Московский государственный университет имени М.В. Ломоносова

Казахстанский филиал

Факультет вычислительной математики и кибернетики

# Отчет о выполнении задания

## Суперкомпьютеры и параллельная обработка данных

**Студент:** Продан Анатолий

**Преподаватель:** Бахтин В.А.

Астана, 2024

# Содержание

# 1 Введение.

Данная работа посвящена изучению и реализации методов распараллеливания вычислений на примере численной задачи. Основной целью является повышение вычислительной эффективности программы. Было выполнено два подхода для выполнения цели:

- 1) Использование SIMD-инструкций(AVX - Advanced Vector Extensions) Данный подход предполагает применение векторных операций для одновременной обработки нескольких элементов данных, что позволяет существенно ускорить вычисления за счет эффективного использования аппаратных возможностей процессора, а за счет использования OpenMP, программа должна существенно ускориться. Так же была идея проверить работу для SSE, но AVX является продвинутой версией SSE. Тут я ограничусь командами из описания INTEL, т.е. работа напрямую через ассемблер дало бы, ну... много проблем для меня.
- 2) Классическое решение с использованием OpenMP. Применяем распараллеливание вычислений с помощью директив OpenMP, что обеспечивает равномерное распределение вычислительной нагрузки между потоками и упрощает реализацию параллельных алгоритмов.

**SIMD** (**Single Instruction, Multiple Data**) – это одна из парадигм параллельных вычислений, в которой одна команда процессора одновременно выполняется над несколькими наборами данных. Эта концепция особенно эффективна в задачах, где нужно обработать большие массивы данных с одинаковыми операциями, например, в графике, обработке сигналов и численных вычислениях.

Исследование и реализация проводились на суперкомпьютере K10 и сервере(т.к. был потерян доступ), тут уже пояснение, что 1)(Глава 4) тут решается на K10, 2) решалось на сервере, после потери доступа к суперкомпьютеру. Оригинальная программа включает обход двумерных массивов, вычисление новых значений (функция `relax`), подсчет погрешностей (функция `resid`) и верификацию результата (функция `verify`).

## 2 Основная программа и ее оптимизация.

Рассмотрим искомую программу:

—

Основной цикл программы

```
int main(int an, char **as)
{
    int it;
    init();
    for (it = 1; it <= itmax; it++)
    {
        eps = 0.;
        relax();
        resid();
        printf("it=%4i   eps=%f\n", it, eps);
        if (eps < maxeps) break;
    }
    verify();
    return 0;
}
```

—

Задает начальные значения массива $A$.

```
void init()
{
    for(j = 0; j <= N - 1; j++)
    for(i = 0; i <= N - 1; i++)
    {
        if(i == 0 || i == N - 1 || j == 0 || j == N - 1)
            A[i][j] = 0.;
        else
            A[i][j] = (1. + i + j);
    }
}
```

—

Выполняет обновление массива $B$ путем усреднения соседних элементов массива $A$.

```
void relax()
{
    for(j=2; j<=N-3; j++)
    for(i=2; i<=N-3; i++)
    {
        B[i][j] = (A[i-2][j] + A[i-1][j] +
                   A[i+2][j] + A[i+1][j] +
                   A[i][j-2] + A[i][j-1] +
                   A[i][j+2] + A[i][j+1]) / 8.;
    }
}
```

—

Проверяет разницу между текущим состоянием массива $A$ и новым состоянием массива $B$. Максимальная разница используется как критерий сходимости.

```
void resid()
{
    for(j=1; j<=N-2; j++)
    for(i=1; i<=N-2; i++)
    {
        double e;
        e = fabs(A[i][j] - B[i][j]);
        A[i][j] = B[i][j];
        eps = Max(eps, e);
    }
}
```

—

Рассчитывает контрольную сумму $S$ для проверки корректности вычислений и вывода результатов.

```
void verify()
{
    double s;
```

```
    s = 0.;
    for(j=0; j<=N-1; j++)
    for(i=0; i<=N-1; i++)
    {
        s = s + A[i][j] * (i + 1) * (j + 1) / (N * N);
    }
    printf("  S = %f\n", s);
}
```

## 2.1 Ключевые этапы программы.

Программа состоит из трех основных этапов:

1. **Инициализация массива $A$:**

   - Задается двумерная сетка $A$ размером $N \times N$.
   - Граничные значения сетки (где $i = 0$, $i = N - 1$, $j = 0$, $j = N - 1$) заполняются нулями:

   $$A[i][j] = 0 \quad \text{для граничных ячеек.}$$

   - Внутренние значения инициализируются формулой:

   $$A[i][j] = 1 + i + j.$$

2. **Основной цикл итераций:** Выполняется до достижения заданного количества итераций $it_{\max}$ или пока не будет выполнен критерий сходимости (ошибка $\varepsilon < \text{maxeps}$).

   - На каждом шаге:

     (a) Функция `relax` обновляет массив $B$ путем усреднения соседей из массива $A$:

     $$B[i][j] = \frac{1}{8}(A[i-2][j] + A[i-1][j] + A[i+1][j] + A[i+2][j] + A[i][j-2] + A[i][j-1] + A[i][j+1] + A[i][j+2]).$$

     (b) Функция `resid` вычисляет максимальное различие между массивами $A$ и $B$:

     $$\varepsilon = \max_{i,j} |A[i][j] - B[i][j]|.$$

     Если $\varepsilon < \text{maxeps}$, итерационный процесс завершается.

3. **Верификация (`verify`):** После завершения итераций программа проверяет итоговое состояние массива $A$, вычисляя контрольную сумму:

   $$S = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} \frac{A[i][j] \cdot (i+1) \cdot (j+1)}{N^2}.$$

## 2.2 Вычислительная сложность.

Основные вычисления программы заключаются в операциях усреднения (`relax`) и проверки (`resid`). Оба этапа имеют сложность $\mathcal{O}(N^2)$ для каждой итерации. Общая сложность программы зависит от числа итераций $k$, то есть:

$$\mathcal{O}_{\text{total}} = k \cdot \mathcal{O}(N^2).$$

## 2.3 Ключевые проблемы в программе.

Программа содержит несколько моментов, которые снижают ее производительность и читаемость. Эти проблемы можно разделить на несколько категорий:

### 2.3.1 Несогласованность использования индексов `i` и `j`

В программе индексы `i` и `j` задаются разными способами:

- В некоторых местах индексация идет от 0 до $N - 1$.
- В других местах используется условие $j \leq N - 3$, что не всегда очевидно, это действительно **тонко**.

Такая неоднородность затрудняет чтение кода, вызывает путаницу и может приводить к ошибкам. Например, в функциях `relax` и `resid` использование `i` и `j` отличается.

### 2.3.2 Неэффективная работа с памятью

Каждый элемент массива `A` или `B` обрабатывается по отдельности:

$$B[i][j] = \frac{1}{8}(A[i-2][j] + A[i-1][j] + A[i+1][j] + A[i+2][j] + \\ A[i][j-2] + A[i][j-1] + A[i][j+1] + A[i][j+2]).$$

Это приводит к следующим проблемам:

- Многократное обращение к одним и тем же элементам массива.
- Увеличение количества операций чтения/записи, что снижает производительность при больших размерах массива $N$.

## 2.4 Оптимизация используя AVX.

### 2.4.1 Что такое AVX?

**AVX (Advanced Vector Extensions)** — это набор SIMD (Single Instruction, Multiple Data) инструкций, разработанных для современных процессоров. Эти инструкции позволяют выполнять одну и ту же операцию одновременно над несколькими числами, что значительно ускоряет вычисления.

Используется 256-битный регистр, который может хранить:

- 4 числа с плавающей точкой двойной точности (`double`),
- 8 чисел с плавающей точкой одинарной точности (`float`).

## 2.5 Основные функции AVX.

Для работы с AVX используются функции из библиотеки <**immintrin.h**>. Вот основные из них:

| **Функция** | **Описание** |
|---|---|
| `_mm256_loadu_pd(ptr)` | Загружает 4 числа `double` из памяти в регистр. |
| `_mm256_storeu_pd(ptr, reg)` | Сохраняет содержимое регистра в память (4 числа). |
| `_mm256_add_pd(a, b)` | Поэлементное сложение двух регистров. |
| `_mm256_sub_pd(a, b)` | Поэлементное вычитание двух регистров. |
| `_mm256_mul_pd(a, b)` | Поэлементное умножение двух регистров. |
| `_mm256_div_pd(a, b)` | Поэлементное деление двух регистров. |
| `_mm256_max_pd(a, b)` | Поэлементное вычисление максимума двух регистров. |
| `_mm256_set1_pd(value)` | Заполняет все элементы регистра одним значением. |
| `_mm256_andnot_pd(mask, a)` | Поэлементная операция **AND NOT** для обработки битов. |

## 2.6 Пример работы функций.

Рассмотрим пример поэлементного сложения двух массивов с использованием AVX:

$$\mathrm{a} = \{1.0, 2.0, 3.0, 4.0\}, \quad \mathrm{b} = \{5.0, 6.0, 7.0, 8.0\}.$$

```
// Исходные массивы
double a[4] = {1.0, 2.0, 3.0, 4.0};
```

```
double b[4] = {5.0, 6.0, 7.0, 8.0};
double result[4];
// Загружаем массивы в регистры
__m256d vec_a = _mm256_loadu_pd(a);
__m256d vec_b = _mm256_loadu_pd(b);
// Складываем элементы
__m256d vec_result = _mm256_add_pd(vec_a, vec_b);
// Сохраняем результат в массив
_mm256_storeu_pd(result, vec_result);
// Результат: {6.0, 8.0, 10.0, 12.0}
```

# 3 Описание программы с использованием AVX.

## 3.1 Функция `relax`.

Функция `relax` обновляет массив $B$, вычисляя среднее значение соседних элементов массива $A$. Это достигается с использованием инструкций AVX для обработки сразу 4 элементов за одну операцию.

```
void relax()
{
    for (int i = 2; i < N - 2; i++)
    {
        for (int j = 2; j < N - 2; j += 4)
        {
            __m256d ai_2j = _mm256_loadu_pd(&A[i - 2][j]);
            __m256d ai_1j = _mm256_loadu_pd(&A[i - 1][j]);
            __m256d ai_2jp2 = _mm256_loadu_pd(&A[i + 2][j]);
            __m256d ai_1jp1 = _mm256_loadu_pd(&A[i + 1][j]);

            __m256d aijm2 = _mm256_loadu_pd(&A[i][j - 2]);
            __m256d aijm1 = _mm256_loadu_pd(&A[i][j - 1]);
            __m256d aijp2 = _mm256_loadu_pd(&A[i][j + 2]);
            __m256d aijp1 = _mm256_loadu_pd(&A[i][j + 1]);
```

```
            __m256d sum = _mm256_add_pd(ai_2j, ai_1j);
            sum = _mm256_add_pd(sum, ai_2jp2);
            sum = _mm256_add_pd(sum, ai_1jp1);
            sum = _mm256_add_pd(sum, aijm2);
            sum = _mm256_add_pd(sum, aijm1);
            sum = _mm256_add_pd(sum, aijp2);
            sum = _mm256_add_pd(sum, aijp1);

            __m256d avg = _mm256_div_pd(sum,
            _mm256_set1_pd(8.0));
            _mm256_storeu_pd(&B[i][j], avg);
        }
    }
}
```

### 3.2 Функция `resid`.

Функция `resid` вычисляет разницу между массивами $A$ и $B$, обновляет значения массива $A$ и находит максимальную ошибку.

```
void resid()
{
    eps = 0.0;
    __m256d max_eps = _mm256_set1_pd(0.0);

    for (int i = 1; i < N - 1; i++)
    {
        for (int j = 1; j < N - 1; j += 4)
        {
            __m256d aij = _mm256_loadu_pd(&A[i][j]);
            __m256d bij = _mm256_loadu_pd(&B[i][j]);

            __m256d diff = _mm256_sub_pd(aij, bij);
            __m256d abs_diff =
            _mm256_andnot_pd(_mm256_set1_pd(-0.0), diff);
```

```
            max_eps = _mm256_max_pd(max_eps, abs_diff);
            _mm256_storeu_pd(&A[i][j], bij);
        }
    }

    double temp_eps[4];
    _mm256_storeu_pd(temp_eps, max_eps);
    for (int idx = 0; idx < 4; idx++)
    {
        eps = Max(eps, temp_eps[idx]);
    }
}
```

### 3.3 Функция `verify`.

Функция `verify` проверяет корректность работы программы, вычисляя контрольную сумму $S$.

```
void verify()
{
    double s = 0.0;
    double norm_factor = 1.0 / (N * N);

    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j += 4)
        {
            __m256d aij = _mm256_loadu_pd(&A[i][j]);
            __m256d indices = _mm256_set_pd(
                (i + 1) * (j + 4), (i + 1) * (j + 3),
                (i + 1) * (j + 2), (i + 1) * (j + 1)
            );
            __m256d prod = _mm256_mul_pd(aij, indices);
            __m256d scaled_prod =
```

```
            _mm256_mul_pd(prod, _mm256_set1_pd(norm_factor));

            double temp_s[4];
            _mm256_storeu_pd(temp_s, scaled_prod);
            s += temp_s[0] + temp_s[1] + temp_s[2] + temp_s[3];
        }
    }
    printf("  S = %f\n", s);
}
```

## 3.4 Преимущества и недостатки оптимизации с использованием AVX.

### 3.4.1 Преимущества.

1. **Увеличение производительности:**
   - AVX позволяет обрабатывать сразу несколько (4 или 8) элементов данных за одну операцию.
2. **Эффективное использование ресурсов процессора:**
   - SIMD-инструкции задействуют векторные регистры и арифметико-логические устройства (ALU) процессора.
   - Обеспечивают высокую производительность за счет параллельной обработки данных.
3. **Снижение числа итераций в циклах:**
   - Обработка 4 элементов за одну итерацию вместо одного уменьшает общее количество итераций цикла.
4. **Уменьшение количества обращений к памяти:**
   - Загрузка данных блоками снижает количество операций чтения и записи.
   - Уменьшаются задержки, связанные с доступом к оперативной памяти.

### 3.4.2 Недостатки.

1. **Аппаратная зависимость:**
   - AVX поддерживается только современными процессорами.
   - На старых процессорах код не будет работать.
2. **Ограничения SIMD:**
   - SIMD работает эффективно только с данными, которые можно обрабатывать параллельно.
3. **Сложность разработки и отладки:**
   - Код с SIMD-инструкциями сложнее читать, понимать и отлаживать.
4. **Отсутствие автоматической оптимизации:**
   - Компиляторы не всегда автоматически векторизуют код.

## 3.5 Результаты измерений.

- **Программа без AVX:**
  - Размер матрицы: $4098 \times 4098$.
  - Время выполнения: **92 секунды**.
- **Программа с AVX:**
  - Размер матрицы: $4098 \times 4098$.
  - Время выполнения: **45 секунд**.

## 3.6 Выигрыш в производительности.

1. **Общее ускорение:**

$$\text{Ускорение} = \frac{\text{Время без AVX}}{\text{Время с AVX}} = \frac{92}{45} \approx 2.04.$$

Программа с AVX работает примерно в **2 раза быстрее**.

# 4 OpenMP + AVX.

## 4.1 Программа с использованием `for + collapse`.

Метод распараллеливания с использованием `#pragma omp for collapse(2)` позволяет эффективно распределить итерации двух вложенных циклов между потоками.

### 4.1.1 Основные элементы распараллеливания.

- **В функции `relax`:**

```
#pragma omp for collapse(2)
for (int i = 2; i < N - 2; i++)
{
    for (int j = 2; j < N - 2; j += 4)
    {
        // Обработка 4 элементов с использованием AVX
    }
}
```

  - Директива `collapse(2)` объединяет два цикла (`i` и `j`) в один общий, чтобы улучшить распределение нагрузки.
  - Каждый поток обрабатывает свою часть итераций.

- **В функции `resid`:**

```
#pragma omp for collapse(2)
for (int i = 1; i < N - 1; i++)
{
    for (int j = 1; j < N - 1; j += 4)
    {
        // Вычисление ошибки eps
    }
}
```

Добавлена синхронизация с помощью `#pragma omp barrier`, чтобы обеспечить согласованность обновлений `eps`.

- **В функции** `verify`**:** Используется `reduction(+:s)`, чтобы суммировать значения от каждого потока:

```
#pragma omp parallel for reduction(+:s)
for (int i = 0; i < N; i++)
{
    for (int j = 0; j < N; j += 4)
    {
        // Суммирование значений
    }
}
```

## 4.2 Программа с использованием `task + блочное разбиение.`

Метод распараллеливания с использованием `#pragma omp task` разделяет массив на блоки (`BLOCK_SIZE x BLOCK_SIZE`), которые обрабатываются в виде задач.

### 4.2.1 Основные элементы распараллеливания.

- **В функции** `relax`**:**

```
for (int i = 2; i < N - 2; i += BLOCK_SIZE)
{
    for (int j = 2; j < N - 2; j += BLOCK_SIZE)
    {
        #pragma omp task firstprivate(i, j, i_end, j_end)
        {
            for (int ii = i; ii < i_end; ii++)
            {
                for (int jj = j; jj < j_end; jj += 4)
                {
                    // Обработка блока 4x4
                }
```

```
            }
        }
    }
}
```

Каждый блок обрабатывается как независимая задача.

- **В функции `resid`:** Аналогично функции `relax`, задачи создаются для обработки блоков, а вычисление локальной ошибки объединяется в глобальную:

```
for (int i = 1; i < N - 1; i += BLOCK_SIZE)
{
    for (int j = 1; j < N - 1; j += BLOCK_SIZE)
    {
        #pragma omp task firstprivate(i, j, i_end, j_end)
        {
            // Вычисление ошибки eps
        }
    }
}
```

## 4.3 Сравнение методов.

| **Критерий** | **for + collapse** | **task + блочное разбиение** |
|---|---|---|
| Простота реализации | Легче в реализации, достаточно использовать директиву `for`. | Требует дополнительного кода для управления задачами. |
| Балансировка нагрузки | Хорошо работает при равномерной нагрузке. | Эффективно распределяет нагрузку при больших массивах и нерегулярных данных. |
| *Продолжение на следующей странице* | | |

| **Критерий** | **for + collapse** | **task + блочное разбиение** |
|---|---|---|
| Накладные расходы | Минимальные накладные расходы на распределение итераций. | Более высокие накладные расходы из-за создания задач. |
| Производительность | Высокая при равномерной структуре данных. | Лучше для задач с большими массивами или разреженными данными. |
| Сложность отладки | Проще, так как работа идет с циклами. | Сложнее из-за задач и блочного разбиения. |

## 4.4 Заключение.

- Метод `for + collapse` удобен для задач с равномерной нагрузкой и требует минимальных изменений в коде.
- Метод `task + блочное разбиение` эффективнее для больших массивов и сложных задач с динамическим распределением нагрузки, но требует большего контроля и сложнее в реализации.

# 5 Исследование производительности: AVX + `for` + `collapse`.

## 5.1 Результаты тестирования.

Программа тестировалась на матрице размером $4098 \times 4098$ с различным количеством потоков и уровнями оптимизации компиляторов `gcc` и `icc`. Рассматривались следующие варианты:

- Без оптимизации (`no optim`).
- Оптимизация `O2`.
- Оптимизация `O3`.

## 5.2 Сравнение компиляторов (без оптимизации).

| **Потоки** | **gcc (no optim)** | | **icc (no optim)** | |
|---|---|---|---|---|
| | **S** | **Time (s)** | **S** | **Time (s)** |
| 1 | 22680568506.895 | 45.27 | 22680568506.895 | 8.89 |
| 2 | 22680568506.893 | 22.70 | 22680568506.893 | 4.64 |
| 4 | 22680568506.895 | 11.49 | 22680568494.693 | 2.71 |
| 8 | 22680568495.935 | 6.03 | 22680568492.212 | 2.07 |
| 16 | 22680568506.894 | 5.93 | 22680568506.894 | 2.34 |

## 5.3 Сравнение оптимизации O2 (gcc и icc).

| **Потоки** | **gcc (O2)** | | **icc (O2)** | |
|---|---|---|---|---|
| | **S** | **Time (s)** | **S** | **Time (s)** |
| 1 | 22680568506.895 | 9.60 | 22680568506.895 | 8.96 |
| 2 | 22680568506.893 | 4.92 | 22680568506.892 | 4.94 |
| 4 | 22680568506.895 | 2.82 | 22680568506.893 | 2.72 |
| 8 | 22680568506.895 | 2.05 | 22680568495.935 | 2.01 |
| 16 | 22680568506.894 | 2.05 | 22680568506.894 | 2.34 |

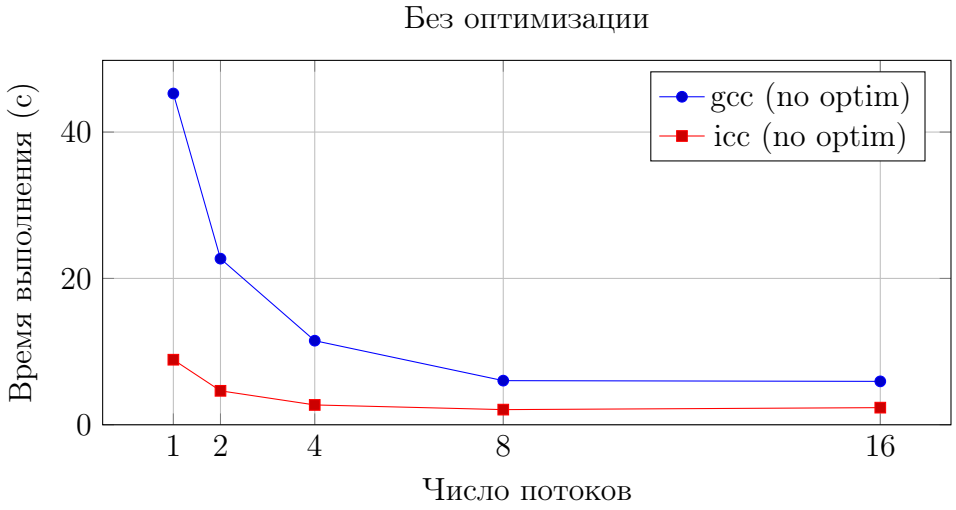
## 5.4 Сравнение оптимизации O3 (gcc и icc).

| Потоки | gcc (O3) | | icc (O3) | |
|---|---|---|---|---|
| | S | Time (s) | S | Time (s) |
| 1 | 22680568506.895 | 9.37 | 22680568506.895 | 9.55 |
| 2 | 22680568506.893 | 4.89 | 22680568506.893 | 4.77 |
| 4 | 22680568506.895 | 2.81 | 22680568506.895 | 2.74 |
| 8 | 22680568506.894 | 2.02 | 22680568506.894 | 2.24 |
| 16 | 22680568506.894 | 2.21 | 22680568506.894 | 2.48 |

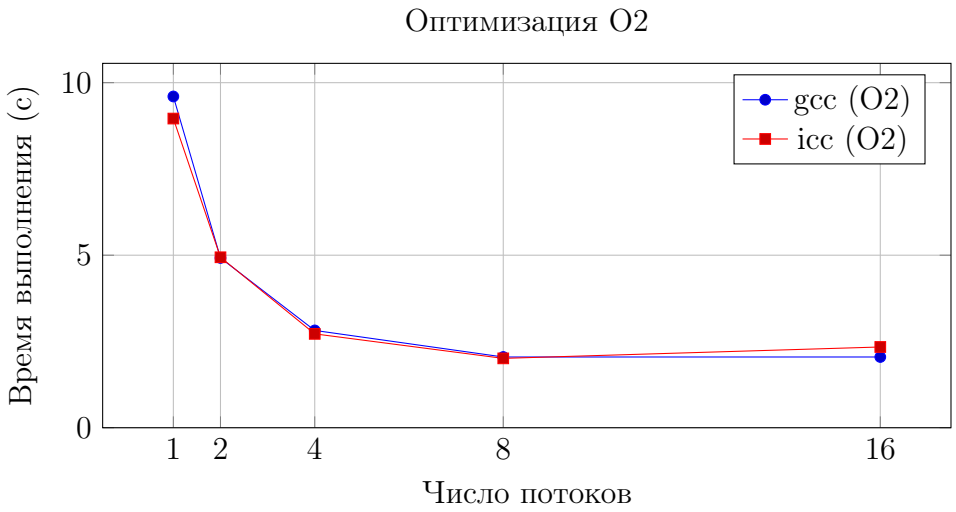
## 5.5 Выводы.

- Эффективность многопоточности:
  - Увеличение числа потоков значительно снижает время выполнения, особенно с компилятором `icc`.
  - При 16 потоках наблюдается насыщение производительности, так как дополнительные потоки слабо влияют на время выполнения.
- Роль оптимизации:
  - Без оптимизации (`no optim`) время выполнения существенно выше.
  - Оптимизация `O2` значительно улучшает производительность, особенно на `icc`.
  - Оптимизация `O3` дает наилучшие результаты благодаря агрессивной векторизации и оптимизации циклов.
- Сравнение компиляторов:
  - `gcc`: Хорошо справляется с оптимизацией на уровнях `O2` и `O3`.
  - `icc`: Лучше оптимизирует код для многопоточных вычислений и векторизации, что приводит к лучшей производительности.

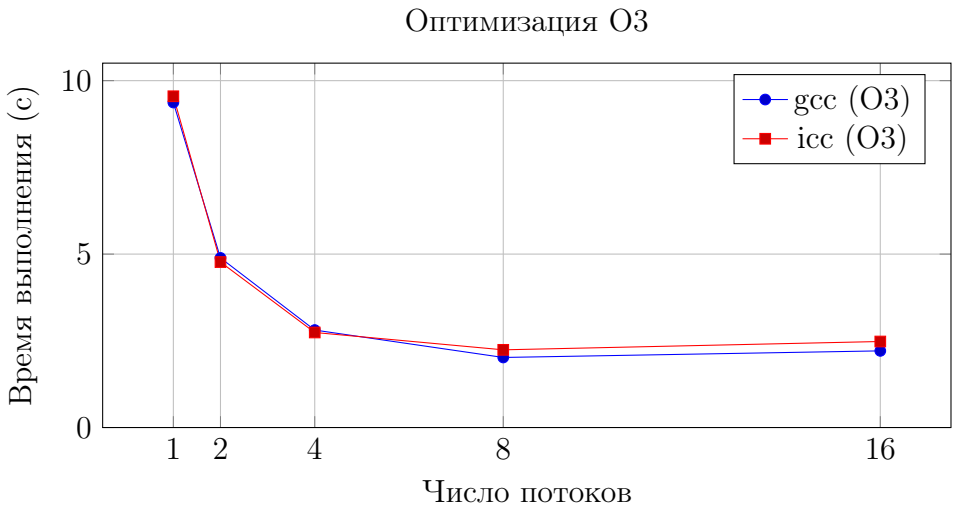
## 5.6 График времени выполнения без оптимизации (gcc vs icc).



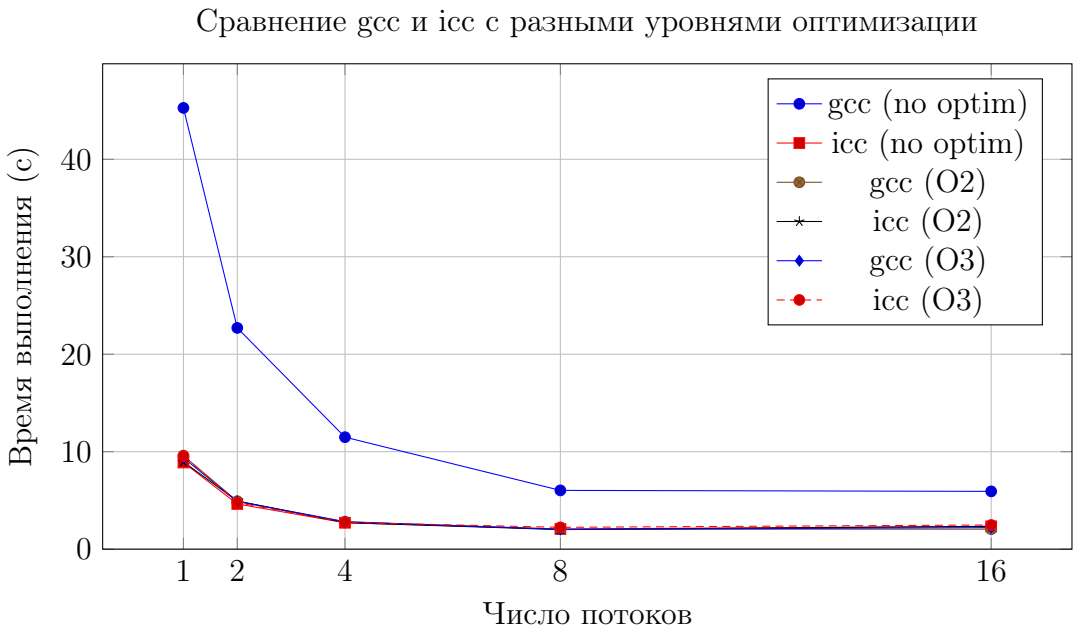

## 5.7 График времени выполнения с оптимизацией O2 (gcc vs icc).



## 5.8 График времени выполнения с оптимизацией O3 (gcc vs icc).




## 5.9 Общий график времени выполнения для всех конфигураций.



# 6 Исследование производительности: `task`.

## 6.1 Результаты тестирования.

Программа тестировалась на матрице размером $4098 \times 4098$ с различным количеством потоков и уровнями оптимизации компиляторов `gcc` и `icc`. Рассматривались следующие варианты:

- Без оптимизации (`no optim`).
- Оптимизация `O2`.
- Оптимизация `O3`.

## 6.2 Результаты без оптимизации (`no optim`).

| **Потоки** | **gcc (no optim)** | | **icc (no optim)** | |
|---|---|---|---|---|
| | **S** | **Time (s)** | **S** | **Time (s)** |
| 1 | 22680463238.511 | 76.42 | 22680568506.895 | 18.69 |
| 2 | 22680568506.893 | 45.89 | 22680568506.892 | 12.57 |
| 4 | 22680568506.895 | 23.52 | 22680568506.893 | 6.74 |
| 8 | 22680568506.894 | 12.47 | 22680568506.894 | 4.21 |
| 16 | 22680568506.894 | 8.84 | 22680568506.894 | 8.10 |

## 6.3 Результаты с оптимизацией O2.

| **Потоки** | **gcc (no optim)** | | **icc (no optim)** | |
|---|---|---|---|---|
| | **S** | **Time (s)** | **S** | **Time (s)** |
| 1 | 22680463238.511 | 13.43 | 22680568506.895 | 18.79 |
| 2 | 22680568506.893 | 9.79 | 22680568506.892 | 12.55 |
| 4 | 22680568506.895 | 6.44 | 22680568506.893 | 6.53 |
| 8 | 22680568506.894 | 11.17 | 22680568506.894 | 3.93 |
| 16 | 22680568506.894 | 15.58 | 22680568506.894 | 8.68 |

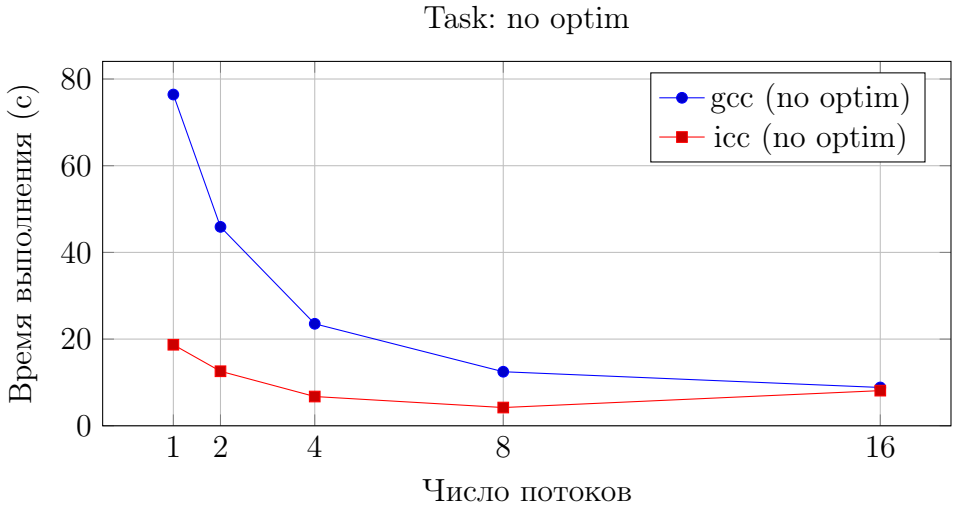
## 6.4 Результаты с оптимизацией O3.

| Потоки | gcc (no optim) | | icc (no optim) | |
|---|---|---|---|---|
| | S | Time (s) | S | Time (s) |
| 1 | 22680463238.511 | 13.35 | 22680568506.895 | 14.52 |
| 2 | 22680568506.893 | 9.76 | 22680568506.893 | 11.57 |
| 4 | 22680568506.895 | 6.41 | 22680568506.895 | 6.64 |
| 8 | 22680568506.894 | 12.04 | 22680568506.894 | 4.24 |
| 16 | 22680568506.894 | 15.67 | 22680568506.894 | 7.95 |

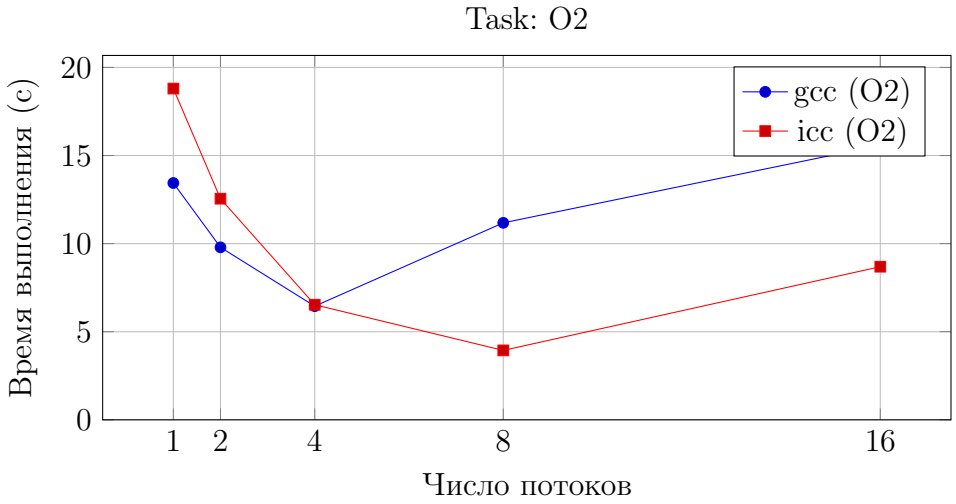
## 6.5 Выводы.

- Общие результаты:
  - При увеличении потоков время выполнения уменьшается до 8 потоков, после чего наблюдается насыщение.
  - Компилятор `icc` значительно опережает `gcc` на всех уровнях оптимизации.
- Оптимизация `O2`:
  - Время выполнения уменьшается, но у `gcc` при 16 потоках наблюдается рост времени выполнения из-за накладных расходов.
  - `icc` демонстрирует стабильные результаты, достигая минимального времени при 8 потоках.
- Оптимизация `O3`:
  - У `gcc` аналогичные проблемы с ростом времени при большом числе потоков.
  - `icc` демонстрирует стабильное поведение благодаря агрессивной оптимизации (AVX/SIMD).

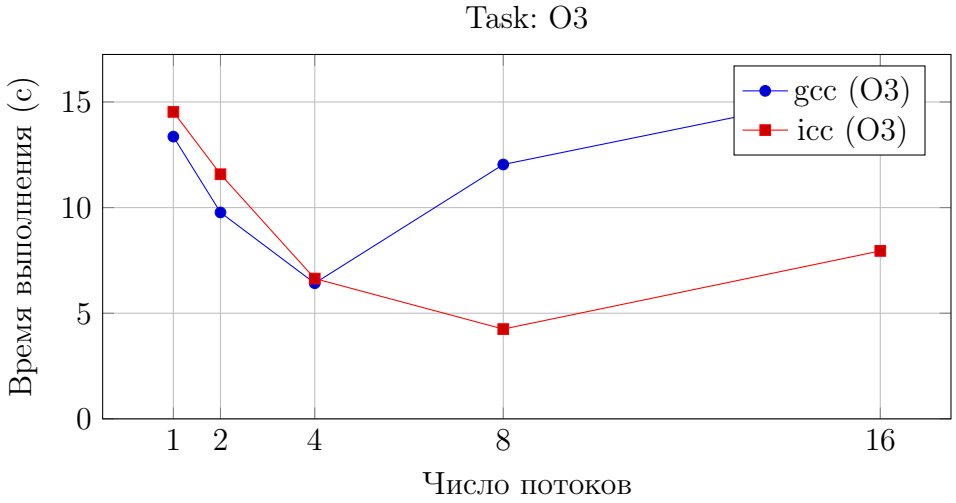
## 6.6 График для no optim (task).




## 6.7 График для O2 (task).

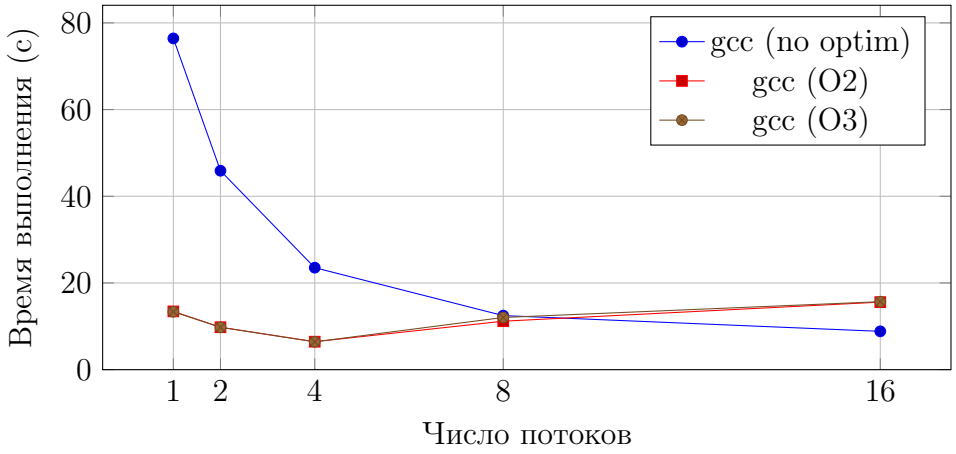



## 6.8 График для O3 (task).



### 6.9 Общий график для task.

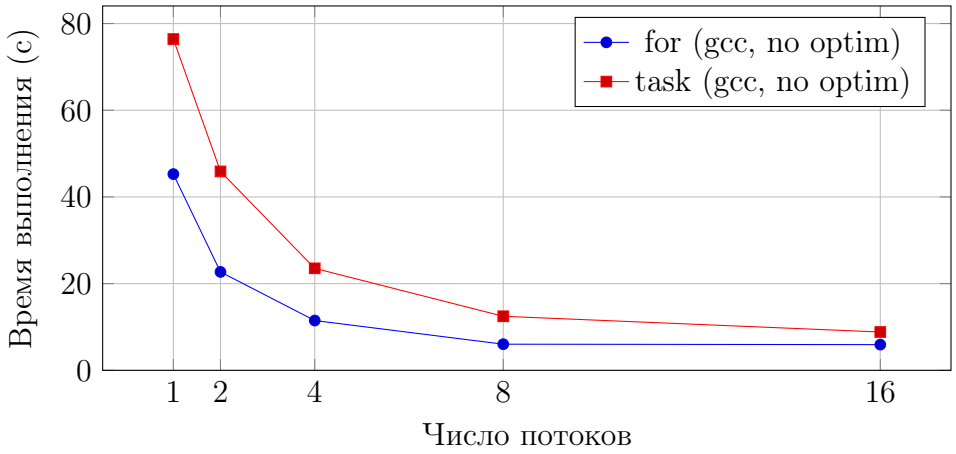
Task: общий график для всех оптимизаций



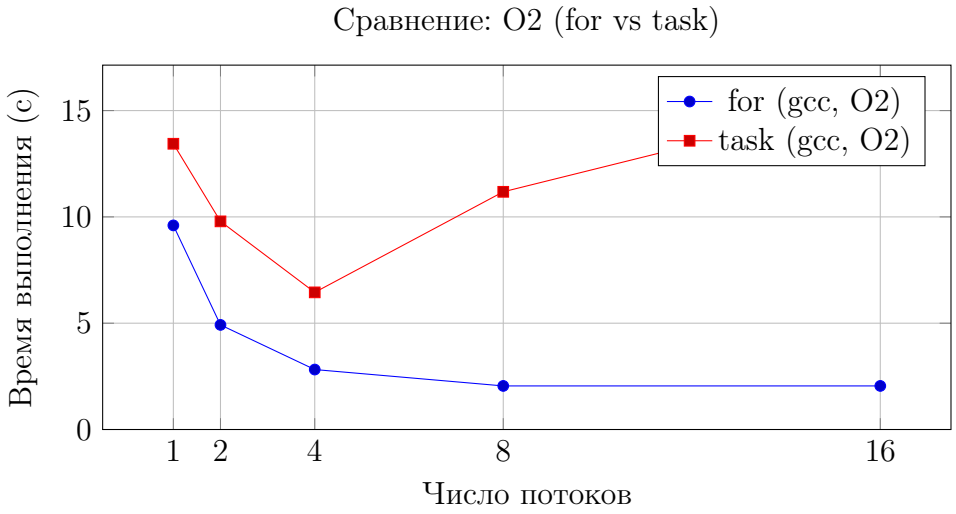

## 7 Сравнение for и task.

### 7.1 Сравнение no optim (for vs task).

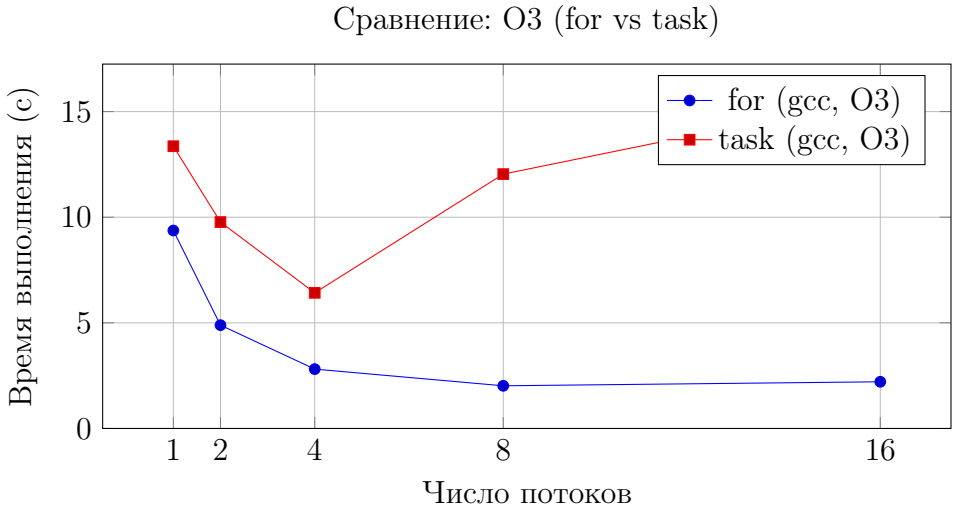
Сравнение: no optim (for vs task)



## 7.2 Сравнение O2 (for vs task).




## 7.3 Сравнение O3 (for vs task).



# 8 Классическое решение проблемы через `#pragma omp for`.

## 8.1 Основная идея OpenMP-параллелизации

- При использовании `#pragma omp parallel` среда OpenMP создает пул потоков.
- Директива `#pragma omp for` (или `#pragma omp parallel for`, когда совмещаются сразу оба действия) делит итерации цикла между всеми активными потоками.
- Каждый поток получает свою часть итераций и выполняет их параллельно с другими.
- По завершении параллельного участка или по достижении `#pragma omp barrier` потоки синхронизируются.

Таким образом, если у нас есть вложенные циклы по `i` и `j`, то OpenMP будет параллельно выполнять блоки итераций.

## 8.2 Запуск параллельной области.

```
#pragma omp parallel
{
    int chunk = get_chunk_size();

    for(int it = 1; it <= itmax; it++)
    {
        ...
    }
}
```

- `#pragma omp parallel` Создает несколько потоков и заходит в эту область всеми потоками.
- Внутри параллельной области, каждый поток выполнит тело блока.
- Переменная `chunk` (локальная для каждого потока) задает размер куска (*chunk size*) для статического расписания (`schedule(static, chunk)`).

## 8.3 Параллельный проход по двумерному массиву (обновление B)

```
#pragma omp for schedule(static, chunk) nowait
for(int i = 2; i < N - 2; i++)
{
    for(int j = 2; j < N - 2; j++)
    {
        B[i][j] = ( A[i-2][j] + A[i-1][j] + ... ) / 8.0;
    }
}
```

1. `#pragma omp for` сообщает OpenMP, что следующий цикл `for` можно распределить между потоками.
2. `schedule(static, chunk)` — это политика распределения итераций:
   - `static` означает, что диапазон `i` статически разбивается на куски (размером `chunk`) и раздается потокам так, чтобы каждый поток знал заранее, какие итерации ему выполнять.
   - `chunk` берется функцией `get_chunk_size()`, в которой, например, есть привязка к числу потоков.
3. `nowait` говорит, что после выполнения этого цикла не нужна обязательная барьерная синхронизация; но поскольку дальше в коде стоит `#pragma omp barrier`, потоки встретятся там.

В итоге каждый поток обрабатывает свою часть строк `i`, заполняя матрицу `B`.

## 8.4 Параллельный проход для обновления A и вычисления eps

```
#pragma omp barrier
#pragma omp for schedule(static, chunk) reduction(max:eps)
for(int i = 1; i < N - 1; i++)
{
    for(int j = 1; j < N - 1; j++)
```

```
    {
        double e = fabs(A[i][j] - B[i][j]);
        A[i][j] = B[i][j];
        eps = Max(eps, e);
    }
}
```

1. `#pragma omp barrier` — гарантирует, что все потоки завершили предыдущий цикл (обновление `B`), прежде чем кто-то начнет следующий этап. Это нужно, чтобы данные в `B` были актуальными и целостными.

2. Новый `#pragma omp for` снова делит итерации между потоками.

3. `reduction(max:eps)` позволяет собирать из каждого потока свое локальное значение `eps` через операцию `max`, а в конце цикла результат записывается в общий `eps`. Иначе говоря, каждый поток вычисляет свой максимум, а OpenMP аккуратно соединяет все локальные максимумы в глобальный.

## 8.5 #pragma omp single

```
#pragma omp single
{
    if (eps < maxeps)
    {
        itmax = it - 1;
    }
}
```

- `#pragma omp single` означает, что этот участок кода выполнит ровно один поток.

Таким образом, в каждой итерации цикла `it`:

1. Все потоки параллельно выполняют вычисление `B`.

2. Все синхронизируются (`barrier`).

3. Все параллельно вычисляют и обновляют `A`, считая `eps` (через `reduction(max:eps)`).

4. Ровно один поток проверяет условие `eps < maxeps`.

## 8.6 Параллелизация в `init_data` и `verify`

Схожим образом выполняются инициализация массива `A` и финальная проверка:

- `#pragma omp parallel for` делит циклы по `i` (и внутри — по `j`) на все потоки.
- В `verify` при суммировании `s` используется `reduction(+: s)`, чтобы суммировать локальные значения от каждого потока к общему результату.

## 8.7 Почему это работает быстро.

1. Вместо последовательного прохода по всему массиву (размером `N` × `N`) одним потоком, несколько потоков делят себе строчки (или блоки строчек) и обрабатывают их одновременно.
2. При достаточно большой задаче, если `N` велико и количество доступных ядер значительно, можно достичь почти линейного ускорения за счет того, что каждый поток параллельно обрабатывает свой участок.
3. Тонкая настройка `schedule(static, chunk)` помогает избежать нагрузочного дисбаланса (например, когда часть итераций достается только одному потоку).

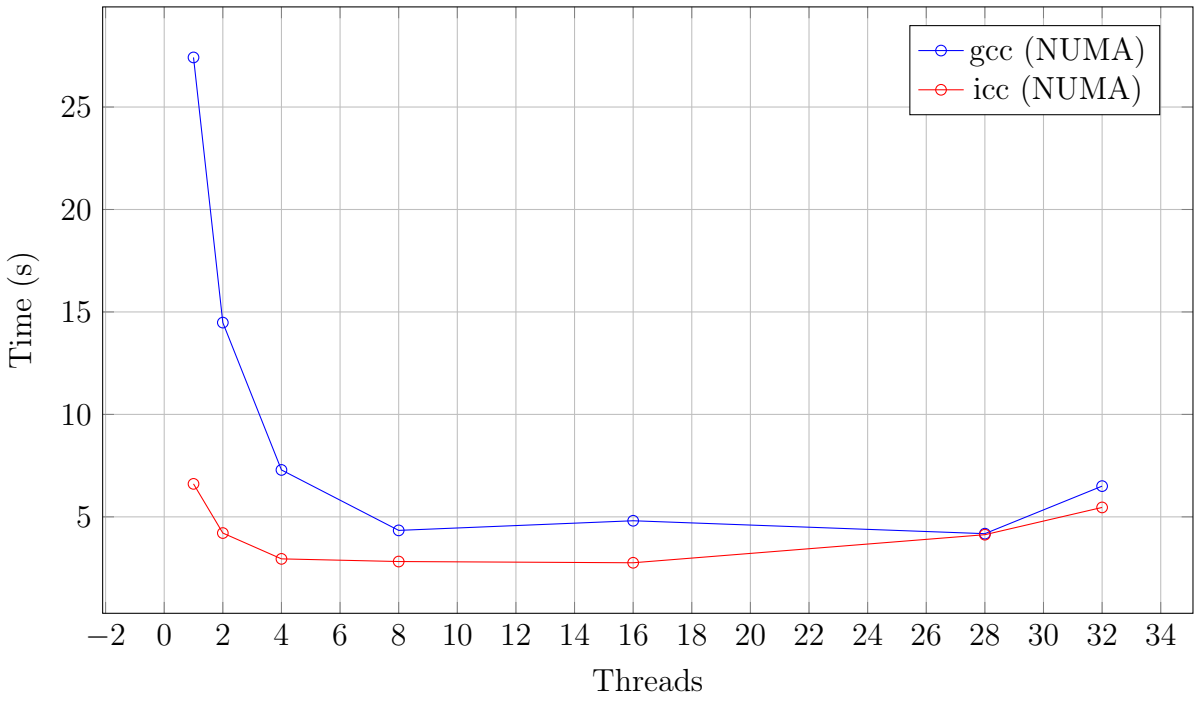
## 8.8 Результаты выполнения.

| **Потоки** | **gcc(NUMA)** | | **icc(NUMA)** | |
|---|---|---|---|---|
| | **S** | **Time (s)** | **S** | **Time (s)** |
| 1 | 22667151283.233 | 27.42 | 22667151283.232 | 6.61 |
| 2 | 22667151283.231 | 14.48 | 22667151283.233 | 4.21 |
| 4 | 22667151283.233 | 7.29 | 22667151283.233 | 2.95 |
| 8 | 22667151283.233 | 4.34 | 22667151283.233 | 2.82 |
| 16 | 22667151283.233 | 4.81 | 22667151283.233 | 2.76 |
| 28 | 22667151283.233 | 4.18 | 22667151283.233 | 4.13 |
| 32 | 22667151283.233 | 6.50 | 22667151283.233 | 5.46 |

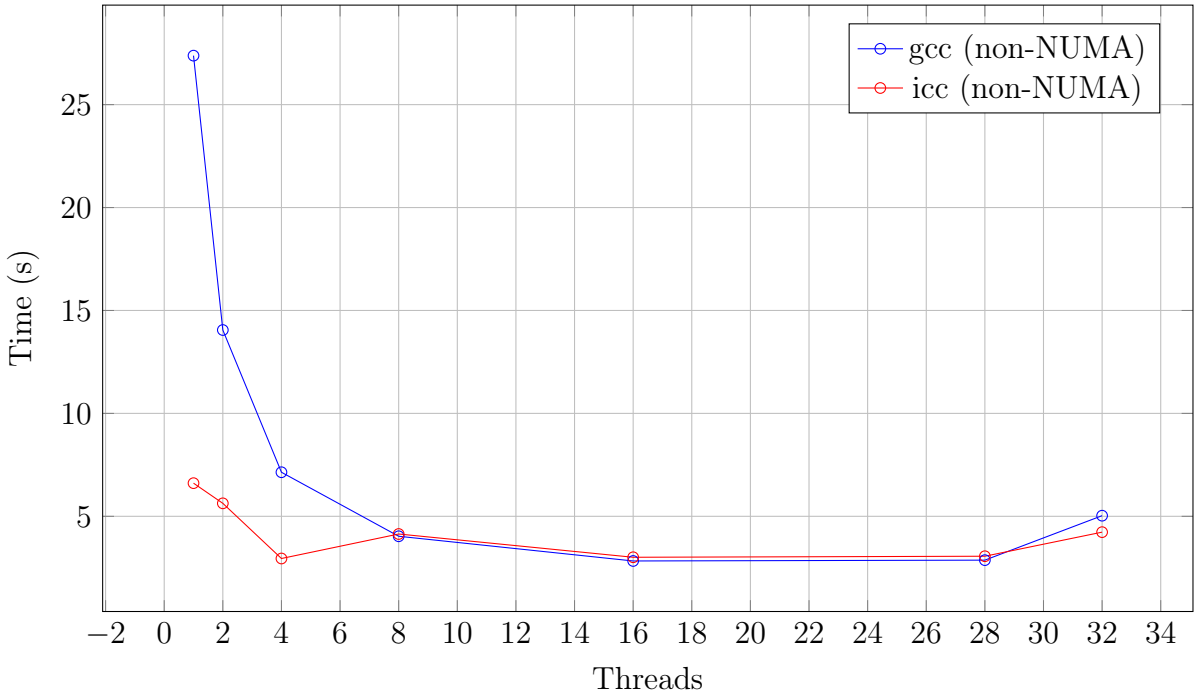
| Потоки | gcc(non-NUMA) | | icc(non-NUMA) | |
|---|---|---|---|---|
| | S | Time (s) | S | Time (s) |
| 1 | 22667151283.233 | 27.38 | 22667151283.232 | 6.61 |
| 2 | 22667151283.231 | 14.05 | 22667151283.233 | 5.63 |
| 4 | 22667151283.233 | 7.14 | 22667151283.233 | 2.95 |
| 8 | 22667151283.233 | 4.03 | 22667151283.233 | 4.14 |
| 16 | 22667151283.233 | 2.83 | 22667151283.233 | 3.01 |
| 28 | 22667151283.233 | 2.87 | 22667151283.233 | 3.06 |
| 32 | 22667151283.233 | 5.03 | 22667151283.233 | 4.23 |

## 8.9 Графики сравнения.

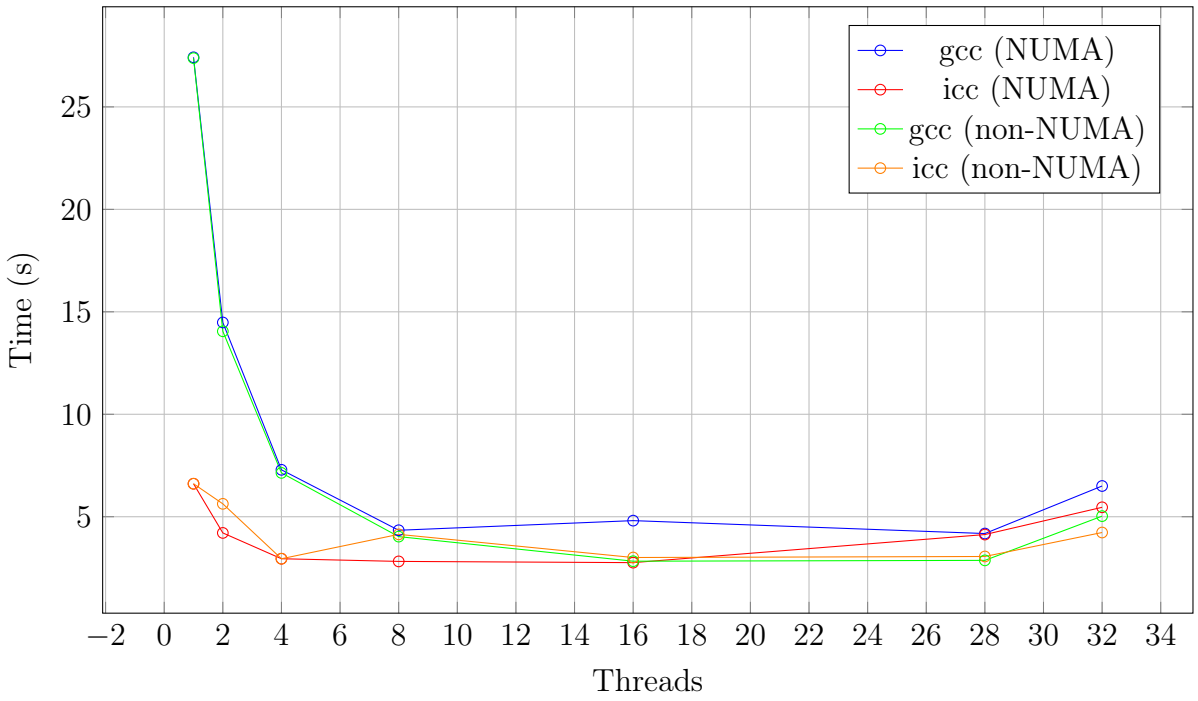
### 8.9.1 Производительность с NUMA.



### 8.9.2 Производительность без NUMA.




### 8.9.3 Сравнение NUMA и non-NUMA.



# 9 Классическое решение проблемы через `#pragma omp task`.

## 9.1 Общая идея параллелизации с `task` + блочное разбиение.

Вместо привычного:

```
#pragma omp parallel for
for (int i = 0; i < N; i++)
{
    ...
}
```

здесь делается следующее:

1. Исходный массив ($N \times N$) разбивается на блоки фиксированного размера (например, $128 \times 128$), определяемого `BLOCK_SIZE`.
2. На каждый блок создается отдельная **задача** (`#pragma omp task`), которая параллельно обрабатывается одним из доступных потоков (внутри общего пула).
3. Когда все задачи (блоки) для одного этапа (`relax` или `resid`) завершены, код собирает результаты:
   - либо это максимум `eps`,
   - либо просто завершение фазы без редукции.

Преимущество подхода в том, что распараллеливание более гибкое: не просто каждый поток выполняет часть строк, а каждый поток может взять любую из доступных задач, как только освободится.

## 9.2 Первый блок: обновление матрицы $B$ (аналог `relax`)

```
int blocksCount1 = 0;
for(int i = 2; i < N - 2; i += BLOCK_SIZE)
{
    for(int j = 2; j < N - 2; j += BLOCK_SIZE)
    {
```

```
            blocksCount1++;
        }
    }

    double *max_per_task1 = (double*)calloc(blocksCount1, sizeof(double));

    int task_id1 = 0;
    #pragma omp taskgroup
    for(int i = 2; i < N - 2; i += BLOCK_SIZE)
    {
        int i2 = (i + BLOCK_SIZE < N - 2) ? (i + BLOCK_SIZE) : (N - 2);

        for(int j = 2; j < N - 2; j += BLOCK_SIZE)
        {
            int j2 = (j + BLOCK_SIZE < N - 2) ? (j + BLOCK_SIZE) : (N - 2);

            int local_id = task_id1++;
            #pragma omp task firstprivate(i, j, i2, j2, local_id) \
                             shared(A, B, max_per_task1)
            {
                for(int ii = i; ii < i2; ii++)
                {
                    for(int jj = j; jj < j2; jj++)
                    {
                        B[ii][jj] = (
                            A[ii-2][jj] + A[ii-1][jj] +
                            A[ii][jj-2] + A[ii][jj+1] +
                            A[ii][jj+2] + A[ii+1][jj] +
                            A[ii+2][jj]
                        ) / 8.0;

                        max_per_task1[local_id] = 0.0;
                    }
```

```
            }
        }
    }
}
free(max_per_task1);
```

- Подсчет числа блоков (`blocksCount1`) нужен, чтобы потом завести массив `max_per_task1`, если бы внутри этих задач нужно было еще что-то аккумулировать и собирать. Здесь, по сути, он не особо используется, но оставлен для единообразия.
- `#pragma omp taskgroup` означает, что все задачи, созданные внутри этого блока, должны завершиться до выхода из `taskgroup`. То есть запланированные задачи будут обязательно выполнены до того, как поток перейдет дальше.
- Сама задача (внутри `#pragma omp task`) обрабатывает блок размером $[i..i2] \times [j..j2]$ и вычисляет матрицу $B$ по формуле усреднения.

## 9.3 Второй блок: обновление $A$ и вычисление `eps` (аналог `resid`).

```
int blocksCount2 = 0;
for(int i = 1; i < N - 1; i += BLOCK_SIZE)
{
    for(int j = 1; j < N - 1; j += BLOCK_SIZE)
    {
        blocksCount2++;
    }
}

double *max_per_task2 = (double*)calloc(blocksCount2, sizeof(double));

int task_id2 = 0;
#pragma omp taskgroup
for(int i = 1; i < N - 1; i += BLOCK_SIZE)
{
    int i2 = (i + BLOCK_SIZE < N - 1) ? (i + BLOCK_SIZE) : (N - 1);
```

```
    for(int j = 1; j < N - 1; j += BLOCK_SIZE)
    {
        int j2 = (j + BLOCK_SIZE < N - 1) ? (j + BLOCK_SIZE) : (N - 1);

        int local_id = task_id2++;
        #pragma omp task firstprivate(i, j, i2, j2, local_id) \
                         shared(A, B, max_per_task2)
        {
            double local_max = 0.0;
            for(int ii = i; ii < i2; ii++)
            {
                for(int jj = j; jj < j2; jj++)
                {
                    double e = fabs(A[ii][jj] - B[ii][jj]);
                    A[ii][jj] = B[ii][jj];
                    if(e > local_max)
                        local_max = e;
                }
            }
            max_per_task2[local_id] = local_max;
        }
    }
}
double global_max = 0.0;
for(int i = 0; i < blocksCount2; i++)
{
    global_max = Max(global_max, max_per_task2[i]);
}
free(max_per_task2);

eps = global_max;
```

- То же самое, но теперь каждый блок вычисляет $e = |A - B|$, обновляет $A$ и

ищет локальный максимум внутри блока.

- Каждая задача возвращает локальный максимум в `max_per_task2[local_id]`.
- После всех задач (по выходу из `taskgroup`), программа пробегается по всем элементам `max_per_task2[]`, чтобы найти глобальный максимум `eps`.

## 9.4 Логика цикла по `it`.

Все упомянутое выше запаковано в цикл:

```
for(int it = 1; it <= itmax; it++)
{
    #pragma omp single
    {
        eps = 0.0;

        //relax
        ...
        //resid
        ...

        eps = global_max;
        if (eps < maxeps) ...
    }
}
```

- `#pragma omp single`: только один поток генерирует задачи для всех блоков. Остальные потоки, тем временем, не простаивают, а могут сразу брать эти задачи из очереди задач и начинать их выполнять.
- По окончании `single` блока (который включает в себя `taskgroup`) все задачи будут завершены, и мы движемся к следующей итерации `it`.

## 9.5 Зачем нужен `taskgroup`?

`taskgroup` гарантирует, что все задачи, созданные внутри, завершаются до выхода из `taskgroup`. Без `taskgroup` задачи могли бы убежать дальше, в то время как код

приступил бы к следующему этапу.

Это важно, чтобы на момент, когда мы собираем `eps` или переходим к следующему итерационному шагу, все блоки точно были посчитаны.

### Почему так можно параллелить?

- **Блочное разбиение.** Мы разбиваем диапазон $[2..N-2]$ (или $[1..N-1]$) на блоки `BLOCK_SIZE = 128`. Для больших $N$ это позволяет одновременно иметь много мелких задач.
- **OpenMP Tasks.** Каждый блок — независимое вычисление. Поэтому задачи могут выполняться параллельно на разных потоках.
- Поскольку все блоки независимы, OpenMP не требует дополнительной синхронизации на уровне каждой задачи. Только в конце (в `taskgroup`) нужно дождаться, чтобы все задачи завершились, прежде чем переходить к следующему шагу.

## 9.6 Сравнение с `parallel for`.

1. `parallel for`
   - Разбивает кусочки цикла по строкам (или по $i$-интервалам) статически или динамически.
   - Проще писать, но не всегда гибко при более сложных шаблонах доступа к данным.
2. `task` + блоки
   - Дает возможность делить задачу на блоки произвольной геометрии.
   - Каждый блок обрабатывается как единица работы (`task`), а OpenMP сама распределяет эти задачи между потоками.
   - Более гибкая балансировка.
   - Код чуть сложнее: нужно вручную считать число блоков, создавать массивы для локальных результатов, аккуратно их собирать.

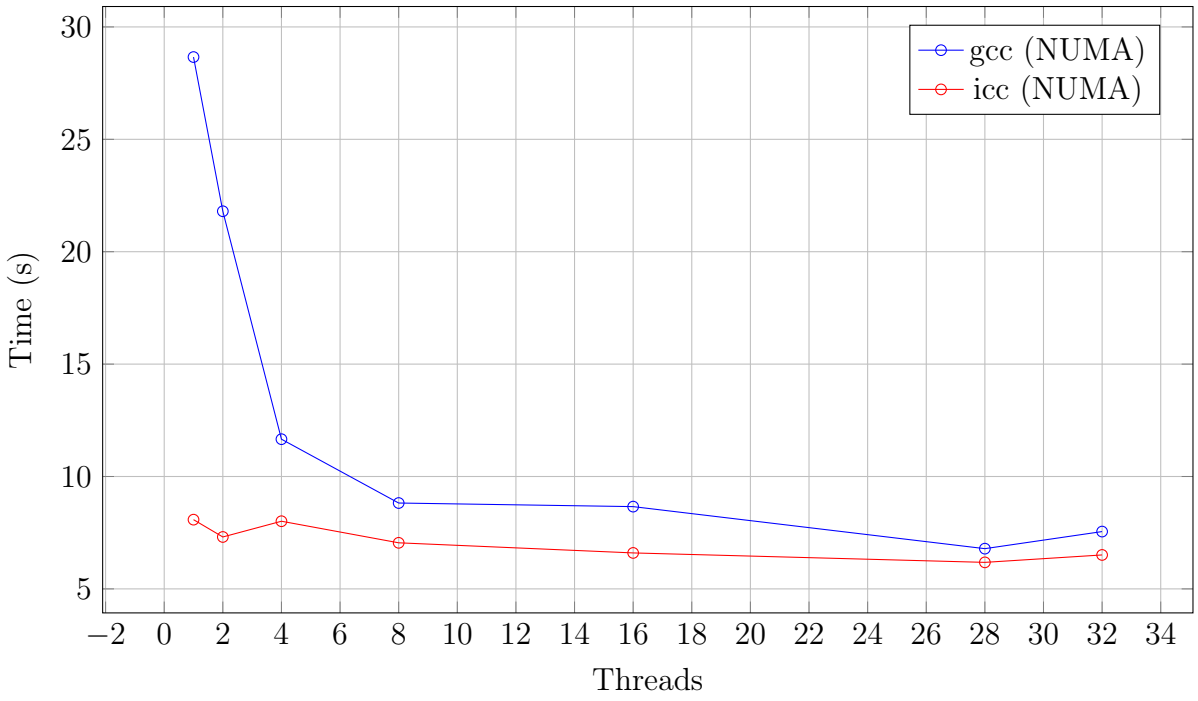
## 9.7 Результаты выполнения.

| Потоки | gcc(NUMA) | | icc(NUMA) | |
|---|---|---|---|---|
| | S | Time (s) | S | Time (s) |
| 1 | 22667151283.233 | 28.66 | 22667151283.232 | 8.08 |
| 2 | 22667151283.231 | 21.80 | 22667151283.233 | 7.31 |
| 4 | 22667151283.233 | 11.66 | 22667151283.233 | 8.01 |
| 8 | 22667151283.233 | 8.82 | 22667151283.233 | 7.05 |
| 16 | 22667151283.233 | 8.66 | 22667151283.233 | 6.60 |
| 28 | 22667151283.233 | 6.79 | 22667151283.233 | 6.18 |
| 32 | 22667151283.233 | 7.55 | 22667151283.233 | 6.51 |

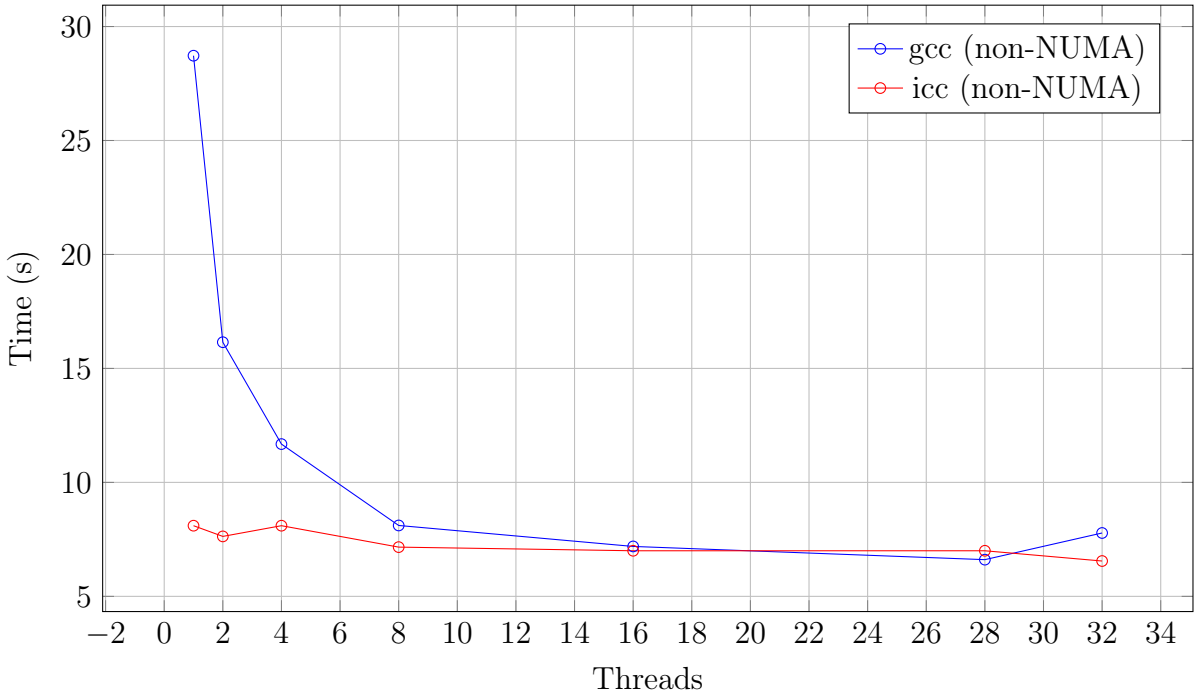
| Потоки | gcc(non-NUMA) | | icc(non-NUMA) | |
|---|---|---|---|---|
| | S | Time (s) | S | Time (s) |
| 1 | 22667151283.233 | 28.72 | 22667151283.232 | 8.10 |
| 2 | 22667151283.231 | 16.15 | 22667151283.233 | 7.63 |
| 4 | 22667151283.233 | 11.68 | 22667151283.233 | 8.10 |
| 8 | 22667151283.233 | 8.11 | 22667151283.233 | 7.16 |
| 16 | 22667151283.233 | 7.19 | 22667151283.233 | 7.60 |
| 28 | 22667151283.233 | 6.61 | 22667151283.233 | 6.23 |
| 32 | 22667151283.233 | 7.78 | 22667151283.233 | 6.55 |

## 9.8 Графики сравнения.

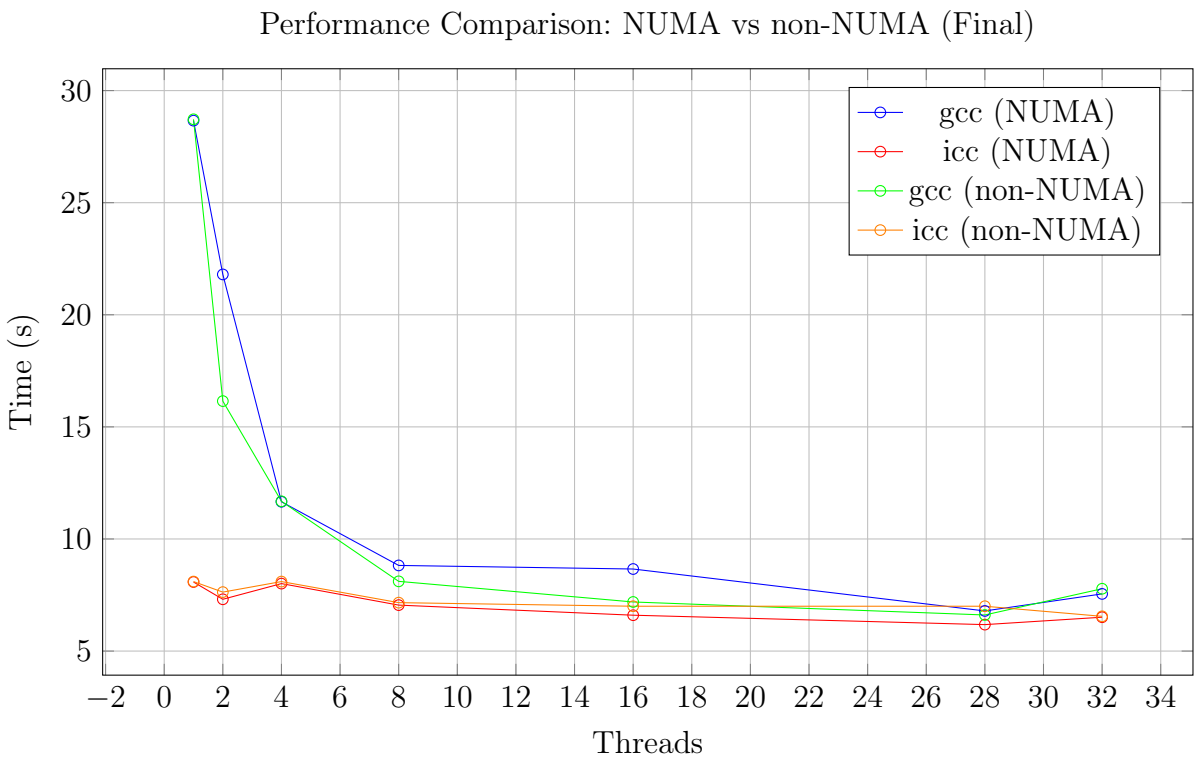
### 9.8.1 Производительность с NUMA.




**Производительность без NUMA**



### 9.8.2 Сравнение NUMA и non-NUMA.




## 9.9 Выводы.

- Пример демонстрирует способ параллелизации: вместо классического `for` мы вручную разруливаем блоки и используем `task`-модель.
- Это удобно, когда структура данных / область итераций может быть фрагментирована или обрабатывается порциями.
- За счет `taskgroup` мы синхронизируемся после обработки всех блоков.
- Важно, что каждый блок вычисляется независимо от других, и нет гонок при записи. Иначе потребовались бы дополнительные механизмы синхронизации или `depend`-клавзу в задачах.

Таким образом, с этим тоже можно означать, что программа успешно параллелится с помощью OpenMP-задач (`pragma task`) и блочного разбиения, аналогично тому, как мы обычно делаем `parallel for`, но с большей гибкостью.

## 9.10 Итоги.

1. **Заметно более высокое время** при использовании OpenMP `tasks` (против `#pragma omp for`) для этой конкретной задачи.

- На малом числе потоков (1–2–4) время может быть в разы выше.
- Да, при росте потоков все еще происходит параллелизация, но потолок производительности ниже, чем в варианте с классическими параллельными циклами.

2. **Оптимальное число потоков:** в большинстве случаев 28 дает наименьшее время. Это похоже на результаты с `for`, только абсолютные значения времени выше.

3. **Причины отставания `tasks`:**
   - Для шаблонной двумерной итерации `#pragma omp for` очень хорошо оптимизирован и имеет низкие накладные расходы.
   - OpenMP `tasks` лучше работают в ситуациях, где есть **неровная** или динамическая структура вычислений (например, рекурсивные задачи, адаптивные сетки и т.д.). Но если у нас просто прямой цикл по элементам массива, задачный подход зачастую дает избыточный оверхед.
   - В результате мы видим, что при одинаковом числе потоков `tasks`-подход почти всегда медленнее, чем `for`-подход, причем иногда сильно медленнее.

**Итог:** для задачи с однородным циклом классический `omp for` показывает себя эффективнее, чем OpenMP `tasks`. У `tasks` есть преимущества для нерегулярных вычислительных графов, но здесь, судя по всему, накладные расходы дают заметный проигрыш по времени.

# 10 Приложение. Программы.

## 10.1 Основная программа.

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#define  Max(a,b) ((a)>(b)?(a):(b))

#define  N   (2*2*2*2*2*2+2)
double   maxeps = 0.1e-7;
```

```
int itmax = 100;
int i,j,k;
double eps;
double A [N][N],  B [N][N];

void relax();
void resid();
void init();
void verify();

int main(int an, char **as)
{
 int it;
 init();
 for(it=1; it<=itmax; it++)
 {
  eps = 0.;
  relax();
  resid();
  printf( "it=%4i   eps=%f\n", it,eps);
  if (eps < maxeps) break;
 }
 verify();
 return 0;
}

void init()
{
 for(j=0; j<=N-1; j++)
 for(i=0; i<=N-1; i++)
 {
  if(i==0 || i==N-1 || j==0 || j==N-1) A[i][j]= 0.;
  else A[i][j]= ( 1. + i + j ) ;
 }
```

```
}

void relax()
{
 for(j=2; j<=N-3; j++)
 for(i=2; i<=N-3; i++)
 {
  B[i][j]=(A[i-2][j]+A[i-1][j]+A[i+2][j]+A[i+1][j]+
        A[i][j-2]+A[i][j-1]+A[i][j+2]+A[i][j+1])/8.;
 }
}

void resid()
{
 for(j=1; j<=N-2; j++)
 for(i=1; i<=N-2; i++)
 {
  double e;
  e = fabs(A[i][j] - B[i][j]);
  A[i][j] = B[i][j];
  eps = Max(eps,e);
 }
}

void verify()
{
 double s;
 s=0.;
 for(j=0; j<=N-1; j++)
 for(i=0; i<=N-1; i++)
 {
  s=s+A[i][j]*(i+1)*(j+1)/(N*N);
 }
 printf("  S = %f\n",s);
```

```
}
```

## 10.2 Оптимизированная AXV.

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <immintrin.h>

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

double maxeps = 0.1e-7;
int itmax = 100;
double eps;
double A[N][N], B[N][N];

void relax();
void resid();
void init();
void verify();

int main(int argc, char **argv)
{
    int it;
    double time_start, time_fin;

    init();

    time_start = clock() / (double)CLOCKS_PER_SEC;

    for (it = 1; it <= itmax; it++)
```

```
    {
        eps = 0.0;
        relax();
        resid();
        printf("it=%4i   eps=%f\n", it, eps);
        if (eps < maxeps) break;
    }

    verify();

    time_fin = clock() / (double)CLOCKS_PER_SEC;
    printf("Time: %g sec\n", time_fin - time_start);
    printf("NxN: %i\nN: %i\n", N*N, N);

    return 0;
}

void init()
{
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            A[i][j] = (i == 0 || i == N - 1 || j == 0 || j == N - 1)
            ? 0.0 : (1.0 + i + j);
}

void relax()
{
    for (int i = 2; i < N - 2; i++)
    {
        for (int j = 2; j < N - 2; j += 4)
        {
            __m256d ai_2j = _mm256_loadu_pd(&A[i - 2][j]);
            __m256d ai_1j = _mm256_loadu_pd(&A[i - 1][j]);
            __m256d ai_2jp2 = _mm256_loadu_pd(&A[i + 2][j]);
```

```
            __m256d ai_1jp1 = _mm256_loadu_pd(&A[i + 1][j]);

            __m256d aijm2 = _mm256_loadu_pd(&A[i][j - 2]);
            __m256d aijm1 = _mm256_loadu_pd(&A[i][j - 1]);
            __m256d aijp2 = _mm256_loadu_pd(&A[i][j + 2]);
            __m256d aijp1 = _mm256_loadu_pd(&A[i][j + 1]);

            __m256d sum = _mm256_add_pd(ai_2j, ai_1j);
            sum = _mm256_add_pd(sum, ai_2jp2);
            sum = _mm256_add_pd(sum, ai_1jp1);
            sum = _mm256_add_pd(sum, aijm2);
            sum = _mm256_add_pd(sum, aijm1);
            sum = _mm256_add_pd(sum, aijp2);
            sum = _mm256_add_pd(sum, aijp1);

            __m256d avg = _mm256_div_pd(sum, _mm256_set1_pd(8.0));
            _mm256_storeu_pd(&B[i][j], avg);
        }
    }
}

void resid()
{
    eps = 0.0;
    __m256d max_eps = _mm256_set1_pd(0.0);

    for (int i = 1; i < N - 1; i++)
    {
        for (int j = 1; j < N - 1; j += 4)
        {
            __m256d aij = _mm256_loadu_pd(&A[i][j]);
            __m256d bij = _mm256_loadu_pd(&B[i][j]);

            __m256d diff = _mm256_sub_pd(aij, bij);
```

```
            __m256d abs_diff = _mm256_andnot_pd(_mm256_set1_pd(-0.0), diff);

            max_eps = _mm256_max_pd(max_eps, abs_diff);
            _mm256_storeu_pd(&A[i][j], bij);
        }
    }

    double temp_eps[4];
    _mm256_storeu_pd(temp_eps, max_eps);
    for (int idx = 0; idx < 4; idx++) {
        eps = Max(eps, temp_eps[idx]);
    }
}

void verify()
{
    double s = 0.0;
    double norm_factor = 1.0 / (N * N);

    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j += 4)
        {
            __m256d aij = _mm256_loadu_pd(&A[i][j]);
            __m256d indices = _mm256_set_pd((i + 1) * (j + 4), (i + 1) *
            (j + 3), (i + 1) * (j + 2), (i + 1) * (j + 1));
            __m256d prod = _mm256_mul_pd(aij, indices);
            __m256d scaled_prod =
            _mm256_mul_pd(prod, _mm256_set1_pd(norm_factor));

            double temp_s[4];
            _mm256_storeu_pd(temp_s, scaled_prod);
            s += temp_s[0] + temp_s[1] + temp_s[2] + temp_s[3];
        }
```

```
    }
    printf("  S = %f\n", s);
}
```

## 10.3 AVX-for.

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

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

double maxeps = 0.1e-7;
int itmax = 100;
double eps;
double A[N][N], B[N][N];

void relax();
void resid();
void init();
void verify();

int main(int argc, char **argv)
{
    int it;
    double time_start, time_fin;

    init();

    time_start = omp_get_wtime();
```

```
    for (it = 1; it <= itmax; it++)
    {
        eps = 0.0;

        #pragma omp parallel
        {
            relax();
            resid();
        }
        printf("it=%4i   eps=%f\n", it, eps);
        if (eps < maxeps) break;
    }

    verify();

    time_fin = omp_get_wtime();
    printf("Time: %g sec\n", time_fin - time_start);
    printf("NxN: %i\nN: %i\n", N*N, N);

    return 0;
}

void init()
{
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            A[i][j] = (i == 0 || i == N - 1 || j == 0 || j == N - 1)
            ? 0.0 : (1.0 + i + j);
}

void relax()
{
    #pragma omp for collapse(2)
```

```
    for (int i = 2; i < N - 2; i++)
    {
        for (int j = 2; j < N - 2; j += 4)
        {
            __m256d ai_2j = _mm256_loadu_pd(&A[i - 2][j]);
            __m256d ai_1j = _mm256_loadu_pd(&A[i - 1][j]);
            __m256d ai_2jp2 = _mm256_loadu_pd(&A[i + 2][j]);
            __m256d ai_1jp1 = _mm256_loadu_pd(&A[i + 1][j]);

            __m256d aijm2 = _mm256_loadu_pd(&A[i][j - 2]);
            __m256d aijm1 = _mm256_loadu_pd(&A[i][j - 1]);
            __m256d aijp2 = _mm256_loadu_pd(&A[i][j + 2]);
            __m256d aijp1 = _mm256_loadu_pd(&A[i][j + 1]);

            __m256d sum = _mm256_add_pd(ai_2j, ai_1j);
            sum = _mm256_add_pd(sum, ai_2jp2);
            sum = _mm256_add_pd(sum, ai_1jp1);
            sum = _mm256_add_pd(sum, aijm2);
            sum = _mm256_add_pd(sum, aijm1);
            sum = _mm256_add_pd(sum, aijp2);
            sum = _mm256_add_pd(sum, aijp1);

            __m256d avg = _mm256_div_pd(sum, _mm256_set1_pd(8.0));
            _mm256_storeu_pd(&B[i][j], avg);
        }
    }
}

void resid()
{
    eps = 0.0;
    __m256d max_eps = _mm256_set1_pd(0.0);

    #pragma omp for collapse(2)
```

```
    for (int i = 1; i < N - 1; i++)
    {
        for (int j = 1; j < N - 1; j += 4)
        {
            __m256d aij = _mm256_loadu_pd(&A[i][j]);
            __m256d bij = _mm256_loadu_pd(&B[i][j]);

            __m256d diff = _mm256_sub_pd(aij, bij);
            __m256d abs_diff =
            _mm256_andnot_pd(_mm256_set1_pd(-0.0), diff);

            max_eps = _mm256_max_pd(max_eps, abs_diff);

            _mm256_storeu_pd(&A[i][j], bij);
        }
    }
    double temp_eps[4];
    _mm256_storeu_pd(temp_eps, max_eps);

    for (int idx = 0; idx < 4; idx++)
    {
        eps = Max(eps, temp_eps[idx]);
    }
    #pragma omp barrier
}

void verify()
{
    double s = 0.0;
    double norm_factor = 1.0 / (N * N);

    #pragma omp parallel for reduction(+:s)
    for (int i = 0; i < N; i++)
    {
```

```
        for (int j = 0; j < N; j += 4)
        {
            __m256d aij = _mm256_loadu_pd(&A[i][j]);
            __m256d indices = _mm256_set_pd((i + 1) * (j + 4), (i + 1) *
            (j + 3), (i + 1) * (j + 2), (i + 1) * (j + 1));
            __m256d prod = _mm256_mul_pd(aij, indices);
            __m256d scaled_prod =
            _mm256_mul_pd(prod, _mm256_set1_pd(norm_factor));

            double temp_s[4];
            _mm256_storeu_pd(temp_s, scaled_prod);
            s += temp_s[0] + temp_s[1] + temp_s[2] + temp_s[3];
        }
    }
    printf("  S = %f\n", s);
}
```

## 10.4 AVX-task.

```
#include <stdio.h>
#include <omp.h>
#include <immintrin.h>

#define Max(a, b) ((a) > (b) ? (a) : (b))
#define  N   (2*2*2*2*2*2*2*2*2*2*2*2+2)
#define BLOCK_SIZE 32

double maxeps = 0.1e-7;
int itmax = 100;
double eps;
double A[N][N], B[N][N];

void relax();
```

```
void resid();
void init();
void verify();

int main(int argc, char **argv)
{
    int it;
    double time_start, time_fin;

    init();

    time_start = omp_get_wtime();

    #pragma omp parallel
    {
        #pragma omp single
        {
            for (it = 1; it <= itmax; it++)
            {
                eps = 0.0;

                relax();
                resid();

                printf("it=%4i   eps=%f\n", it, eps);
                if (eps < maxeps) break;
            }
        }
    }

    verify();

    time_fin = omp_get_wtime();
    printf("Time: %g sec\n", time_fin - time_start);
```

```
    printf("NxN: %i\nN: %i\n", N*N, N);

    return 0;
}

void init()
{
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            A[i][j] = (i == 0 || i == N - 1 || j == 0 || j == N - 1)
            ? 0.0 : (1.0 + i + j);
}

void relax()
{
    #pragma omp taskgroup
    {
        for (int i = 2; i < N - 2; i += BLOCK_SIZE)
        {
            for (int j = 2; j < N - 2; j += BLOCK_SIZE)
            {
                int i_end = (i + BLOCK_SIZE < N - 2) ? i +
                BLOCK_SIZE : N - 2;
                int j_end = (j + BLOCK_SIZE < N - 2) ? j +
                BLOCK_SIZE : N - 2;
                int i_start = i;
                int j_start = j;
                #pragma omp task firstprivate(i_start, i_end, j_start, j_end)
                {
                    for (int ii = i_start; ii < i_end; ii++)
                    {
                        for (int jj = j_start; jj < j_end; jj += 4)
                        {
                            __m256d ai_2j = _mm256_loadu_pd(&A[ii - 2][jj]);
```

```
                            __m256d ai_1j = _mm256_loadu_pd(&A[ii - 1][jj]);
                            __m256d ai_2jp2 = _mm256_loadu_pd(&A[ii + 2][jj]);
                            __m256d ai_1jp1 = _mm256_loadu_pd(&A[ii + 1][jj]);

                            __m256d aijm2 = _mm256_loadu_pd(&A[ii][jj - 2]);
                            __m256d aijm1 = _mm256_loadu_pd(&A[ii][jj - 1]);
                            __m256d aijp2 = _mm256_loadu_pd(&A[ii][jj + 2]);
                            __m256d aijp1 = _mm256_loadu_pd(&A[ii][jj + 1]);

                            __m256d sum = _mm256_add_pd(ai_2j, ai_1j);
                            sum = _mm256_add_pd(sum, ai_2jp2);
                            sum = _mm256_add_pd(sum, ai_1jp1);
                            sum = _mm256_add_pd(sum, aijm2);
                            sum = _mm256_add_pd(sum, aijm1);
                            sum = _mm256_add_pd(sum, aijp2);
                            sum = _mm256_add_pd(sum, aijp1);

                            __m256d avg = _mm256_div_pd(sum, _mm256_set1_pd(8.0));
                            _mm256_storeu_pd(&B[ii][jj], avg);
                        }
                    }
                }
            }
        }
    }
}

void resid()
{
    eps = 0.0;

    #pragma omp taskgroup
    {
        for (int i = 1; i < N - 1; i += BLOCK_SIZE)
```

```
{
    for (int j = 1; j < N - 1; j += BLOCK_SIZE)
    {
        int i_end = (i + BLOCK_SIZE < N - 1)
        ? i + BLOCK_SIZE : N - 1;
        int j_end = (j + BLOCK_SIZE < N - 1)
        ? j + BLOCK_SIZE : N - 1;
        int i_start = i;
        int j_start = j;
        #pragma omp task firstprivate(i_start, i_end, j_start, j_end)
        {
            __m256d local_max_eps = _mm256_set1_pd(0.0);
            for (int ii = i_start; ii < i_end; ii++)
            {
                for (int jj = j_start; jj < j_end; jj += 4)
                {
                    __m256d aij = _mm256_loadu_pd(&A[ii][jj]);
                    __m256d bij = _mm256_loadu_pd(&B[ii][jj]);

                    __m256d diff = _mm256_sub_pd(aij, bij);
                    __m256d abs_diff = _mm256_andnot_pd(
                    _mm256_set1_pd(-0.0), diff);

                    local_max_eps = _mm256_max_pd(local_max_eps,
                    abs_diff);

                    _mm256_storeu_pd(&A[ii][jj], bij);
                }
            }
            double temp_eps[4];
            _mm256_storeu_pd(temp_eps, local_max_eps);

            double local_eps = 0.0;
            for (int idx = 0; idx < 4; idx++)
```

```
                    {
                        local_eps = Max(local_eps, temp_eps[idx]);
                    }

                    //#pragma omp critical
                   // {
                        eps = Max(eps, local_eps);
                   // }
                }
            }
        }
    }
}

void verify()
{
    double s = 0.0;
    double norm_factor = 1.0 / (N * N);

    #pragma omp parallel for reduction(+:s)
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j += 4)
        {
            __m256d aij = _mm256_loadu_pd(&A[i][j]);
            __m256d indices = _mm256_set_pd((i + 1) *
            (j + 4), (i + 1) * (j + 3), (i + 1) * (j + 2), (i + 1) *
            (j + 1));
            __m256d prod = _mm256_mul_pd(aij, indices);
            __m256d scaled_prod =
            _mm256_mul_pd(prod, _mm256_set1_pd(norm_factor));

            double temp_s[4];
            _mm256_storeu_pd(temp_s, scaled_prod);
```

```
            s += temp_s[0] + temp_s[1] + temp_s[2] + temp_s[3];
        }
    }
    printf("  S = %f\n", s);
}
```

## 10.5 for.

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

static inline unsigned long long rdtsc(void)
{
    unsigned int lo, hi;
    __asm__ __volatile__ ("rdtsc" : "=a"(lo), "=d"(hi));
    return ((unsigned long long)hi << 32) | lo;
}

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

static alignas(64) double A[N][N];
static alignas(64) double B[N][N];

static const double maxeps = 0.1e-7;
static int itmax = 100;

static double eps;
void init_data(void);
void relax(void);
```

```
void resid(void);
void verify(void);

static int get_chunk_size(void)
{
    int nThreads = omp_get_max_threads();
    int chunk = 8 * nThreads;
    if (chunk < 1) chunk = 1;
    return chunk;
}

int main(void)
{
    unsigned long long start_ticks = rdtsc();
    double t_start = omp_get_wtime();

    init_data();

    #pragma omp parallel
    {
        int chunk = get_chunk_size();

        for(int it = 1; it <= itmax; it++)
        {
            #pragma omp single
            {
                eps = 0.0;
            }
            #pragma omp for schedule(static, chunk) nowait
            for(int i = 2; i < N - 2; i++)
            {
                for(int j = 2; j < N - 2; j++)
                {
                    B[i][j] = (
```

```
                    A[i-2][j] + A[i-1][j] + A[i+1][j] + A[i+2][j]
                  + A[i][j-2] + A[i][j-1] + A[i][j+1] + A[i][j+2]
                ) / 8.0;
            }
        }
        #pragma omp barrier
        #pragma omp for schedule(static, chunk) reduction(max:eps)
        for(int i = 1; i < N - 1; i++)
        {
            for(int j = 1; j < N - 1; j++)
            {
                double e = fabs(A[i][j] - B[i][j]);
                A[i][j]   = B[i][j];
                eps       = Max(eps, e);
            }
        }
        #pragma omp single
        {
           // printf("it=%4d   eps=%e\n", it, eps);
            if (eps < maxeps)
            {
                itmax = it - 1;
            }
        }
    }
}
verify();

unsigned long long end_ticks = rdtsc();
unsigned long long diff = end_ticks - start_ticks;
double t_end = omp_get_wtime();

printf("ticks: %llu\n", diff);
printf("OMP_WTIME: %f sec\n", (t_end - t_start));
```

```
    return 0;
}
void init_data(void)
{
    int chunk = get_chunk_size();
    #pragma omp parallel for schedule(static, chunk)
    for(int i = 0; i < N; i++)
    {
        for(int j = 0; j < N; j++)
        {
            if (i == 0 || i == N-1 || j == 0 || j == N-1)
                A[i][j] = 0.0;
            else
                A[i][j] = 1.0 + i + j;
        }
    }
}
void relax(void){}
void resid(void){}
void verify(void)
{
    double s = 0.0;
    int chunk = get_chunk_size();

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

```
}
```

## 10.6 task.

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

static inline unsigned long long rdtsc(void)
{
    unsigned int lo, hi;
    __asm__ __volatile__ ("rdtsc" : "=a"(lo), "=d"(hi));
    return ((unsigned long long)hi << 32) | lo;
}

#define Max(a,b) ((a)>(b)?(a):(b))
#define BLOCK_SIZE 128

#define N (2*2*2*2*2*2*2*2*2*2*2*2+2)

static alignas(64) double A[N][N];
static alignas(64) double B[N][N];

static const double maxeps = 0.1e-7;
static int itmax = 100;
static double eps;
void init_data(void);
void verify(void);

int main(void)
```

```
{
    unsigned long long start_ticks = rdtsc();
    double t_start = omp_get_wtime();

    init_data();

    #pragma omp parallel
    {
        for(int it = 1; it <= itmax; it++)
        {
            #pragma omp single
            {
                eps = 0.0;

                int blocksCount1 = 0;
                for(int i = 2; i < N - 2; i += BLOCK_SIZE)
                {
                    for(int j = 2; j < N - 2; j += BLOCK_SIZE)
                    {
                        blocksCount1++;
                    }
                }

                double *max_per_task1 =
                (double*)calloc(blocksCount1, sizeof(double));

                int task_id1 = 0;
                #pragma omp taskgroup
                {
                    for(int i = 2; i < N - 2; i += BLOCK_SIZE)
                    {
                        int i2 = (i + BLOCK_SIZE < N - 2)
                        ? (i + BLOCK_SIZE) : (N - 2);
```

```
        for(int j = 2; j < N - 2; j += BLOCK_SIZE)
        {
            int j2 = (j + BLOCK_SIZE < N - 2)
            ? (j + BLOCK_SIZE) : (N - 2);

            int local_id = task_id1++;
            #pragma omp task
            firstprivate(i, j, i2, j2, local_id)
            shared(A, B, max_per_task1)
            {
                for(int ii = i; ii < i2; ii++)
                {
                    for(int jj = j; jj < j2; jj++)
                    {
                        B[ii][jj] = (
                              A[ii-2][jj] +
                              A[ii-1][jj] +
                              A[ii+1][jj] +
                              A[ii+2][jj]
                            + A[ii][jj-2] +
                            A[ii][jj-1] +
                            A[ii][jj+1] +
                            A[ii][jj+2]
                        ) / 8.0;
                    }
                }
                max_per_task1[local_id] = 0.0;
            }
        }
    }
}
free(max_per_task1);

int blocksCount2 = 0;
```

```
for(int i = 1; i < N - 1; i += BLOCK_SIZE)
{
    for(int j = 1; j < N - 1; j += BLOCK_SIZE)
    {
        blocksCount2++;
    }
}
double *max_per_task2 =
(double*)calloc(blocksCount2, sizeof(double));

int task_id2 = 0;
#pragma omp taskgroup
{
    for(int i = 1; i < N - 1; i += BLOCK_SIZE)
    {
        int i2 = (i + BLOCK_SIZE < N - 1)
        ? (i + BLOCK_SIZE) : (N - 1);

        for(int j = 1; j < N - 1; j += BLOCK_SIZE)
        {
            int j2 = (j + BLOCK_SIZE < N - 1)
            ? (j + BLOCK_SIZE) : (N - 1);

            int local_id = task_id2++;
            #pragma omp task
            firstprivate(i, j, i2, j2, local_id)
            shared(A, B, max_per_task2)
            {
                double local_max = 0.0;
                for(int ii = i; ii < i2; ii++)
                {
                    for(int jj = j; jj < j2; jj++)
                    {
                        double e = fabs(A[ii][jj] - B[ii][jj]);
```

```
                                    A[ii][jj] = B[ii][jj];
                                    if (e > local_max)
                                    {
                                        local_max = e;
                                    }
                                }
                            }
                            max_per_task2[local_id] = local_max;
                        }
                    }
                }
            }

            double global_max = 0.0;
            for(int i = 0; i < blocksCount2; i++)
            {
                global_max = Max(global_max, max_per_task2[i]);
            }
            free(max_per_task2);

            eps = global_max;
            // printf("it = %4d   eps = %e\n", it, eps);
            if (eps < maxeps)
            {
                itmax = it - 1;
            }
        }
    }
}

verify();

unsigned long long end_ticks = rdtsc();
double t_end = omp_get_wtime();
```

```
    unsigned long long diff = end_ticks - start_ticks;
    printf("ticks: %llu\n", diff);
    printf("OMP_WTIME: %f sec\n", (t_end - t_start));

    return 0;
}

void init_data(void)
{
    #pragma omp parallel for
    for(int i = 0; i < N; i++)
    {
        for(int j = 0; j < N; j++)
        {
            if (i == 0 || i == N-1 || j == 0 || j == N-1)
                A[i][j] = 0.0;
            else
                A[i][j] = 1.0 + i + j;
        }
    }
}

void verify(void)
{
    double s = 0.0;
    #pragma omp parallel
    {
        double s_local = 0.0;
        #pragma omp for nowait
        for(int i = 0; i < N; i++)
        {
            for(int j = 0; j < N; j++)
            {
```

```
                s_local += A[i][j] * (i + 1) * (j + 1)
                / (double)(N*N);
            }
        }
        #pragma omp atomic
        s += s_local;
    }
    printf("S = %f\n", s);
}
```