ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО
ОБРАЗОВАНИЯ
«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

КУРСОВОЙ ПРОЕКТ

по дисциплине “Параллельные вычислительные технологии” на тему

Разработка параллельной программы решения 2D-уравнения теплопроводности методом двумерной декомпозиции расчетной области

Выполнил студент Наумов Алексей Александрович
Ф.И.О.

Группы ИС-142

Работу принял _____ профессор д.т.н. М.Г. Курносов
подпись

Защищена_____Оценка _____

Новосибирск – 2023

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Целью данной работы является разработка и реализация параллельной программы решения 2D-уравнения теплопроводности методом двумерной декомпозиции расчетной области. В курсовой рассматривается стационарное уравнение Лапласа с использованием двумерной декомпозиции расчетной области.

Для решения двумерного уравнения Лапласа воспользуемся методом последовательных итераций Якоби. Метод Якоби применяется для последовательного обновления значений потенциала или температуры в узлах двумерной сетки. Основная цель метода заключается в достижении стационарного состояния, при котором разность между последовательными приближениями становится достаточно малой. Процесс итераций продолжается до тех пор, пока не будет достигнута заданная точность или выполнено определенное число итераций.

Данная работа также охватывает параллельную реализацию метода Якоби, направленную на эффективное использование многозадачности и распределение вычислительных задач между несколькими процессами. Это позволяет ускорить процесс решения стационарного двумерного уравнения Лапласа и эффективно использовать вычислительные ресурсы.

1 Этапы решения двумерного уравнения Лапласа:

Решение двумерного уравнения теплопроводности методом Якоби включает несколько основных этапов. Давайте рассмотрим их на основе предоставленного кода:

1. Инициализация MPI и определение сетки процессов:

- Используется MPI для создания коммуникатора и определения размеров сетки процессов.

2. Инициализация и распределение сетки:

- Определение размеров и распределение области вычислений между процессами.

- Выделение памяти для локальных 2D-подсеток с ячейками гало.

3. Инициализация граничных условий:

- Инициализация граничных точек в соответствии с заданными начальными условиями (например, задание температур на границах).

4. Определение соседей и типов границ:

- Определение соседей для обмена данными между процессами.

- Определение типов границ (верхняя, нижняя, левая, правая) для MPI коммуникации.

5. Итерационное решение уравнения:

- Используется цикл для выполнения итераций метода Якоби.

- Обновление внутренних точек в соответствии с уравнением теплопроводности.

6. Проверка условия завершения:

- Проверка разности между последовательными итерациями на достижение заданной точности.

7. Обмен данными между процессами:

- Обмен граничными данными между процессами для учета граничных условий.

8. Вычисление времени выполнения:

- Замер времени выполнения для оценки производительности и эффективности параллельного выполнения.

9. Вывод результатов:

- Вывод результатов, таких как количество итераций, времени выполнения и других характеристик процесса решения.

Каждый из этих этапов включает в себя соответствующие вычислительные и коммуникационные операции.

2 Реализация решения уравнения теплопроводности

Пояснение по алгоритму работы MPI-программы из Приложения 2

2.1 Используемые функции

Функция`**get_block_size**`:- Описание: Функция определяет размер блока данных, который будет обрабатываться каждым процессом.

-Параметры:

-`n`: Общее количество данных.

- `rank`: Ранг текущего процесса.

-`nprocs`:Общее количество процессов.

- Возвращаемое значение: Размер блока данных, обрабатываемого текущим процессом.

Функция `**get_sum_of_prev_blocks**`:

- Описание: Функция определяет сумму размеров предыдущих блоков данных для корректного распределения данных между процессами.

-Параметры:

-`n`:Общее количество данных.

- `rank`: Ранг текущего процесса.

- `nprocs`: Общее количество процессов.

- Возвращаемое значение: Сумма размеров предыдущих блоков данных.

2.2 Основная функция

В функции **main** используются:

Инициализация MPI:

-`**MPI_Init(&argc, &argv)**`: Инициализация MPI.

-`**MPI_Comm_size(MPI_COMM_WORLD, &commsize)**`: Получение общего количества процессов.

-`**MPI_Comm_rank(MPI_COMM_WORLD, &rank)**`: Получение ранга текущего процесса.

- `**MPI_Dims_create(commsize, 2, dims)**`: Создание 2D-решетки процессов.

-`**MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periodic, 0, &cartcomm)**`: Создание коммуникатора для 2D-решетки.

Инициализация сетки и обработка входных данных:

-`**MPI_Bcast(&args, NELEMS(args), MPI_INT, 0, MPI_COMM_WORLD)**`: рассылка размеров сетки по всем процессам.

- Выделение памяти для локальных 2D-подсеток и их инициализация.

Определение граничных условий и соседей:

- Инициализация граничных точек в соответствии с условиями задачи.

- `**MPI_Cart_shift(cartcomm, 0, 1, &left, &right)**`, `**MPI_Cart_shift(cartcomm, 1, 1, &top, &bottom)**`: Определение соседей.

Определение типов данных для границ:

- `**MPI_Type_vector**` и `**MPI_Type_commit**`: Определение типов для передачи данных границ между процессами.

Итерационное решение уравнения теплопроводности:

- Цикл для итераций, в котором обновляются внутренние точки сетки.

Проверка условия завершения:

- Проверка разности между последовательными итерациями на достижение заданной точности.

Обмен данными между процессами:

- `**MPI_Irecv**` и `**MPI_Isend**`: Асинхронный обмен данными между процессами для учета граничных условий.

- `**MPI_Waitall**`: Ожидание завершения асинхронных операций обмена.

Освобождение ресурсов:

- `**MPI_Type_free**`: Освобождение созданных типов данных.

- `**free**`: Освобождение выделенной памяти для локальных сеток.

Вывод результатов и завершение:

- Вывод статистики о времени выполнения, количестве итераций и других характеристик.

- `**MPI_Finalize()**`: Завершение работы MPI.

3 Масштабируемость реализуемой программы

3.1 Результаты запусков на кластере Oak

Таблица 1 - Результаты экспериментов

Количество процессов	Время работы программы для размера матрицы N x N, сек	
	N = 1000	N = 10000
1 (послед.)	5,111	523,074
4 (2 x 2)	1,321	135,632
6 (2 x 3)	0,924	92,214
8 (2 x 4)	0,742	68,400
10 (2 x 5)	0,618	54,536
12 (2 x 6)	0,539	48,053
14 (2 x 7)	0,476	41,124
16 (2 x 8)	0,432	37,277

Таблица 2 – Полученное ускорение относительно последовательной версии

Количество процессов	Ускорение работы программы для размера матрицы N x N относительно последовательной программы (Приложение 1)	
	N = 1000	N = 10000
4 (2 x 2)	3,86	3,85
6 (2 x 3)	5,52	5,67
8 (2 x 4)	6,88	7,64
10 (2 x 5)	8,26	9,59
12 (2 x 6)	9,47	10,88

14 (2 x 7)	10,73	12,71
16 (2 x 8)	11,81	14,03

3.2 Построение результирующего графика

С помощью программы на Python с использованием matplotlib построен график по данным из Таблицы 2.

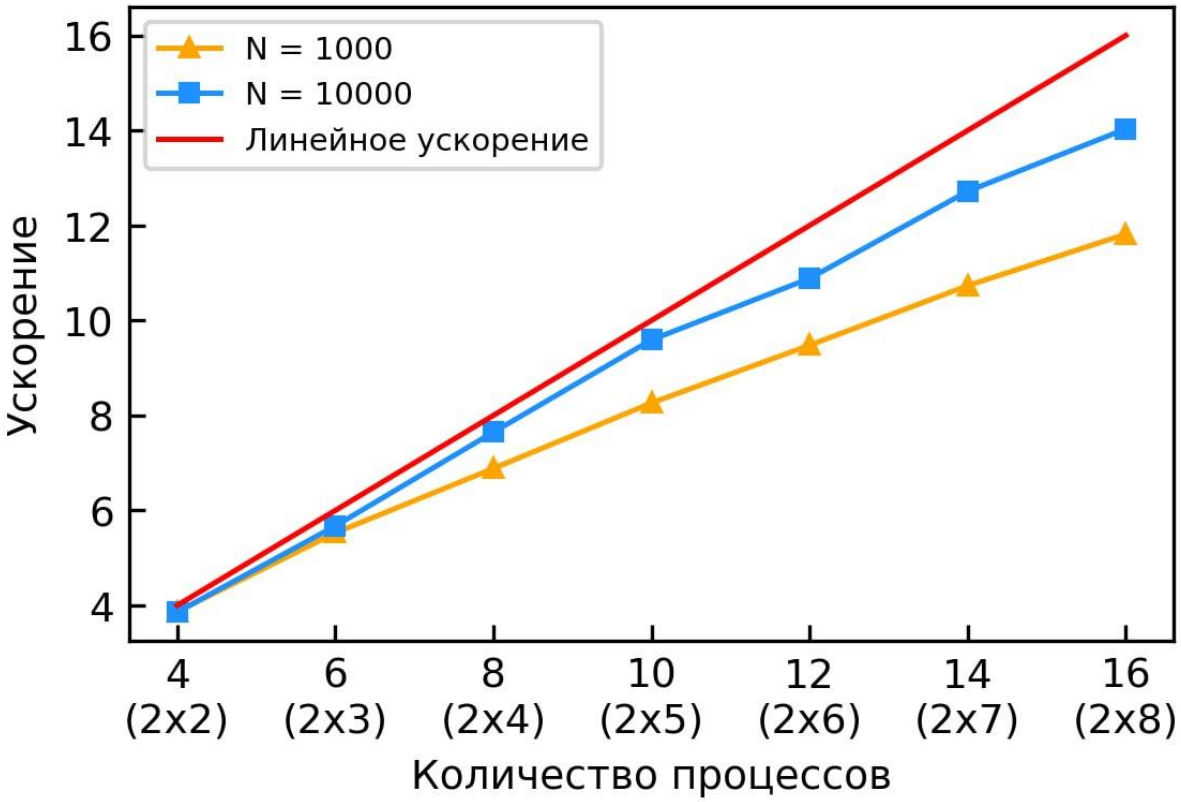

Рисунок 1 - График зависимости ускорения выполнения от количества процессов MPI-программы относительно последовательной версии

ЗАКЛЮЧЕНИЕ

В работе была успешно реализована параллельная программа, осуществляющая решение стационарного двумерного уравнения теплопроводности методом Якоби. Этот метод, основанный на итерационном процессе обновления значений неизвестных переменных, был реализован как в последовательной, так и в параллельной версиях. Программа позволяет эффективно моделировать распределение температуры в заданной области, обеспечивая высокую производительность за счет распределения вычислительных задач между множеством процессов.

Тестирование программы подтвердило эффективность параллельной реализации, продемонстрировав заметное улучшение производительности по сравнению с последовательной версией. Таким образом, результаты данной работы подтверждают применимость параллельных вычислений в решении двумерных уравнений теплопроводности.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Эндрюс Г. Основы многопоточного, параллельного и распределенного программирования

ПРИЛОЖЕНИЕ

1 Исходный код последовательной программы

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

#define IND(i, j) ((i) *nx + (j))

double wtime(void) {
    struct timeval t;
    gettimeofday(&t, NULL);

    return (double) t.tv_sec + (double) t.tv_usec * 1E-6;
}

int main(int argc, char** argv) {
    double ttotal = -wtime();

    const int n = (argc > 1) ? (int) strtol(argv[1], NULL, 10) : 1000;
    const int ny = n;   // rows
    const int nx = ny;  // cols

    double* local_grid = (double*) calloc(ny * nx, sizeof(*local_grid));
    double* local_newgrid = (double*) calloc(ny * nx, sizeof(*local_newgrid));

    const double dx = 1.0 / (nx - 1.0);

    // Initialize top border: u(x, 0) = sin(pi * x)
    for (int j = 0; j < nx; j++) {
        const int ind = IND(0, j);
        local_newgrid[ind] = local_grid[ind] = sin(M_PI * dx * j);
    }

    // Initialize bottom border: u(x, 1) = sin(pi * x) * exp(-pi)
    for (int j = 0; j < nx; j++) {
        const int ind = IND(ny - 1, j);
        local_newgrid[ind] = local_grid[ind] = sin(M_PI * dx * j) * exp(-M_PI);
    }

    const double EPS = 0.001;
    int niters = 0;

    while (1) {
        niters++;

        for (int i = 1; i < ny - 1; i++) {
            // Update interior points
            for (int j = 1; j < nx - 1; j++) {
                local_newgrid[IND(i, j)] = (local_grid[IND(i - 1, j)] + local_grid[IND(i + 1,
j)] +
                                            local_grid[IND(i, j - 1)] + local_grid[IND(i, j +
1)]) *
```

```
                                        0.25;
            }
        }

        // Check termination condition
        double maxdiff = 0.0;

        for (int i = 1; i < ny - 1; i++) {
            for (int j = 1; j < nx - 1; j++) {
                const int ind = IND(i, j);
                maxdiff = fmax(maxdiff, fabs(local_grid[ind] - local_newgrid[ind]));
            }
        }

        // Swap grids (after termination local_grid will contain result)
        double* tmp = local_grid;
        local_grid = local_newgrid;
        local_newgrid = tmp;

        if (maxdiff < EPS) {
            break;
        }
    }

    ttotal += wtime();

    printf("Heat 2D (serial) grid: rows %d, cols %d, niters %d, total time %.6f\n", ny, nx,
niters,
           ttotal);

    // Save grid
    if (argc > 2) {
        const char* filename = argv[2];

        FILE* fout = fopen(filename, "w");
        if (!fout) {
            perror("Can't open file");
            exit(EXIT_FAILURE);
        }

        for (int i = 0; i < ny; i++) {
            for (int j = 0; j < nx; j++) {
                fprintf(fout, "%.4f ", local_grid[IND(i, j)]);
            }
            fprintf(fout, "\n");
        }

        fclose(fout);
    }

    return 0;
}
```

2 Исходный код MPI-программы

```
#include <mpi.h>

#include <cmath>
#include <iostream>

#define EPS 0.001
#define PI 3.14159265358979323846
#define NELEMS(x) (sizeof((x)) / sizeof((x)[0]))
#define IND(i, j) ((i) * (nx + 2) + (j))

int get_block_size(int n, int rank, int nprocs) {
  int s = n / nprocs;
  if (n % nprocs > rank) s++;
  return s;
}

int get_sum_of_prev_blocks(int n, int rank, int nprocs) {
  int rem = n % nprocs;
  return n / nprocs * rank + ((rank >= rem) ? rem : rank);
}

int main(int argc, char **argv) {
  int commsize, rank;
  MPI_Init(&argc, &argv);
  double ttotal = -MPI_Wtime();
  MPI_Comm_size(MPI_COMM_WORLD, &commsize);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  // Создаем 2D-сетку процессов: commsize = px * py
  MPI_Comm cartcomm;
  int dims[2] = {0, 0}, periodic[2] = {0, 0};
  MPI_Dims_create(commsize, 2, dims);
  int px = dims[0];
  int py = dims[1];
  if (px < 2 || py < 2) {
    std::cerr << "Invalid number of processes " << commsize << ": px " << px
              << " and py " << py << " must be greater than 1\n";
    MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
  }
  MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periodic, 0, &cartcomm);
  int coords[2];
  MPI_Cart_coords(cartcomm, rank, 2, coords);
  int rankx = coords[0];
  int ranky = coords[1];
  int rows, cols;

  // Broadcast command line arguments
  if (rank == 0) {
    rows = (argc > 1) ? atoi(argv[1]) : py * 100;
    cols = (argc > 2) ? atoi(argv[2]) : px * 100;
    if (rows < py) {
      std::cerr << "Number of rows " << rows
                << " less then number of py processes " << py << '\n';
      MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
    }
```

```
  if (cols < px) {
    std::cerr << "Number of cols " << cols
              << " less then number of px processes " << px << '\n';
    MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
  }
  int args[2] = {rows, cols};
  MPI_Bcast(&args, NELEMS(args), MPI_INT, 0, MPI_COMM_WORLD);
} else {
  int args[2];
  MPI_Bcast(&args, NELEMS(args), MPI_INT, 0, MPI_COMM_WORLD);
  rows = args[0];
  cols = args[1];
}

// Выделить память для локальных 2D-подсеток с ячейками гало [0..ny + 1][0..nx + 1]
int ny = get_block_size(rows, ranky, py);
int nx = get_block_size(cols, rankx, px);
double *local_grid =
    (double *)calloc((ny + 2) * (nx + 2), sizeof(*local_grid));
double *local_newgrid =
    (double *)calloc((ny + 2) * (nx + 2), sizeof(*local_newgrid));
if (!local_grid || !local_newgrid) {
  std::cerr << "Error while callocing local / new grid\n";
  return 1;
}

// Заполняем граничные точки:
// - левая и правая границы заполнены нулями
// - верхняя граница: u(x, 0) = sin(pi * x)
// - нижняя граница: u(x, 1) = sin(pi * x) * exp(-pi)
double dx = 1.0 / (cols - 1.0);
int sj = get_sum_of_prev_blocks(cols, rankx, px);
if (ranky == 0) {
  // инициализируем верхнюю границу: u(x, 0) = sin(pi * x)
  for (int j = 1; j <= nx; j++) {
    //Перевести индекс col в координату x в [0, 1]
    double x = dx * (sj + j - 1);
    int ind = IND(0, j);
    local_newgrid[ind] = local_grid[ind] = sin(PI * x);
  }
}
if (ranky == py - 1) {
  // инициализируем нижнюю границу : u(x, 1) = sin(pi * x) * exp(-pi)
  for (int j = 1; j <= nx; j++) {
    // Перевести индекс col в координату x в [0, 1]
    double x = dx * (sj + j - 1);
    int ind = IND(ny + 1, j);
    local_newgrid[ind] = local_grid[ind] = sin(PI * x) * exp(-PI);
  }
}

// Соседи
int left, right, top, bottom;
MPI_Cart_shift(cartcomm, 0, 1, &left, &right);
MPI_Cart_shift(cartcomm, 1, 1, &top, &bottom);

// Тип левой и правой границ
```

```
MPI_Datatype col;
MPI_Type_vector(ny, 1, nx + 2, MPI_DOUBLE, &col);
MPI_Type_commit(&col);

// Тип верхней и нижней границ
MPI_Datatype row;
MPI_Type_contiguous(nx, MPI_DOUBLE, &row);
MPI_Type_commit(&row);
MPI_Request reqs[8];
double thalo = 0;
double treduce = 0;

int niters = 0;
for (;;) {
  niters++;
  // обновляем внутренние точки
  for (int i = 1; i <= ny; i++) {
    for (int j = 1; j <= nx; j++) {
      local_newgrid[IND(i, j)] =
          (local_grid[IND(i - 1, j)] + local_grid[IND(i + 1, j)] +
           local_grid[IND(i, j - 1)] + local_grid[IND(i, j + 1)]) *
          0.25;
    }
  }

  //Проверка условии завершения
  double maxdiff = 0;
  for (int i = 1; i <= ny; i++) {
    for (int j = 1; j <= nx; j++) {
      int ind = IND(i, j);
      maxdiff = fmax(maxdiff, fabs(local_grid[ind] - local_newgrid[ind]));
    }
  }

  // Поменять сетки (после завершения local_grid будет содержать результат)
  double *p = local_grid;
  local_grid = local_newgrid;
  local_newgrid = p;
  treduce -= MPI_Wtime();
  MPI_Allreduce(MPI_IN_PLACE, &maxdiff, 1, MPI_DOUBLE, MPI_MAX,
                MPI_COMM_WORLD);
  treduce += MPI_Wtime();
  if (maxdiff < EPS) break;

  // Обмен ореолом: T = 4 * (a + b * (rows/py)) + 4 * (a + b * (cols/px))
  thalo -= MPI_Wtime();
  MPI_Irecv(&local_grid[IND(0, 1)], 1, row, top, 0, cartcomm,
            &reqs[0]);  // top
  MPI_Irecv(&local_grid[IND(ny + 1, 1)], 1, row, bottom, 0, cartcomm,
            &reqs[1]);  // bottom
  MPI_Irecv(&local_grid[IND(1, 0)], 1, col, left, 0, cartcomm,
            &reqs[2]);  // left
  MPI_Irecv(&local_grid[IND(1, nx + 1)], 1, col, right, 0, cartcomm,
            &reqs[3]);  // right
  MPI_Isend(&local_grid[IND(1, 1)], 1, row, top, 0, cartcomm,
            &reqs[4]);  // top
  MPI_Isend(&local_grid[IND(ny, 1)], 1, row, bottom, 0, cartcomm,
```

```
            &reqs[5]);  // bottom
    MPI_Isend(&local_grid[IND(1, 1)], 1, col, left, 0, cartcomm,
            &reqs[6]);  // left
    MPI_Isend(&local_grid[IND(1, nx)], 1, col, right, 0, cartcomm,
            &reqs[7]);  // right
    MPI_Waitall(8, reqs, MPI_STATUS_IGNORE);
    thalo += MPI_Wtime();
  }  // iterations

  MPI_Type_free(&row);
  MPI_Type_free(&col);
  free(local_newgrid);
  free(local_grid);
  ttotal += MPI_Wtime();
  if (rank == 0)
    std::cout << "# Heat 2D (mpi): grid: rows " << rows << ", cols " << cols
            << ", procs " << commsize << " (px " << px << ", py " << py
            << ")\n";
  int namelen;
  char procname[MPI_MAX_PROCESSOR_NAME];
  MPI_Get_processor_name(procname, &namelen);
  std::cout << "# P " << rank << " (" << rankx << ", " << ranky << ") on "
          << procname << ": grid ny " << ny << " nx " << nx << ", total "
          << ttotal << ", mpi " << treduce + thalo << " ("
          << (treduce + thalo) / ttotal << ") = allred " << treduce << " ("
          << treduce / (treduce + thalo) << ") + halo " << thalo << " ("
          << thalo / (treduce + thalo) << ")\n";
  double prof[3] = {ttotal, treduce, thalo};

  if (rank == 0) {
    MPI_Reduce(MPI_IN_PLACE, prof, NELEMS(prof), MPI_DOUBLE, MPI_MAX, 0,
               MPI_COMM_WORLD);
    std::cout << "# procs " << commsize << " : grid " << rows << ' ' << cols
            << " : niters " << niters << " : total time " << prof[0]
            << " : mpi time " << prof[1] + prof[2] << " : allred " << prof[1]
            << " : halo " << prof[2] << '\n';
  } else {
    MPI_Reduce(prof, NULL, NELEMS(prof), MPI_DOUBLE, MPI_MAX, 0,
               MPI_COMM_WORLD);
  }

  MPI_Finalize();
  return 0;
}
```

3 Исходный код Python-программы для построения графика

```
#!/usr/bin/env python3

import matplotlib.pyplot as plt
import numpy as np

def draw(filenames, labels, dest_filename):
    plt.rcParams["legend.markerscale"] = 1.0
    plt.rcParams['font.family'] = 'sans-serif'
    plt.rcParams['font.sans-serif'] = ['PT Sans']
    plt.rcParams['font.size'] = '9'
    plt.rcParams["legend.loc"] = "upper left"
    plt.rcParams["legend.fontsize"] = '7'
    cm = 1 / 2.54 # centimeters in inches
    fig = plt.figure(figsize = (10 * cm, 7 * cm))
    ax = fig.add_subplot(111)
    ax.set_title("")
    ax.set(xlabel = "Количество процессов", ylabel = "Ускорение")
    ax.label_outer()
    #ax.xaxis.set_ticks(np.arange(4, 16, 1))
    #ax.yaxis.set_ticks(np.arange(1.8, 9, 1))
    ax.xaxis.set_tick_params(direction='in', which='both')
    ax.yaxis.set_tick_params(direction='in', which='both')
    for (fname, datalabel) in zip(filenames, labels):
        data = np.loadtxt(fname)
        x = data[:, 0]
        y = data[:, 1]
        if datalabel == "N = 1000":
            marker = '-^'
            color = "orange"
        elif datalabel == "N = 10000":
            marker = '-s'
            color = "dodgerblue"
        else:
            marker = '-'
            color = "red"
        ax.plot(x, y, marker, c = color, markersize = 4.0, linewidth = 1.2, label = datalabel)
    labels = [item.get_text() for item in ax.get_xticklabels()]
    labels[1] = '4\n(2x2)'
    labels[2] = '6\n(2x3)'
    labels[3] = '8\n(2x4)'
    labels[4] = '10\n(2x5)'
    labels[5] = '12\n(2x6)'
    labels[6] = '14\n(2x7)'
    labels[7] = '16\n(2x8)'
    ax.set_xticklabels(labels)
    plt.tight_layout()
    ax.legend()
    fig.savefig(dest_filename, dpi = 300)

if __name__ == "__main__":
    draw(["laplace_1000.dat", "laplace_10000.dat", "linear.dat"], ["N = 1000", "N = 10000",
"Линейное ускорение"], "laplace.png")
```