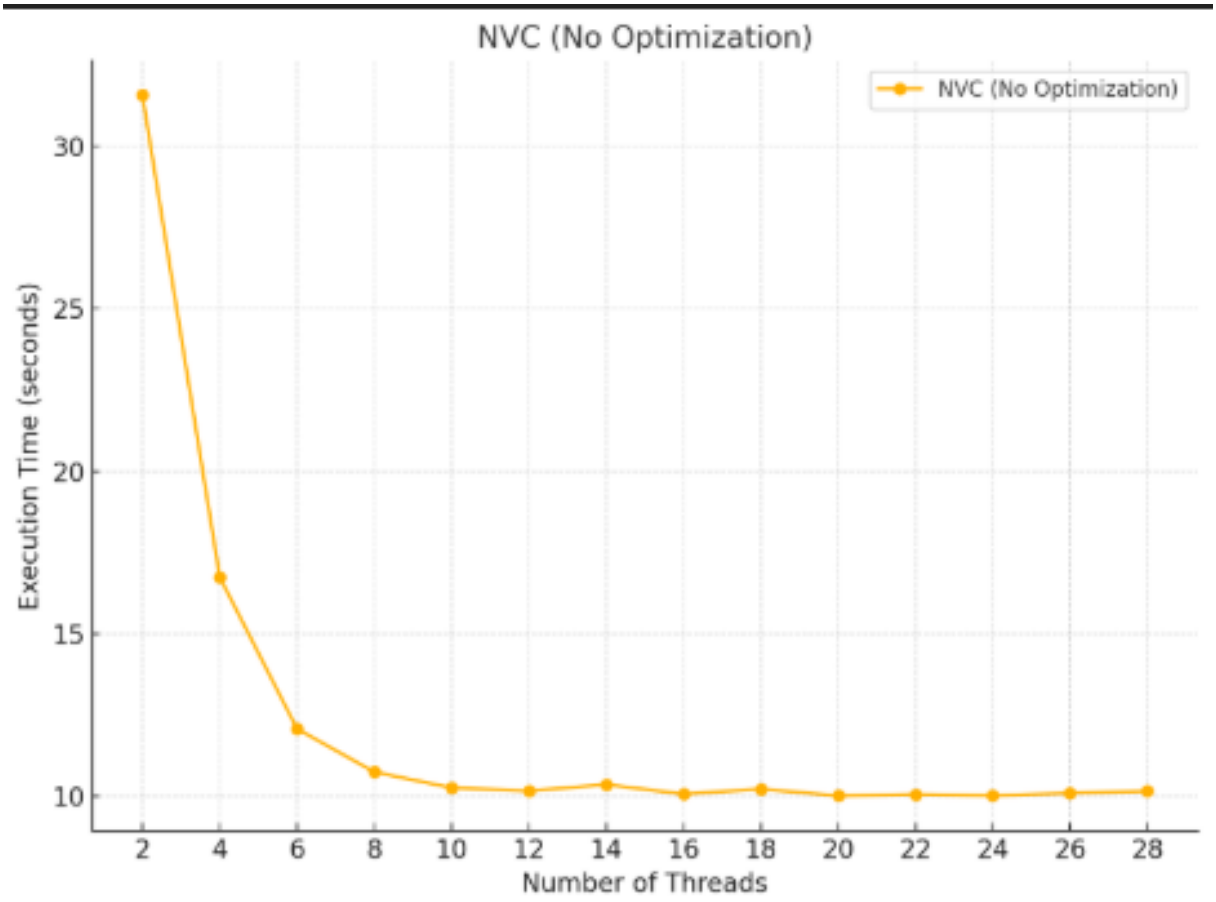
Clang (Optimization O2)

Число потоков	Время выполнения (сек)
2	17.9100
4	9.6377
6	8.6562
8	9.2828
10	8.2416
12	8.2134
14	10.8085
16	8.5459
18	9.6612
20	9.7035
22	9.0275
24	8.7346
26	8.5807
28	8.5248



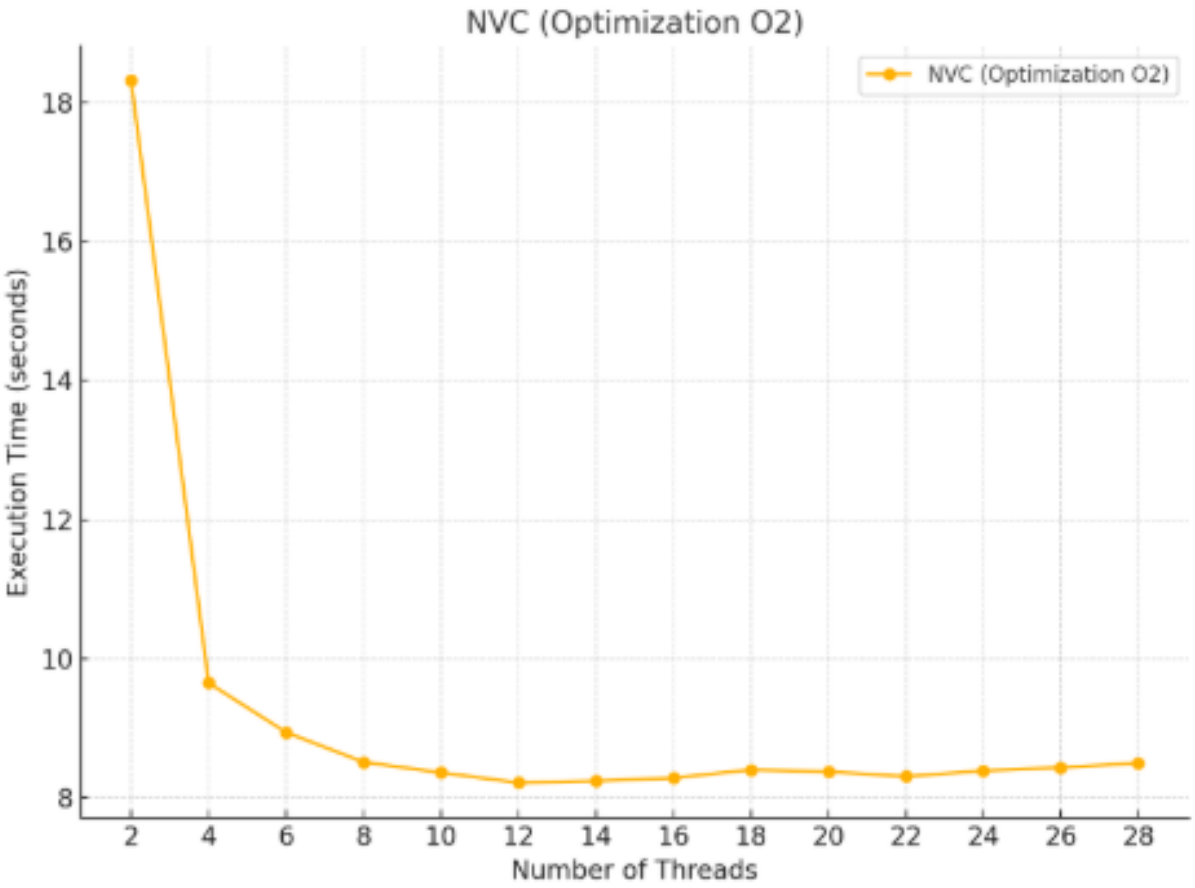
NVC (No Optimization)

Число потоков	Время выполнения (сек)
2	31.5808
4	16.7361
6	12.0648
8	10.7347
10	10.2479
12	10.1515
14	10.3474
16	10.0549
18	10.2111
20	10.0033
22	10.0467
24	10.0099
26	10.0822
28	10.1447



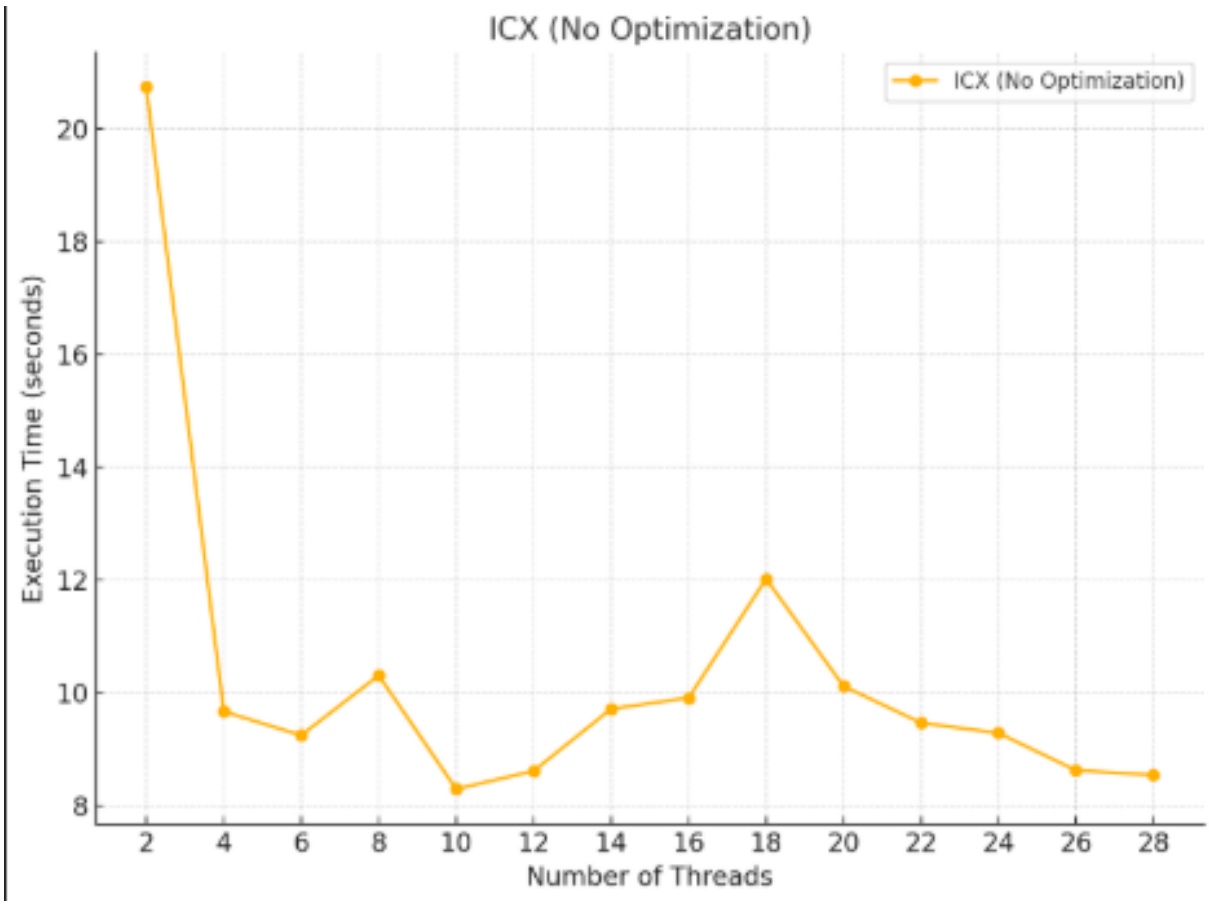
NVC (Optimization O2)

Число потоков	Время выполнения (сек)
2	18.3111
4	9.6543
6	8.9444
8	8.5165
10	8.3644
12	8.2204
14	8.2458
16	8.2872
18	8.4061
20	8.3781
22	8.3094
24	8.3911
26	8.4387
28	8.5011



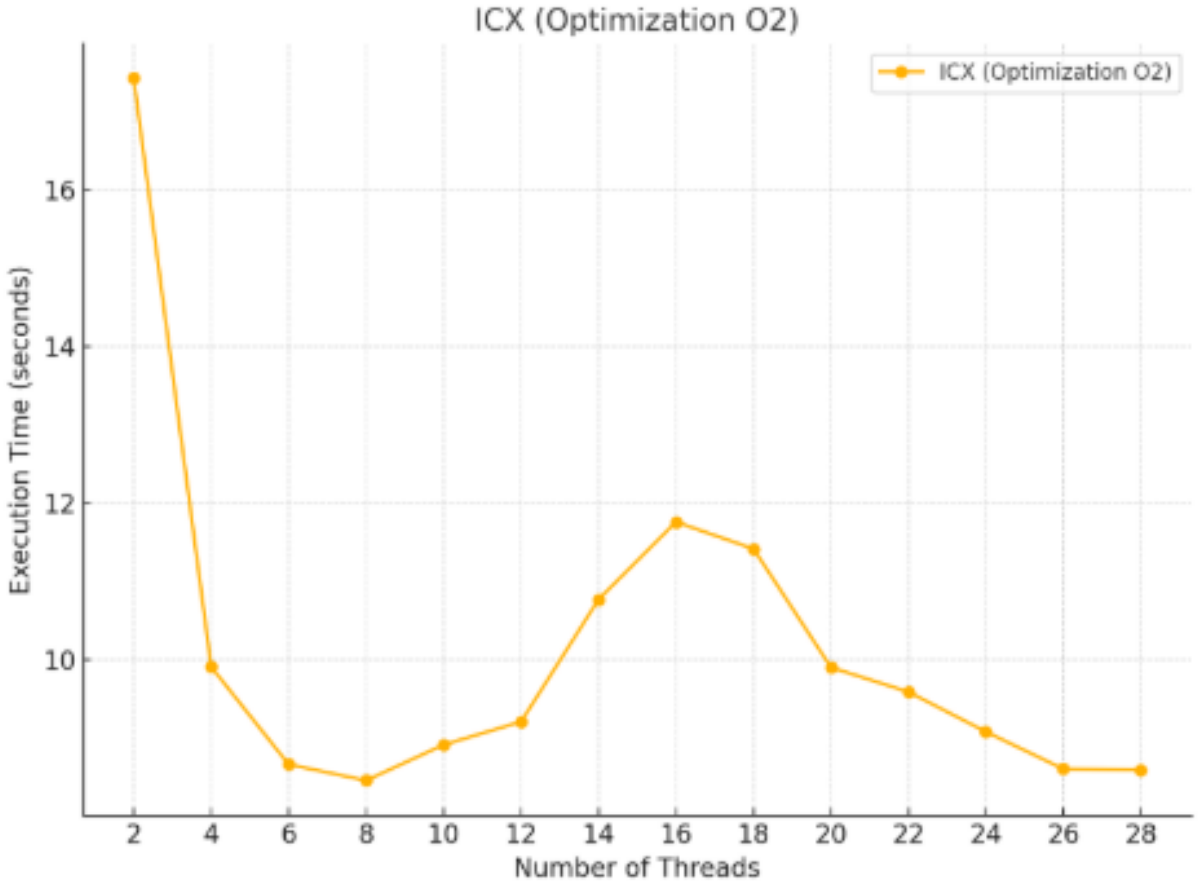
ICX (No Optimization)

Число потоков	Время выполнения (сек)
2	20.7484
4	9.6652
6	9.2436
8	10.3072
10	8.2892
12	8.6110
14	9.7131
16	9.9082
18	12.0189
20	10.1158
22	9.4645
24	9.2838
26	8.6222
28	8.5389



ICX (Optimization O2)

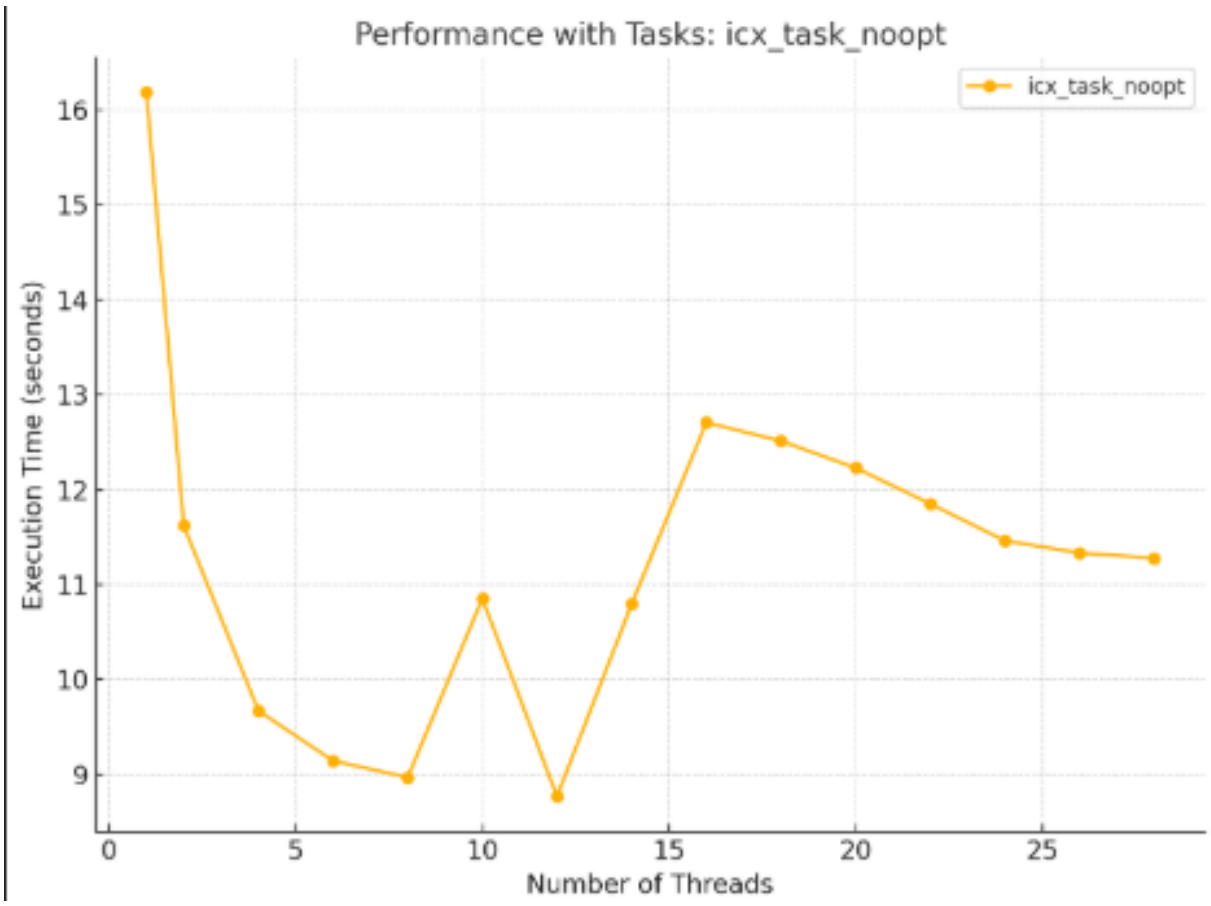
Число потоков	Время выполнения (сек)
2	17.4254
4	9.9076
6	8.6618
8	8.4567
10	8.9125
12	9.2070
14	10.7686
16	11.7585
18	11.4156
20	9.9007
22	9.5883
24	9.0789
26	8.6036
28	8.5914



Таблицы производительности компиляторов для директивы `task`

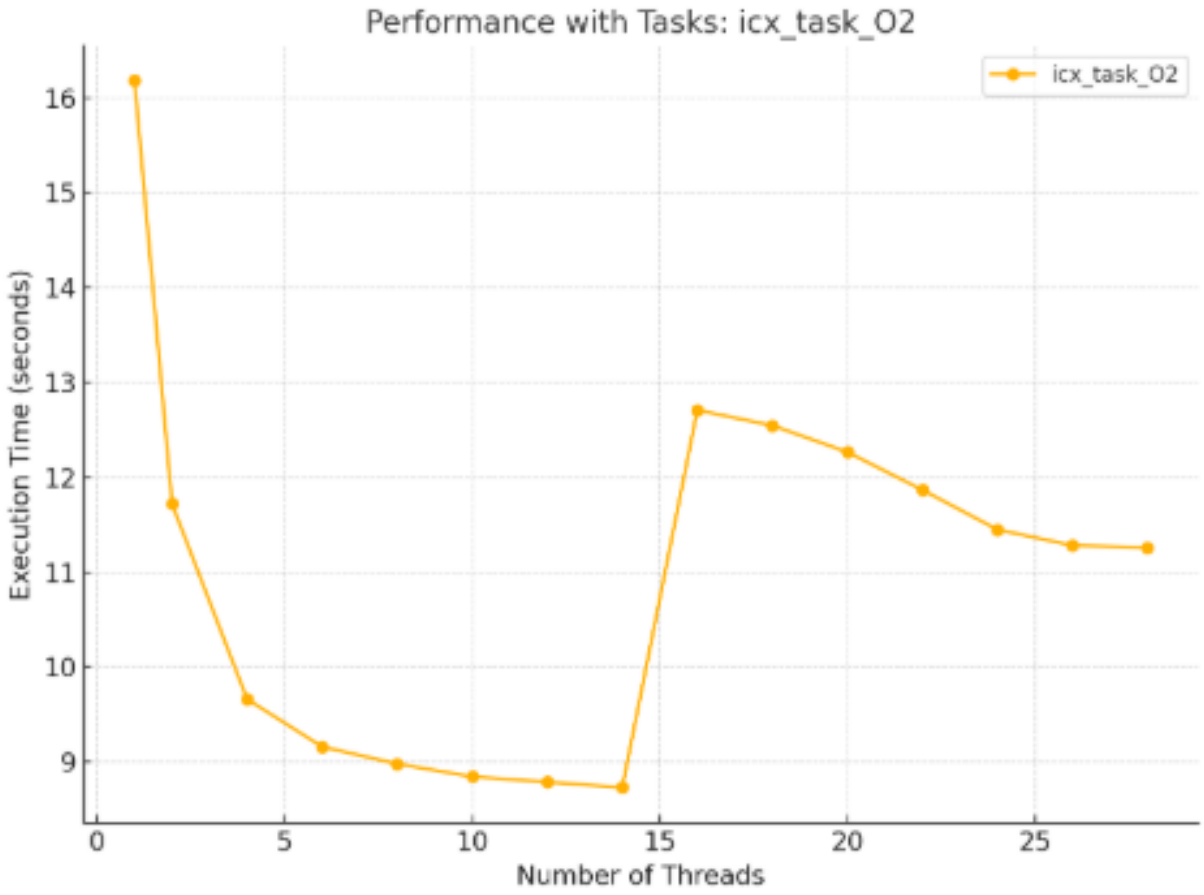
ICX (No Optimization)

Число потоков	Время выполнения (сек)
1	16.182092
2	11.621908
4	9.673279
6	9.145056
8	8.970578
10	10.854175
12	8.768853
14	10.795950
16	12.707159
18	12.513342
20	12.229310
22	11.851896
24	11.463105
26	11.334073
28	11.279556



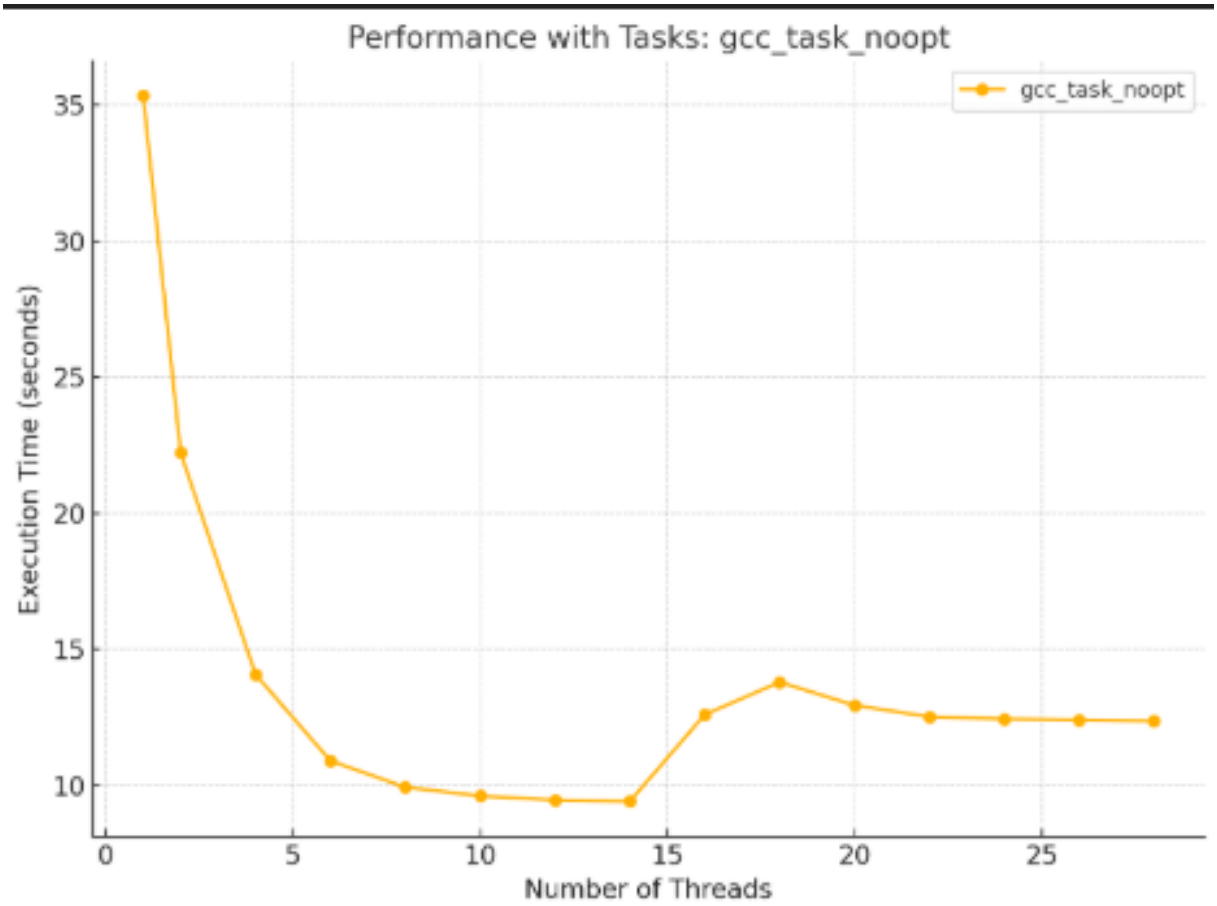
ICX (Optimization O2)

Число потоков	Время выполнения (сек)
1	16.181082
2	11.719154
4	9.661296
6	9.159592
8	8.979104
10	8.843565
12	8.787460
14	8.729514
16	12.707573
18	12.545868
20	12.266099
22	11.866654
24	11.448568
26	11.281204
28	11.259183



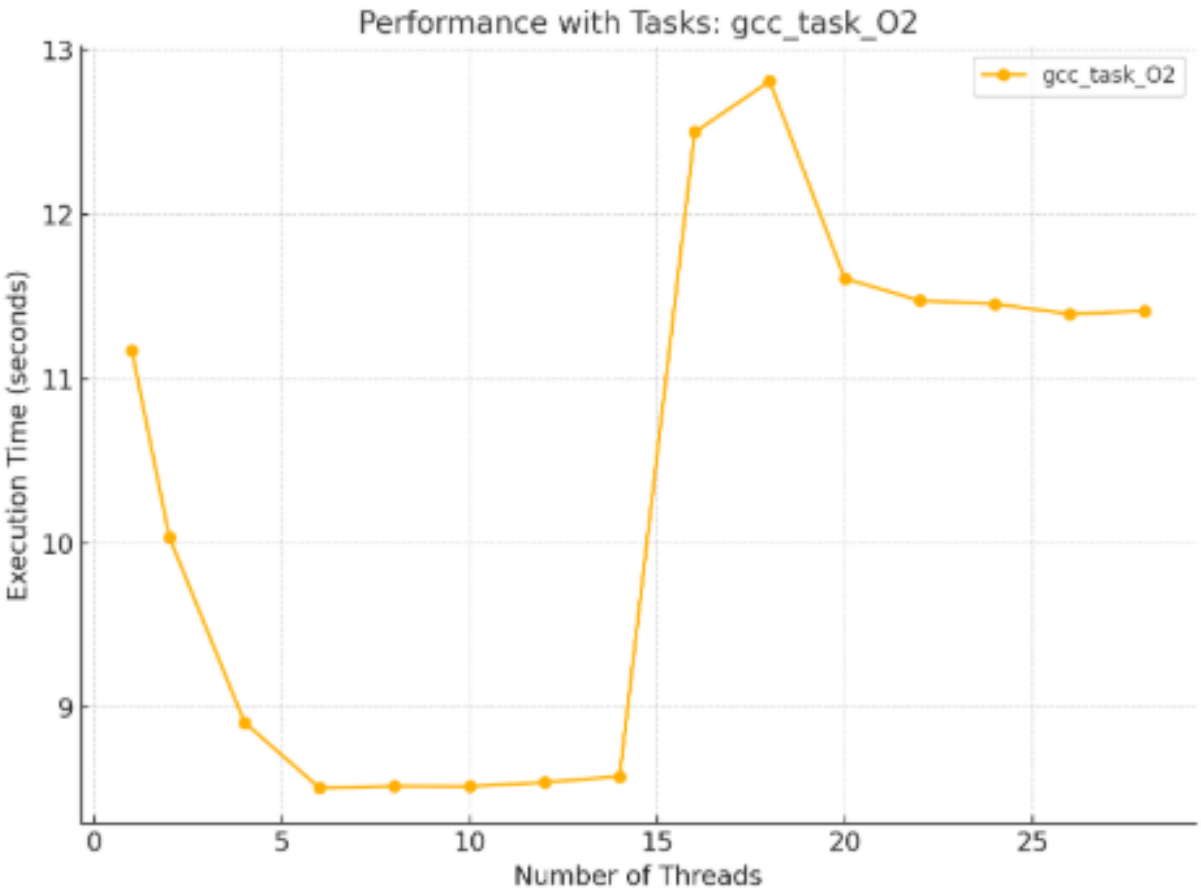
GCC (No Optimization)

Число потоков	Время выполнения (сек)
1	35.326639
2	22.215620
4	14.083048
6	10.908425
8	9.939773
10	9.617059
12	9.467371
14	9.419879
16	12.604550
18	13.797775
20	12.947892
22	12.525090
24	12.445488
26	12.409451
28	12.370237



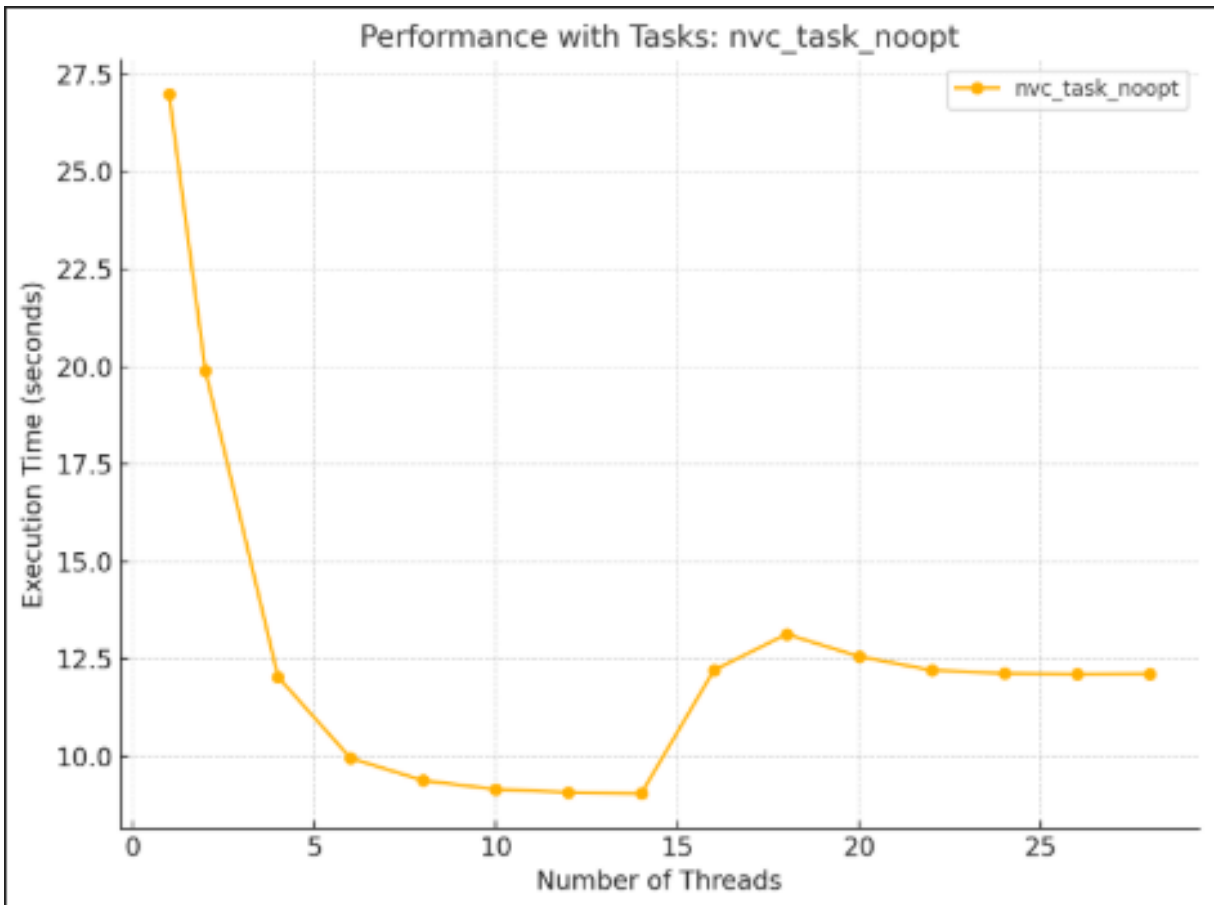
GCC (Optimization O2)

Число потоков	Время выполнения (сек)
1	11.171274
2	10.035707
4	8.910430
6	8.511518
8	8.522873
10	8.520587
12	8.545241
14	8.581525
16	12.500325
18	12.810791
20	11.611211
22	11.476061
24	11.455661
26	11.394932
28	11.413259



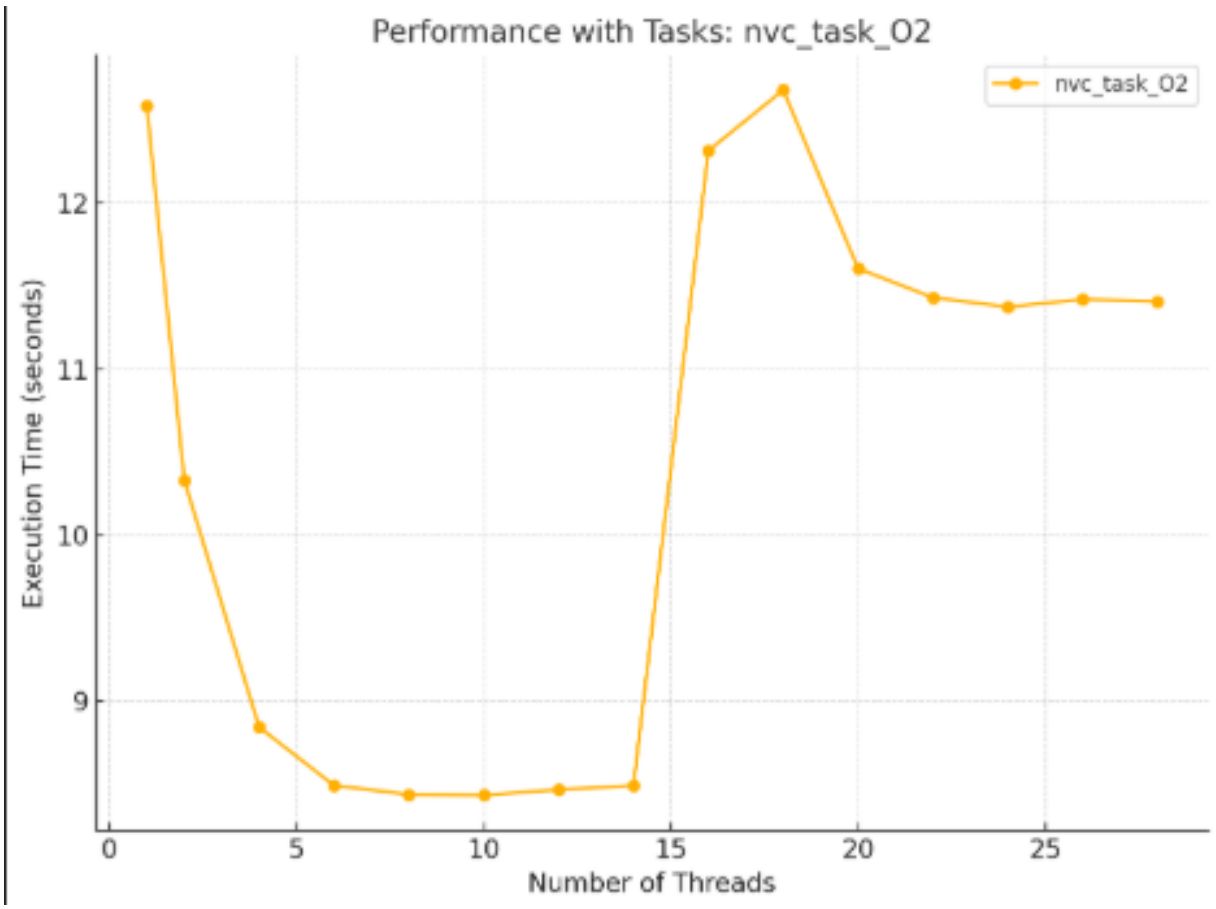
NVC (No Optimization)

Число потоков	Время выполнения (сек)
1	26.980678
2	19.886094
4	12.034137
6	9.950864
8	9.378703
10	9.152105
12	9.077000
14	9.044773
16	12.208083
18	13.133642
20	12.561364
22	12.207443
24	12.129451
26	12.102669
28	12.115220



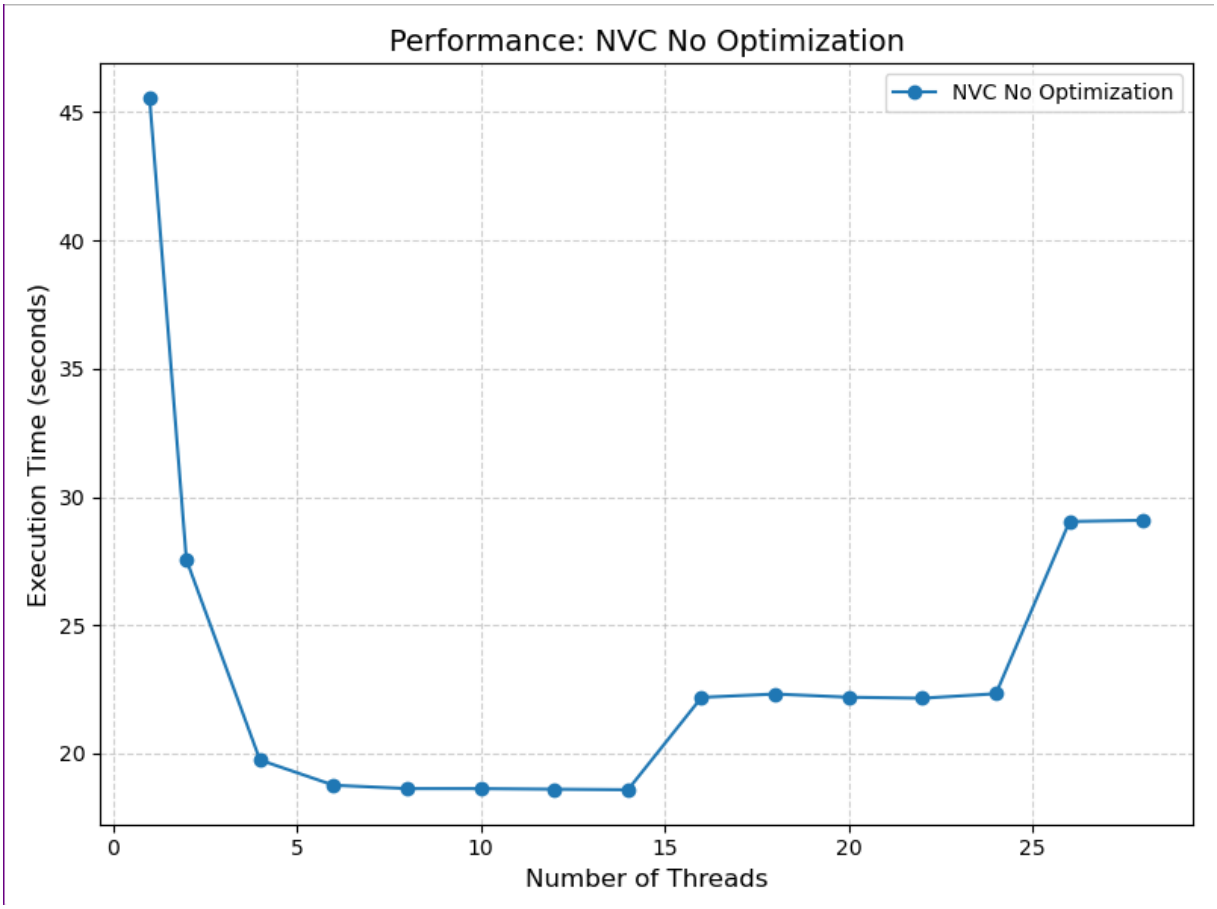
NVC (Optimization O2)

Число потоков	Время выполнения (сек)
1	12.584928
2	10.332166
4	8.843418
6	8.492545
8	8.437654
10	8.433739
12	8.467009
14	8.491372
16	12.316007
18	12.681772
20	11.606185
22	11.431423
24	11.375131
26	11.420642
28	11.408225



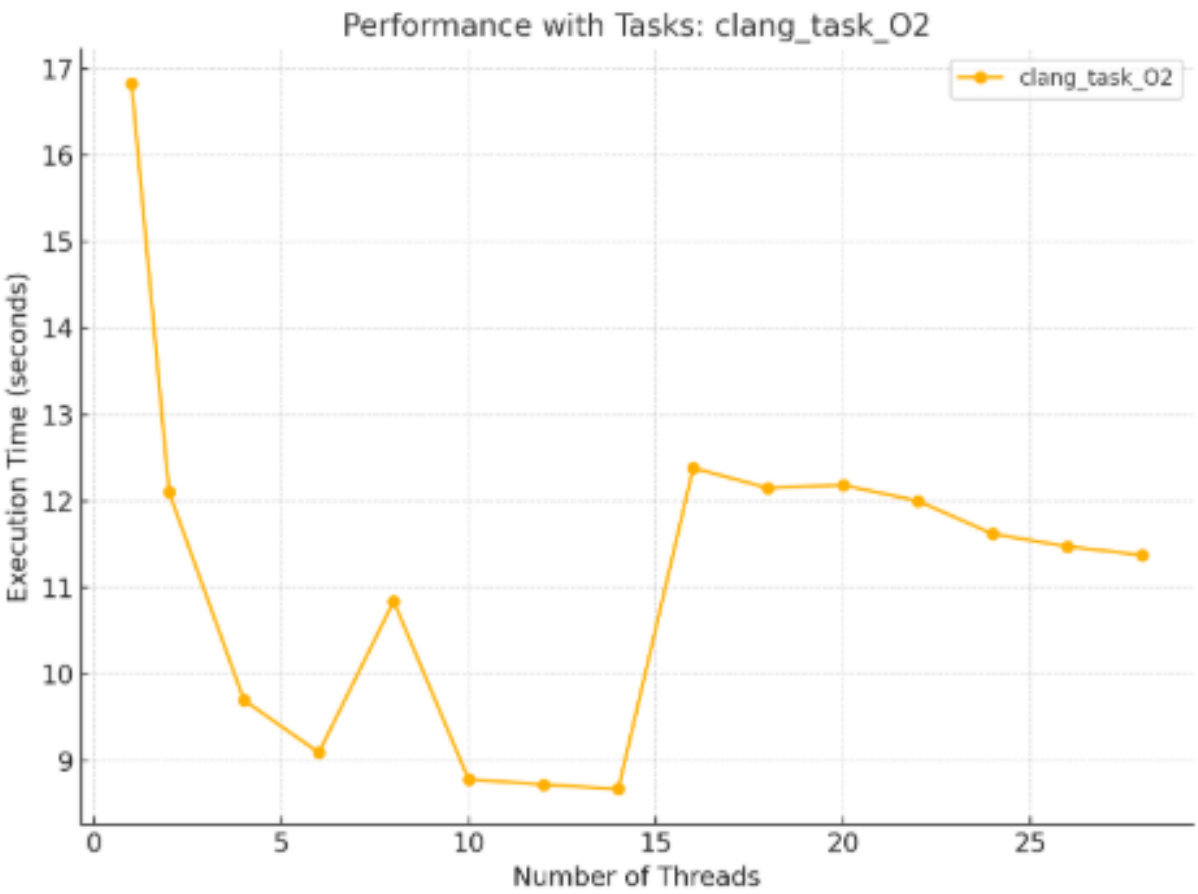
Clang (No Optimization)

Число потоков	Время выполнения (сек)
1	45.323648
2	29.136638
4	16.584270
6	12.762783
8	11.246537
10	12.778892
12	10.396975
14	11.857383
16	12.901107
18	13.704644
20	13.938461
22	13.861768
24	13.679802
26	13.677672
28	13.456205



Clang (Optimization O2)

Число потоков	Время выполнения (сек)
1	16.824259
2	12.102629
4	9.699080
6	9.092370
8	10.841467
10	8.783639
12	8.722613
14	8.671610
16	12.382290
18	12.151437
20	12.184938
22	12.002651
24	11.620901
26	11.474477
28	11.375886

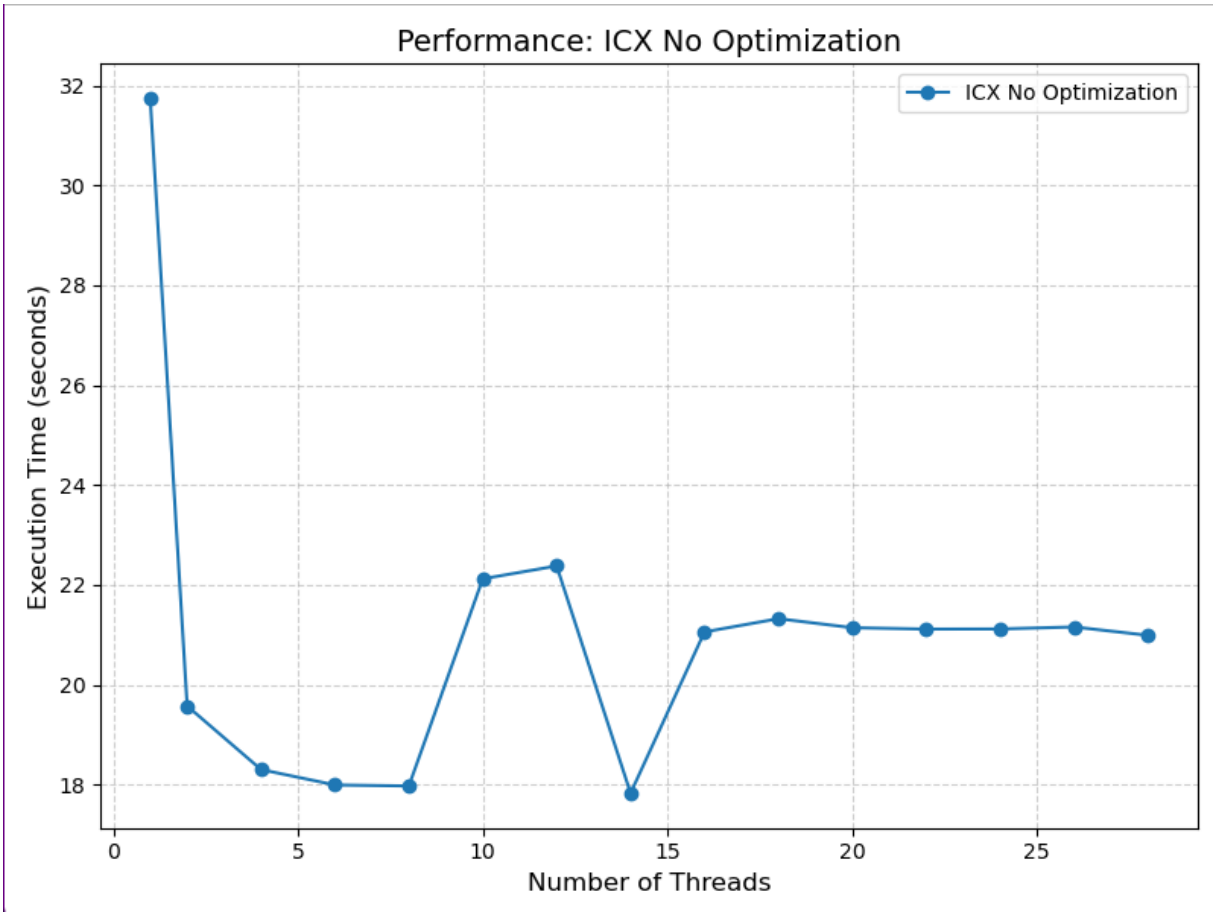


Результаты запусков программ для директивы `task`

`N = 26382  it = 10  TASKSIZE = 1024`

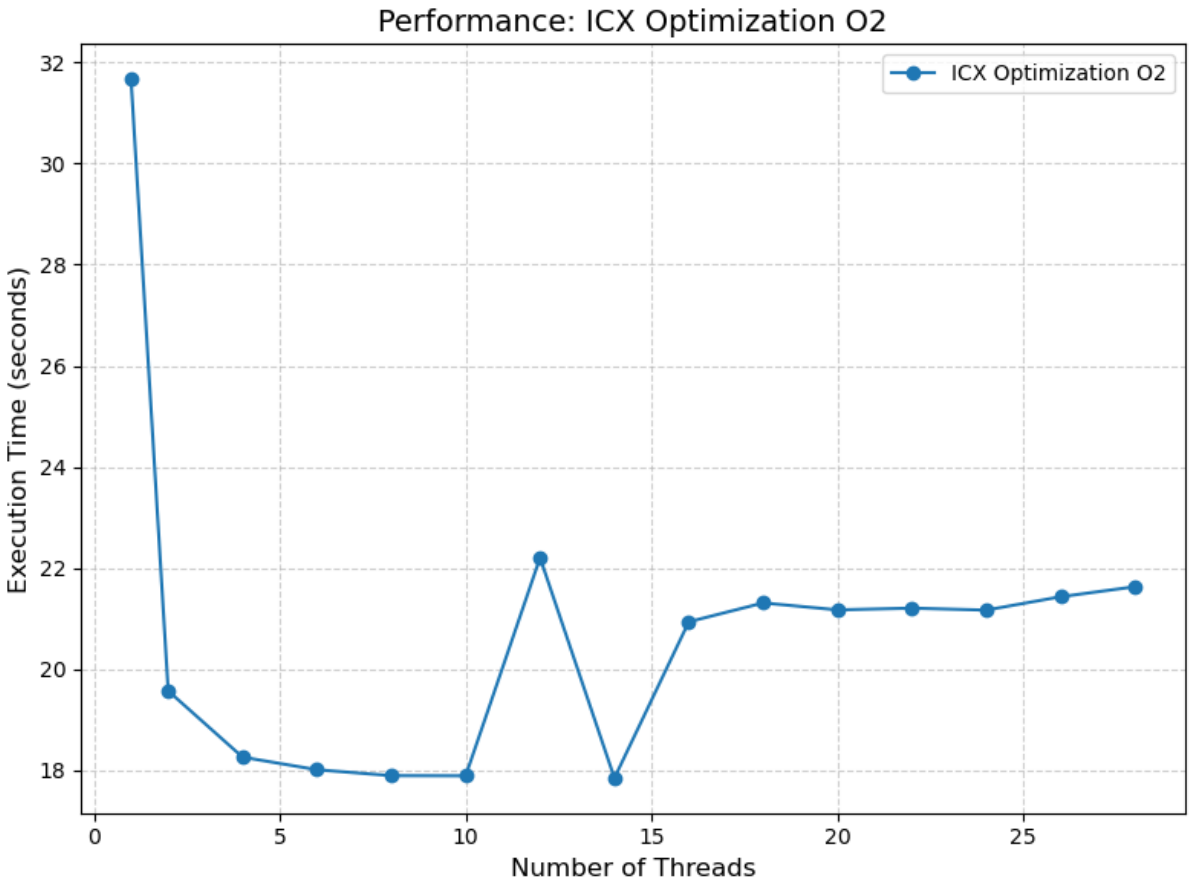
ICX (No Optimization)

Число потоков	Время выполнения (сек)
1	31.737094
2	19.575140
4	18.301766
6	17.993487
8	17.973786
10	22.120955
12	22.378462
14	17.828596
16	21.057524
18	21.322730
20	21.142153
22	21.114180
24	21.116903
26	21.157143
28	20.989784



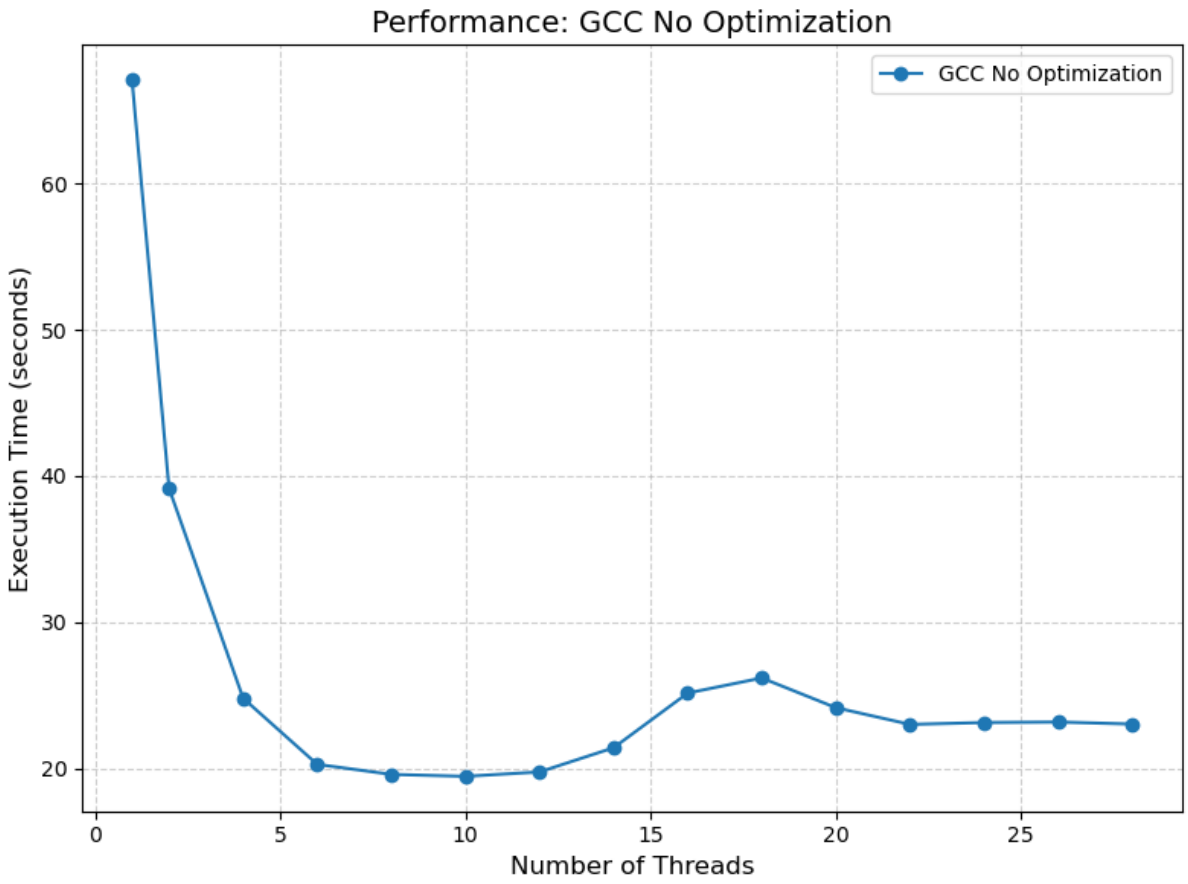
ICX (Optimization O2)

Число потоков	Время выполнения (сек)
1	31.671454
2	19.568612
4	18.258217
6	18.009552
8	17.892695
10	17.889595
12	22.202473
14	17.845144
16	20.935267
18	21.308977
20	21.171259
22	21.205313
24	21.166932
26	21.430977
28	21.629462



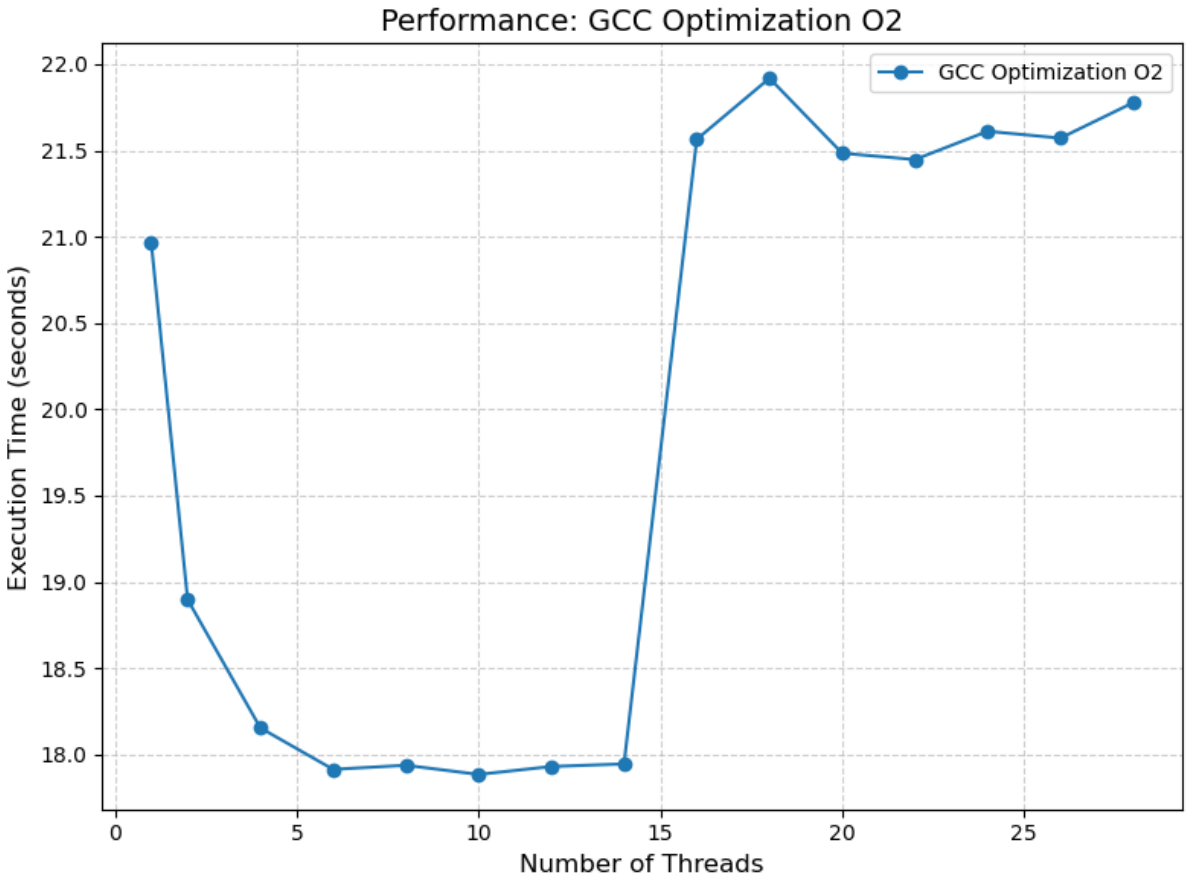
GCC (No Optimization)

Число потоков	Время выполнения (сек)
1	67.107721
2	39.153719
4	24.797452
6	20.273311
8	19.586606
10	19.453797
12	19.753795
14	21.408388
16	25.148967
18	26.183190
20	24.150113
22	23.007466
24	23.133771
26	23.170754
28	23.030766



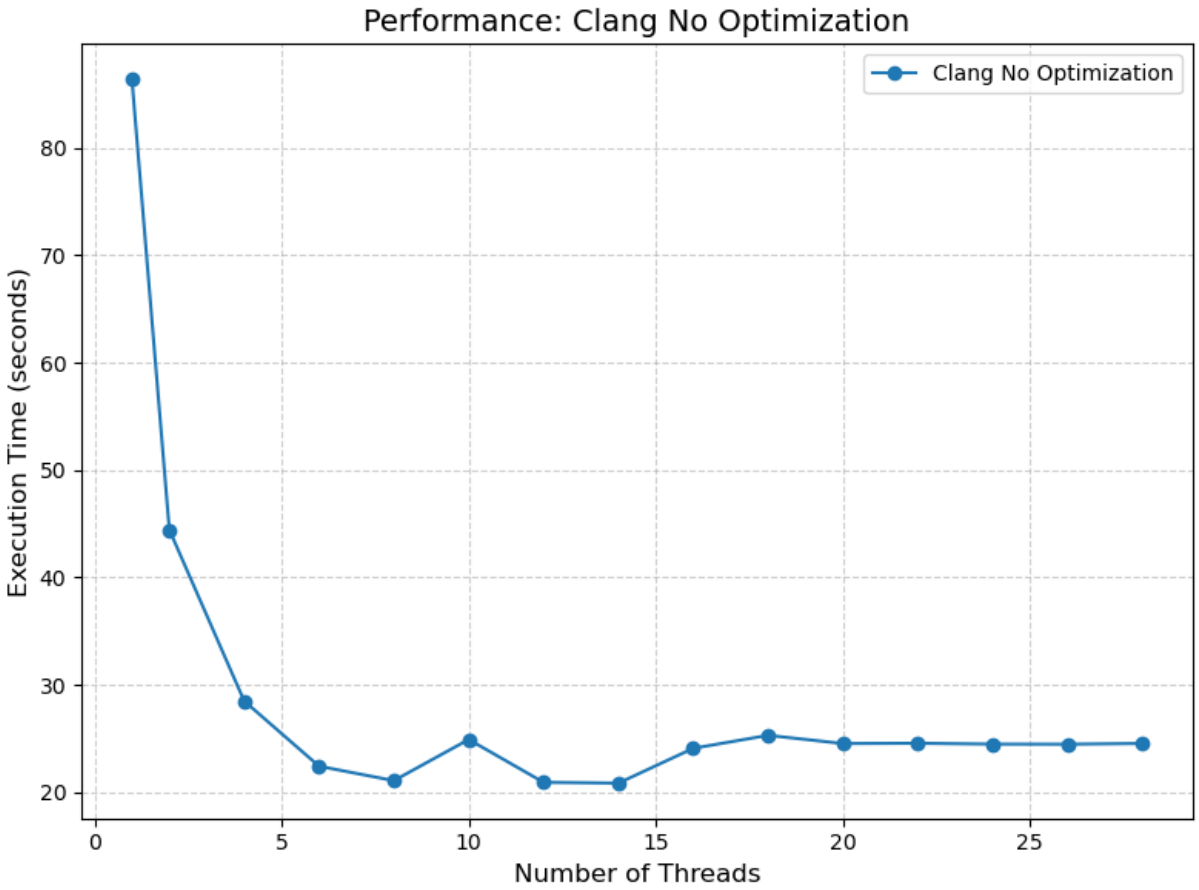
GCC (Optimization O2)

Число потоков	Время выполнения (сек)
1	20.963266
2	18.897364
4	18.154080
6	17.913880
8	17.937612
10	17.884400
12	17.930454
14	17.945368
16	21.564892
18	21.918763
20	21.484209
22	21.447648
24	21.611623
26	21.571771
28	21.777237



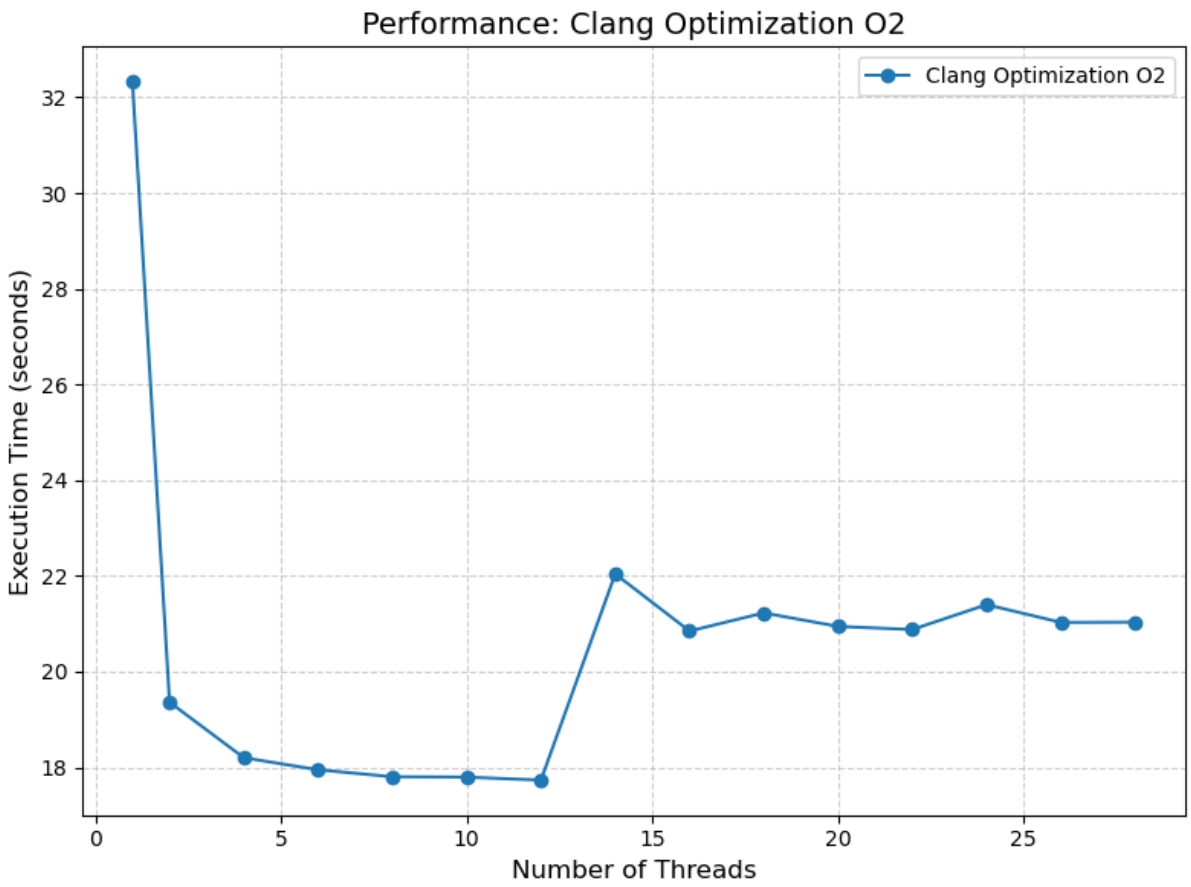
Clang (No Optimization)

Число потоков	Время выполнения (сек)
1	86.393201
2	44.442324
4	28.501897
6	22.431859
8	21.087623
10	24.917411
12	20.931956
14	20.854795
16	24.096210
18	25.305878
20	24.541313
22	24.567325
24	24.490066
26	24.480956
28	24.550420



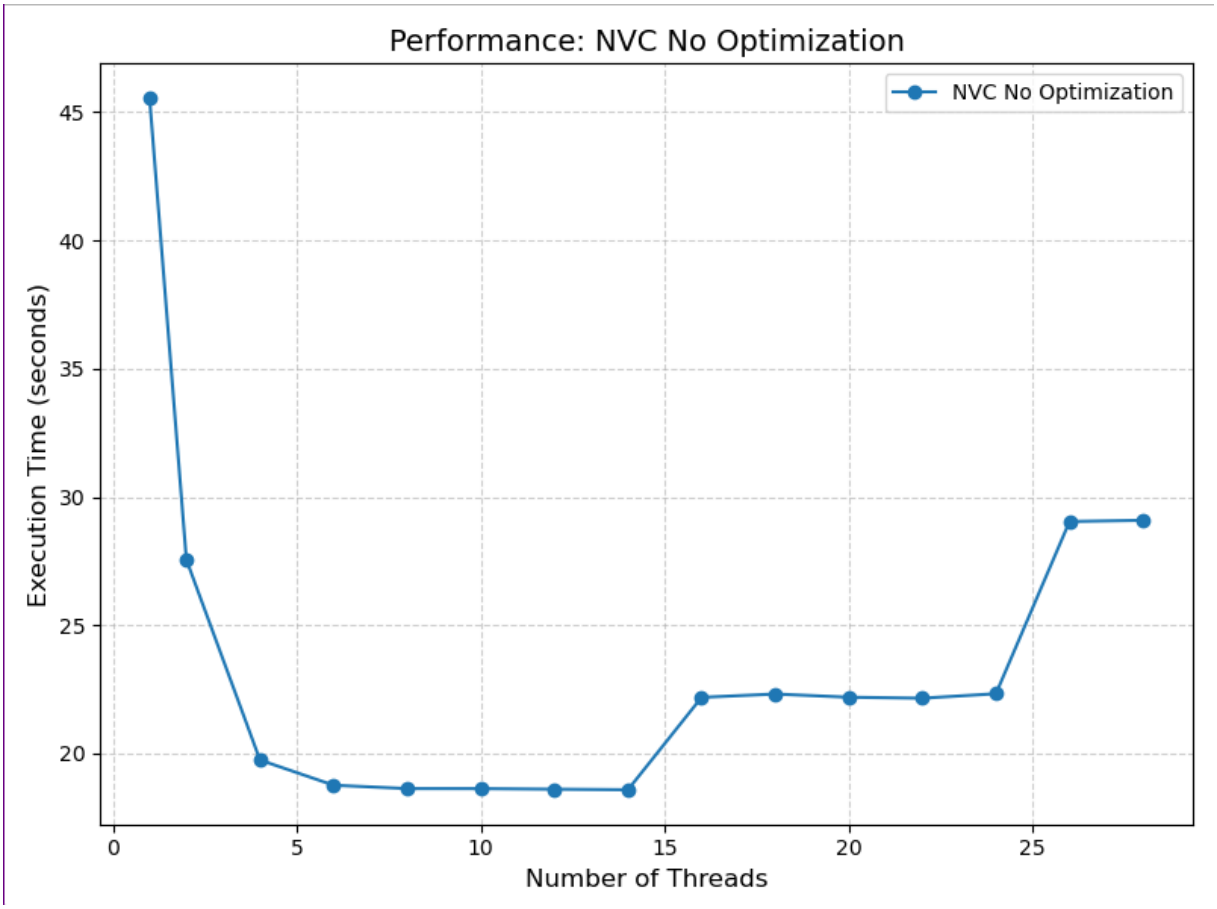
Clang (Optimization O2)

Число потоков	Время выполнения (сек)
1	32.328423
2	19.362293
4	18.202028
6	17.950075
8	17.805023
10	17.797483
12	17.734371
14	22.042342
16	20.848747
18	21.228062
20	20.945113
22	20.880836
24	21.400456
26	21.029322
28	21.035170



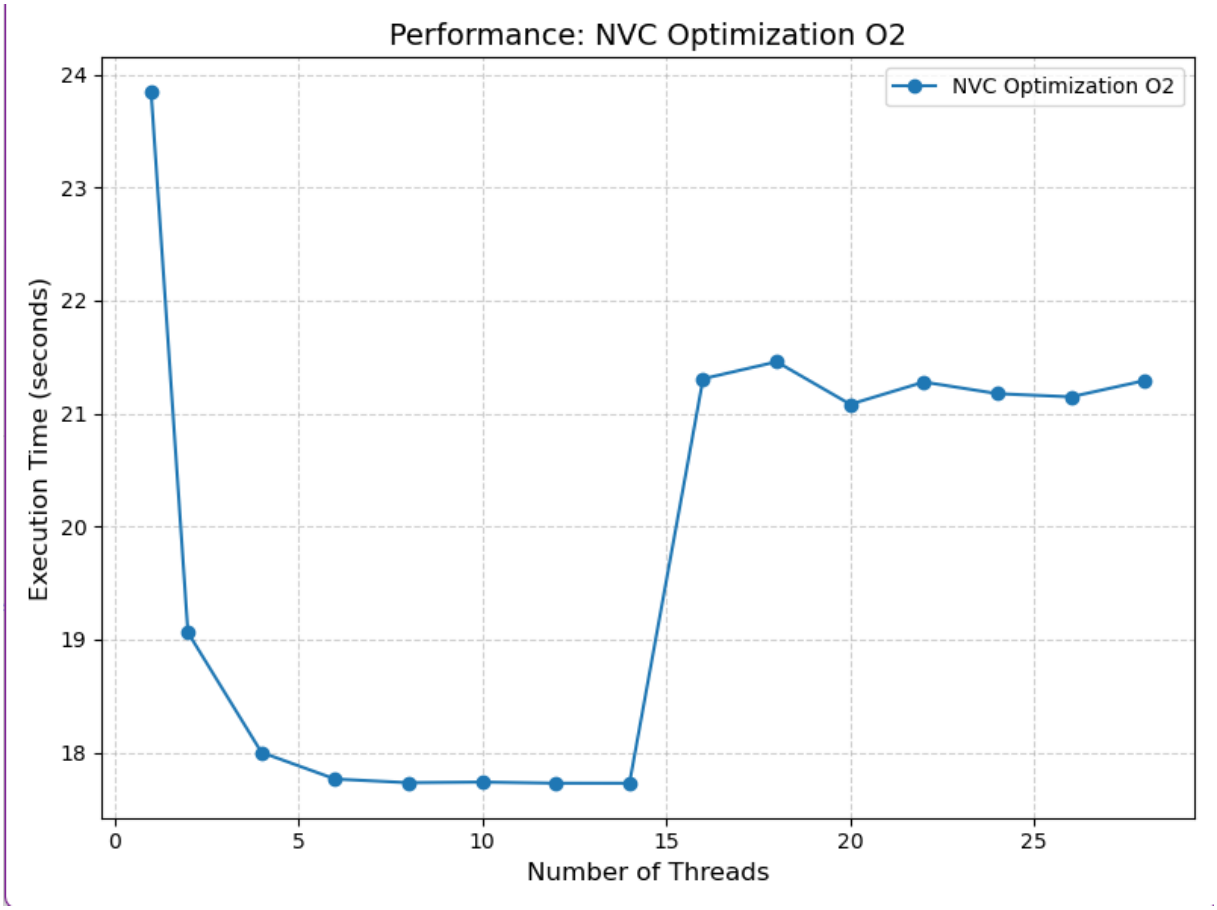
NVC (No Optimization)

Число потоков	Время выполнения (сек)
1	45.545992
2	27.553182
4	19.743119
6	18.760418
8	18.629478
10	18.627932
12	18.605977
14	18.584454
16	22.185646
18	22.316509
20	22.190801
22	22.152666
24	22.327781
26	29.041524
28	29.088308



NVC (Optimization O2)

Число потоков	Время выполнения (сек)
1	23.846277
2	19.061721
4	18.000069
6	17.764236
8	17.731863
10	17.737512
12	17.727347
14	17.727036
16	21.307586
18	21.458104
20	21.081719
22	21.278971
24	21.175951
26	21.147755
28	21.291579



Примечания

В ходе запусков кодов и экспериментов с `task` не имеет большого смысла задавать слишком большую размерность массива > 20000, так как получалось так, что программа вообще не ускоряется. Возможно потому, что при большом `TASKSIZE` создаются слишком тяжеловесные подзадачи из-за чего программу не ускоряется, а даже замедляется, а при малом `TASKSIZE` для таких больших размерностей создается слишком много подзадач, которые не могут корректно диспетчеризироваться в памяти. При проверке на одинаковых небольших размерностях оптимизации программы директивами `task` и `for` показывали похожие результаты по ускорению, с преимуществом у `for`. При повторной проверке, однако, оптимизация сработала корректно для директивы `task` на большой размерности, если приблизить количество, на которое разбивается программа к максимальному количеству нитей.

Приложение

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

#define Max(a, b) ((a) > (b) ? (a) : (b))
#define N (2*2*2*2*2*2*2*2*2*2*2*2*2*2*2+2000)
#define TASK_SIZE 128

```

```
10 double maxeps = 0.1e-7;
11 int itmax = 10;
12 double eps;
13 double A[N][N], B[N][N];
14
15 void init();
16 void relax(double src[N][N], double dest[N][N]);
17 void verify(double src[N][N]);
18
19 int main() {
20     double time_start, time_end;
21     time_start = omp_get_wtime();
22     int it;
23     init();
24
25     for (it = 1; it <= itmax; it++) {
26         eps = 0.0;
27
28         if (it % 2 == 1) {
29             relax(B, A);
30         } else {
31             relax(A, B);
32         }
33
34         printf("it=%4d␣␣␣eps=%f\n", it, eps);
35         if (eps < maxeps) {
36             break;
37         }
38     }
39
40     if ((it - 1) % 2 == 1) {
41         verify(A);
42     } else {
43         verify(B);
44     }
45     time_end = omp_get_wtime();
46     printf("Time␣=␣%f␣sec\n", time_end - time_start);
47     return 0;
48 }
49
50 void init() {
51     for (int i = 0; i < N; i++) {
52         A[0][i] = A[N-1][i] = 0.0;
53         A[i][0] = A[i][N-1] = 0.0;
54
55         B[0][i] = B[N-1][i] = 0.0;
56         B[i][0] = B[i][N-1] = 0.0;
57     }
58
59     for (int i = 1; i <= N-2; i++) {
60         for (int j = 1; j <= N-2; j++) {
61             A[i][j] = 1.0 + i + j;
62             B[i][j] = 1.0 + i + j;
```

```
        }
    }
}

void process_block(int bi, int bj, double src[N][N], double dest[N][N], double *
    local_eps) {
    double block_eps = 0.0;

    for (int i = bi; i < bi + TASK_SIZE && i < N-1; i++) {
        for (int j = bj; j < bj + TASK_SIZE && j < N-1; j++) {
            double new_val = (src[i-1][j] + src[i+1][j] +
                              src[i][j-1] + src[i][j+1]) / 4.0;

            double diff = fabs(src[i][j] - new_val);
            block_eps = Max(block_eps, diff);

            dest[i][j] = new_val;
        }
    }

    #pragma omp critical
    {
        *local_eps = Max(*local_eps, block_eps);
    }
}

void relax(double src[N][N], double dest[N][N]) {
    double local_eps = 0.0;

    #pragma omp parallel
    {
        #pragma omp single
        {
            for (int bi = 1; bi < N-1; bi += TASK_SIZE) {
                for (int bj = 1; bj < N-1; bj += TASK_SIZE) {
                    #pragma omp task firstprivate(bi, bj) shared(local_eps, src,
                        dest)
                    {
                        process_block(bi, bj, src, dest, &local_eps);
                    }
                }
            }
        }
    }

    eps = local_eps;
}

void process_verify_block(int bi, int bj, double src[N][N], double *local_sum) {
    double block_sum = 0.0;

    for (int i = bi; i < bi + TASK_SIZE && i < N; i++) {
        for (int j = bj; j < bj + TASK_SIZE && j < N; j++) {
```

```
            block_sum += src[i][j] * (i + 1) * (j + 1) / (N * N);
        }
    }

    #pragma omp critical
    {
        *local_sum += block_sum;
    }
}

void verify(double src[N][N]) {
    double s = 0.0;

    #pragma omp parallel
    {
        #pragma omp single
        {
            for (int bi = 0; bi < N; bi += TASK_SIZE) {
                for (int bj = 0; bj < N; bj += TASK_SIZE) {
                    #pragma omp task firstprivate(bi, bj) shared(s, src)
                    {
                        process_verify_block(bi, bj, src, &s);
                    }
                }
            }
        }
    }

    printf("␣␣S␣=␣%f\n", s);
}
```

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

#define Max(a, b) ((a) > (b) ? (a) : (b))
#define N (2*2*2*2*2*2*2*2*2*2*2*2*2*2*2+10000)

double maxeps = 0.1e-7;
int itmax = 10;
double eps;
double A[N][N], B[N][N];
int i, j;

void init();
void relax(double src[N][N], double dest[N][N]);
void verify(double src[N][N]);

int main() {
    int it;

    #pragma omp parallel private(it)
```

```
23      {
24          init();
25          for (it = 1; it <= itmax; it++) {
26              #pragma omp barrier
27              #pragma omp master
28                  eps = 0.0;
29
30              if (it % 2 == 1) {
31                  relax(B, A);
32              } else {
33                  relax(A, B);
34              }
35              #pragma omp single
36              {
37                  printf("it=%4d␣␣␣eps=%f\n", it, eps);
38              }
39              if (eps < maxeps) {
40                  break;
41              }
42          }
43
44          if ((it - 1) % 2 == 1) {
45              verify(A);
46          } else {
47              verify(B);
48          }
49      }
50      return 0;
51  }
52
53  void init() {
54      #pragma omp for schedule(static) private(i)
55      for (int i = 0; i < N; i++) {
56          A[0][i] = A[N-1][i] = 0.0;
57          A[i][0] = A[i][N-1] = 0.0;
58
59          B[0][i] = B[N-1][i] = 0.0;
60          B[i][0] = B[i][N-1] = 0.0;
61      }
62
63      #pragma omp for schedule(static) private(i, j)
64      for (int i = 1; i <= N-2; i++) {
65          for (int j = 1; j <= N-2; j++) {
66              A[i][j] = 1.0 + i + j;
67              B[i][j] = 1.0 + i + j;
68          }
69      }
70  }
71
72  void relax(double src[N][N], double dest[N][N]) {
73
74      #pragma omp for reduction(max:eps) private(i, j) schedule(static)
75      for (int i = 1; i <= N-2; i++) {
```

```
76          for (int j = 1; j <= N-2; j++) {
77              double new_val = (src[i-1][j] + src[i+1][j] +
78                                src[i][j-1] + src[i][j+1]) / 4.0;
79
80              double diff = fabs(src[i][j] - new_val);
81              eps = Max(eps, diff);
82
83              dest[i][j] = new_val;
84          }
85      }
86  }
87
88  void verify(double src[N][N]) {
89      double s = 0.0;
90
91      #pragma omp parallel for reduction(+:s) private(i, j) schedule(static)
92      for (int i = 0; i < N; i++) {
93          for (int j = 0; j < N; j++) {
94              s += src[i][j] * (i + 1) * (j + 1);
95          }
96      }
97      #pragma omp master
98      {
99          s /= (N * N);
100         printf("␣␣S␣=␣%f\n", s);
101     }
102
103 }
```