Московский государственный университет имени М.В.Ломоносова
Факультет вычислительной математики и кибернетики

Отчёт по заданию по курсу «Суперкомпьютерное моделирование и технологии»
2025

Вариант «8»

Биктимиров Михаил Геннадьевич
Группа 627

Дата подачи OMP отчёта – 27.10.2025
Дата подачи MPI и MPI+OpenMP отчёта – 24.11.2025
Дата подачи MPI+CUDA отчёта – 06.12.2025

Содержание

1. Математическая постановка задачи

В рамках данного задания решается трехмерная задача для гиперболического уравнения в частных производных в области, представляющей из себя прямоугольный параллелепипед.

Дифференциальная задача

Рассмотрим трехмерную замкнутую область, представляющую собой прямоугольный параллелепипед:

$$\Omega = [0 \le x \le L_x] \times [0 \le y \le L_y] \times [0 \le z \le L_z]$$

В области Ω для времени $(0 < t \le T]$ требуется найти решение u(x, y, z, t) **уравнения в частных производных**:

$$\partial^2 u/\partial t^2 = a^2 \Delta u \tag{1}$$

с начальными условиями:

$$u|_{t=0} = \varphi(x, y, z) \tag{2}$$

$$\partial u/\partial t|_{t=0} = 0 \tag{3}$$

Граничные условия

Периодические **граничные условия**:

$$u(0, y, z, t) = u(Lx, y, z, t),\ u_x(0, y, z, t) = u_x(L_x, y, z, t) \quad (7)$$

$$u(x, 0, z, t) = u(x, L_\gamma, z, t),\ u_\gamma(x, 0, z, t) = u_\gamma(x, L_\gamma, z, t) \quad (8)$$

$$u(x, y, 0, t) = u(x, y, L\_z, t),\ u_z(x, y, 0, t) = u_z(x, y, L_z, t) \quad (9)$$

Аналитическое решение

Для проверки точности численного решения используется **аналитическое решение**, которое задано в соответствии с вариантом и имеет вид:

$$\text{analytical\_solution} = \sin(2\pi/L_x \cdot x) \cdot \sin(4\pi/L_\gamma \cdot y) \cdot \sin(6\pi/L_z \cdot z) \cdot \cos(a_t \cdot t)$$

где $a_t = \pi\sqrt{(4/L_x^2 + 16/L_y^2 + 36/L\_z^2)}$, $a^2 = 1$

Начальная функция φ(x, y, z) определяется как analytical_solution при t = 0:

$$\varphi(x, y, z) = u\_analytical|\_\{t=0\} = \sin(\pi/L_x\, x) \cdot \sin(\pi/L_y\, y) \cdot \sin(\pi/L\_z\, z)$$

После вычисления численного решения необходимо проверить его точность путем сравнения с известным аналитическим решением и вычисления максимальной погрешности на сетке.

2. Численный метод решения

Для численного решения трехмерной гиперболической задачи с периодическими граничными условиями (8-й вариант задания) используется явная разностная схема на равномерной сетке.

Сеточная аппроксимация

Введем на области $\Omega = [0 \le x \le L_x] \times [0 \le y \le L_y] \times [0 \le z \le L\_z]$ равномерную сетку $\omega_h\tau = \bar{\omega_h} \times \omega\tau$, где:

$$\bar{\omega_h} = \{(x_i = ih_x, y_j = jh_y, z_k = kh\_z), i, j, k = 0, 1, ..., N, h_xN = L_x, h_yN = L_y, h\_zN = L\_z\}$$

$$\omega\tau = \{t_n = n\tau, n = 0, 1, ..., STEPS, \tau \cdot STEPS = T\}$$

Через ω_h обозначим множество внутренних узлов сетки, а через γ_h — множество граничных узлов.

Разностная схема

Для аппроксимации исходного уравнения (1) с периодическими граничными условиями (7)-(9) и начальными условиями (2)-(3) используется следующая явная разностная схема:

$$(u^{n+1}_{ijk} - 2u^n_{ijk} + u^{n-1}_{ijk})/\tau^2 = a^2\Delta_h u^n, (x_i, y_j, z_k) \in \omega_h, n = 1, 2, ..., STEPS-1$$

где $a^2 = 1$, а Δ_h — семиточечный разностный аналог оператора Лапласа:

$$\Delta_h u^n = (u^n_{i-1,j,k} - 2u^n_{i,j,k} + u^n_{i+1,j,k})/h_x^2 + (u^n_{i,j-1,k} - 2u^n_{i,j,k} + u^n_{i,j+1,k})/h_y^2 + (u^n_{i,j,k-1} - 2u^n_{i,j,k} + u^n_{i,j,k+1})/h\_z^2$$

Начальные условия на сетке

Для 8-го варианта задания начальные условия (**нулевой временной слой**) на сетке определяются следующим образом:

$$u^0_{ijk} = \varphi(x_i, y_j, z_k) = u|t=0 = analytical\_solution(x_i, y_j, z_k) \qquad (10)$$

Для обеспечения второго порядка аппроксимации по τ и численного определения **первого временного слоя** используется формула:

$$u^1_{ijk} = u^0_{ijk} + (a^2\tau^2/2) \cdot \Delta_h\varphi(x_i, y_j, z_k) = \{a^2=1, \varphi=u|t=0=u0\} = u^0_{ijk} + (\tau^2/2) \cdot \Delta_h u0 \qquad (11)$$

Для численного определения **следующих временных слоёв** используется формула:

$$(u^{n+1}_{ijk} - 2u^n_{ijk} + u^{n-1}_{ijk})/\tau^2 = a^2\Delta_h u^n, (x_i, y_j, z_k) \in \omega_h, n = 1, 2, ..., STEPS-1$$

откуда с с учётом $a^2=1$ получаем

$$u^{n+1}_{ijk} = \tau^2 \Delta_h u^n + 2u^n_{ijk} - u^{n-1}_{ijk},\ (x_i, y_j, z_k) \in \omega_h,\ n = 1, 2, ..., STEPS-1,\ i, j, k = 0, 1, ..., N \quad (12)$$

Таким образом, получаем следующий алгоритм вычислений

1. Инициализация начальных условий:
2. Вычисление u^0_{ijk} по формуле (15)
3. Применение периодических граничных условий к u^0
4. Вычисление u^1_{ijk} по формуле (16)
5. Применение периодических граничных условий к u^1
6. Для каждого временного шага n = 2, 3, ..., STEPS-1:
7. Установка периодических граничных условий для слоя n
8. Вычисление следующего слоя по формуле:

$$u^{n+1}_{ijk} = 2u^n_{ijk} - u^{n-1}_{ijk} + \tau^2 \cdot \Delta_h u^n$$

3. Программная реализация численного метода с использованием OpenMP

Программная реализация численного метода решения трехмерной гиперболической задачи с периодическими граничными условиями (8-й вариант задания) выполнена на языке C++ с использованием библиотеки **OpenMP** для параллельных вычислений. Ниже приведено описание основных функций, реализующих численный метод.

Вычисление начального условия u^0 (нулевой временной слой)

```
void compute_u0(const Grid& g, std::vector<double>& u_layer, double t)
```

Назначение: функция вычисляет начальное условие u^0_{ijk} согласно формуле (10) из математической постановки задачи.

Аргументы:

- g — объект класса Grid, содержащий параметры сетки (размеры, шаги)
- u_layer — вектор для хранения вычисленных значений u^0 на текущем временном слое
- t — текущее время (для начального условия t = 0)

Использование: вызывается один раз в начале расчета для инициализации начального состояния системы.

Вычисление первого временного слоя u^1

```
void compute_u1(const Grid& g, std::vector<double>& u1,
                const std::vector<double>& u0)
```

Назначение: функция вычисляет первый временной слой u^1_{ijk} согласно формуле (11) из математической постановки задачи.

Аргументы:

- g — объект класса Grid
- u1 — вектор для хранения вычисленных значений u^1
- u0 — вектор, содержащий значения u^0

Использование: вызывается один раз после вычисления u^0 для подготовки первого временного шага.

Вычисление следующего временного слоя u^{n+1}

```
void compute_u_next(const Grid& g, std::vector<double>& u_next,
     const std::vector<double>& u_curr, const std::vector<double>& u_prev)
```

Назначение: функция вычисляет следующий временной слой u^{n+1}_{ijk} согласно основной разностной схеме (12).

Аргументы:
- g — объект класса Grid
- u_next — вектор для хранения вычисленных значений u^{n+1}
- u_curr — вектор, содержащий значения u^n
- u_prev — вектор, содержащий значения u^{n-1}

Использование: вызывается на каждом временном шаге (начиная со второго) для продвижения решения во времени.

Установка периодических граничных условий

```
void set_boundary_conditions(const Grid& g, std::vector<double>& u_layer,
double t)
```

Назначение: функция устанавливает периодические граничные условия (7)-(9) на текущем временном слое.

Аргументы:
- g — объект класса Grid
- u_layer — вектор, содержащий значения решения на текущем временном слое
- t — текущее время

Использование: вызывается после расчета каждого временного слоя

Вычисление погрешности

```
double compute_error(const Grid& g, const std::vector<double>& numerical,
double t)
```
Назначение: функция вычисляет максимальную погрешность между численным и аналитическим решениями.

Аргументы:

- g — объект класса Grid
- numerical — вектор, содержащий численное решение
- t — текущее время

Использование: вызывается на каждом временном шаге для оценки точности численного решения и вывода информации о погрешности.

Основная функция решения задачи

```
void solver_execute(Grid& g, double& inaccuracy_1st,
      double& inaccuracy_last, double& inaccuracy_max,
      std::vector<double>& result, double& time, int& threads_num)
```

Назначение: функция реализует основной алгоритм решения задачи, объединяя все вышеописанные компоненты.

Аргументы:
- g — объект класса Grid
- inaccuracy_1st — переменная для хранения погрешности на первом шаге
- inaccuracy_last — переменная для хранения погрешности на последнем шаге
- inaccuracy_max — переменная для хранения максимальной погрешности
- result — вектор для хранения результата (решение на последнем временном слое)
- time — переменная для хранения времени выполнения
- threads_num — количество потоков OpenMP

Реализация: функция последовательно:

- Инициализирует начальные условия u^0 и u^1
- Устанавливает граничные условия
- Запускает цикл по временным шагам, на каждом шаге:
- Устанавливает граничные условия
- Вычисляет следующий временной слой
- Вычисляет и анализирует погрешность
- Сохраняет результат и измеряет время выполнения
- Особенности реализации:

Используется циклическое хранение трех временных слоев (u^0, u^1, u^2) для минимизации использования памяти. Все вычисления распараллеливаются с использованием OpenMP. Для каждого временного шага вычисляется и выводится погрешность. В конце выполнения формируется сводная информация о точности и производительности.

4. Программная реализация численного метода с использованием MPI (и MPI+OpenMP)

Каждый процесс выполняет описанный выше алгоритм численного решения уравнения в своей области, при этом пересылки сообщений между процессами выполняются средствами MPI в следующих случаях:

- обмен граничными областями между соседними процессами. Данный тип обмена необходим для вычисления оператора Лапласа внутри всех узлов решетки, принадлежащей процессу.
- обмен граничными областями для поддержания выполнения периодических условий.
- вычисление метрик для оценки качества работы программы. Директивы OpenMP использовались в программе для распараллеливания циклов, так как значения на различных итерациях внутри циклов вычисляются независимо

Разбиение

При этом использовалось **блочное разбиение**, т.к. в этом случае предполагается меньшее число межпроцессных коммуникаций, чем в случае ленточного разбиения. Предполагается, что наилучшим разбиением является наиболее равномерное разбиение: число узлов сетки, принадлежащих процессу, для почти всех процессов одинаково. Количество процессов по осям определяется функцией `balance_process_grid`, которая:

- раскладывает общее число процессов на простые множители
- распределяет множители по измерениям для максимальной сбалансированности (например, для 8 процессов: 2x2x2 или 4x2x1)

Таким образом, разбиение происходит следующим образом:

1. Определение структуры разбиения:
   - Общее число процессов `num_procs` раскладывается на множители с помощью функции `balance_process_grid`
   - Эти множители распределяются по трем осям (x, y, z) так, чтобы получить максимально равномерное разбиение
2. Создание декартовой топологии:
   - `MPI_Cart_create(MPI_COMM_WORLD, ndim, dims, periods, 0, &comm_cart);`
   - Создается трехмерная декартова решетка процессов
   - Все направления помечены как периодические (`periods[d] = 1`)
3. Определение локальной области для каждого процесса:

```
const int i_min = coords[0] * nodes[0], i_max = std::min(N,
(coords[0] + 1) * nodes[0]) - 1;
const int j_min = coords[1] * nodes[1], j_max = std::min(N,
(coords[1] + 1) * nodes[1]) - 1;
```

```
const int k_min = coords[2] * nodes[2], k_max = std::min(N,
(coords[2] + 1) * nodes[2]) - 1;
```

Каждый процесс получает блок размером примерно N/dims[0] x N/dims[1] x N/dims[2]. Для крайних процессов размеры могут отличаться, если N не делится нацело на dims[d]

4. Добавление обменных областей:
   ```
   dx = i_max - i_min + 1 + 2;
   dy = j_max - j_min + 1 + 2;
   dz = k_max - k_min + 1 + 2;
   ```
   Каждый процесс выделяет память для своей локальной области плюс по 2 элемента с каждой стороны. Итого размер локальной области с учетом обменных областей: (локальный размер по x + 2) x (локальный размер по y + 2) x (локальный размер по z + 2)

Такой подход к разбиению соответствует блочному разбиению, указанному в задании, и обеспечивает меньшее количество коммуникаций по сравнению с ленточным разбиением.

Коммуникация

Коммуникация между процессами реализована через шесть функций для каждой из осей (x, y, z):

1. Прямой обмен (forward):
   - `send_recv_forward_x/y/z` - передача данных от процесса к процессу в "прямом" направлении
   - Каждый процесс (кроме первого) отправляет свой правый край (значение с индексом `dx-2`) процессу слева
   - Каждый процесс (кроме последнего) получает данные от процесса справа и помещает их в свою левую обменную область (индекс 0)
2. Обратный обмен (backward):
   - `send_recv_backward_x/y/z` - передача данных от процесса к процессу в "обратном" направлении
   - Каждый процесс (кроме первого) получает данные от процесса слева и помещает их в свою правую обменную область (индекс `dx-1`)
   - Каждый процесс (кроме последнего) отправляет свой левый край (значение с индексом 1 или 2) процессу справа

В процессе коммуникации используется следующий алгоритм:

1. Подготовка данных в буфер отправки (копирование из рабочей области в буфер)
2. Вызов `MPI_Sendrecv` для одновременной отправки и приема данных
3. Копирование принятых данных в обменную область

После коммуникации вызывается функция `set_boundary_conditions`, которая устанавливает периодические граничные условия, используя данные из обменных областей.

Резюме обновленных и новых функций

Балансировка процессов по осям

```
void balance_process_grid(int* dims, int num_dims, int num_procs)
```

Назначение: функция раскладывает общее число процессов на множители и распределяет их по осям для создания декартовой топологии.

Аргументы:
- dims – массив для хранения числа процессов по каждой оси
- ndim – размерность задачи (всегда 3 для данной задачи)
- num_procs – общее число процессов

Использование: вызывается при инициализации MPI для определения оптимального разбиения области между процессами.

Прямой обмен данными по оси X

```
void send_recv_forward_x(double *data, MPI_Comm& comm_cart, int rank_prev,
int rank_next, bool is_first, bool is_last)
```

Назначение: функция выполняет обмен данными в "прямом" направлении по оси X (каждый процесс отправляет данные процессу слева).

Аргументы:

- data – массив данных текущего временного слоя
- comm_cart – коммуникатор декартовой топологии
- rank_prev – ранг предыдущего процесса
- rank_next – ранг следующего процесса
- is_first – флаг, является ли процесс первым по оси X

- is_last – флаг, является ли процесс последним по оси X

Использование: вызывается на каждом временном шаге для передачи правых краевых значений процессам слева.

Обратный обмен данными по оси X

```
void send_recv_backward_x(double *data, MPI_Comm& comm_cart, int
rank_prev, int rank_next, bool is_first, bool is_last)
```

Назначение: функция выполняет обмен данными в "обратном" направлении по оси X (каждый процесс отправляет данные процессу справа).

Аргументы: аналогично `send_recv_forward_x`

Использование: вызывается на каждом временном шаге для передачи левых краевых значений процессам справа.

Прямой и обратный обмен данными по осям Y и Z

```
void send_recv_forward_y/z(...),
void send_recv_backward_y/z(...)
```

Назначение: аналогично функциям для оси X, но для осей Y и Z соответственно.

Аргументы: аналогичны функциям для оси X

Использование: вызываются последовательно с функциями для оси X на каждом временном шаге для полной синхронизации данных между всеми соседними процессами.

Вычисление линейного индекса

```
inline int index(int i, int j, int k)
```

Назначение: функция преобразует трехмерные индексы в линейный индекс для доступа к одномерному массиву.

Аргументы:
- i, j, k – индексы по осям x, y, z соответственно

Использование: используется повсеместно для доступа к элементам трехмерного массива, хранящегося в одномерном буфере.

Вычисление второй производной

```
inline double calculate_second_derivative(const double data[], int i, int j, int k, int di, int dj, int dk, double h)
```

Назначение: функция вычисляет вторую производную по заданному направлению (x, y или z) с использованием центральной разностной схемы.

Аргументы:
- data – массив данных текущего временного слоя
- i, j, k – индексы текущей точки в локальной области
- di, dj, dk – направление (1 для оси x, 0 для остальных и т.д.)
- h – шаг сетки по соответствующей оси

Использование: вызывается внутри функции laplace для вычисления оператора Лапласа.

Вычисление оператора Лапласа

```
inline double laplace(const double data[], int i, int j, int k)
```

Назначение: функция вычисляет оператор Лапласа в точке (i, j, k) как сумму вторых производных по всем трем осям.

Аргументы:
- data – массив данных текущего временного слоя
- i, j, k – индексы текущей точки в локальной области

Использование: вызывается при вычислении следующего временного слоя согласно разностной схеме.

Флаги is_first и is_last

Переменные `is_first` и `is_last` — это массивы логических значений, которые хранят информацию о положении текущего вычислительного блока (процесса MPI) относительно общих границ всей глобальной 3D-сетки.
Они критически важны для реализации параллельных вычислений и обмена данными (MPI). Иными словами, `is_first` и `is_last` – это навигационные флаги, которые определяют "местоположение" текущего вычислительного процесса в глобальной задаче и помогают координировать параллельные вычисления и коммуникации.

Эти массивы содержат:

1) **`is_first`** (начальная граница)

Массив хранит `true`, если текущий блок примыкает к минимальной (начальной) физической границе всего расчетного домена по данной оси:

- `is_first[0]` (ось X): `true`, если это самый левый блок в глобальной сетке.
- `is_first[1]` (ось Y): `true`, если это самый нижний блок.
- `is_first[2]` (ось Z): `true`, если это самый задний блок.

2) **`is_last`** (Конечная граница)

Массив хранит `true`, если текущий блок примыкает к максимальной (конечной) физической границе всего расчетного домена по данной оси.

- `is_last[0]` (ось X): `true`, если это самый правый блок в глобальной сетке.
- `is_last[1]` (ось Y): `true`, если это самый верхний блок.
- `is_last[2]` (ось Z): `true`, если это самый передний блок.

Эти флаги используются в нескольких ключевых местах:

1) *Управление обменом данными (MPI)*
    a) В функциях типа send_recv_forward_x, эти флаги определяют, нужно ли выполнять коммуникацию.
        i) Если блок находится на физической границе (is_first или is_last равно true), ему не нужно получать данные от "несуществующего" соседа за границей домена.
        ii) Если же блок не является ни первым, ни последним (!is_first && !is_last), он является внутренним и обязан обмениваться данными с обоими соседями (слева и справа).
2) *Пропуск вычислений в основном вычислительном цикле* (в шаге солвера):
    В основном вычислительном цикле данные переменные используются для того, чтобы потоки, работающие на внутренних границах блока, не пытались получить доступ к данным, которые находятся за пределами текущего блока и еще не были синхронизированы:

5. Программная реализация численного метода с использованием MPI+CUDA

Идея реализации

В MPI+CUDA версии произошли следующие ключевые изменения по сравнению с MPI+OpenMP

– **перенос вычислений на GPU**
Большинство циклов теперь реализованы в виде CUDA-ядер (kernels), поэтому основные вычисления (вычисление временных слоев, оценка ошибки, обработка граничных условий) перенесены на GPU

– **использование GPU-памяти**
Все данные хранятся на GPU и обрабатываются там же (т.е. вместо выделения памяти на CPU с `new double[...]` используется `cudaMalloc`), а для удобства и экономии введена структура `solver_params` для передачи параметров в ядра

– **изменения в подсчёте ошибки**
Оценка ошибки теперь происходит на GPU с использованием библиотеки `thrust`, и теперь вычисление среднеквадратичной и макс-ошибок разделено на два этапа: вычисление частичных результатов в ядрах и агрегация через `Thrust`

В коде используется библиотека `thrust` при подсчёте норм и нет прямого использования разделяемой памяти (`shared memory`, `__shared__` в кернелах), при этом при подсчёте также используется дополнительный буфер на GPU, поскольку `thrust` больше заточен под работу с непрерывными массивами, а в решаемой мной задаче данные с обменными областями смешаны, и это оказалось самым эффективным решением для данной ситуации (p.s. хотя я также пытался в данной ситуации использовать траст для вычисления только в рабочей области, без обменной, и пробовать использовать итераторы, но это не дало хороших результатов), т.е. кернелы `compute_mse_error` и `compute_max_error` фильтруют данные, вычисляя ошибки только во внутренних точках (исключая обменные области) и записывают их во временный буфер `u_error`, а после библиотека `thrust` (`thrust::reduce`) выполняет редукцию по временному буферу

Также в MPI+CUDA версии граничные условия обрабатываются иначе, чем в MPI+OpenMP, используются специальные кернелы: `update_halo_x, update_halo_y, update_halo_z` – отдельные ядра для обновления граничных условий по каждой оси, которые работают только с граничными слоями, что эффективнее, чем обработка всей области. При этом для передачи данных между процессами используется двухэтапный подход: сначала данные копируются в промежуточный буфер с помощью `pack_slice_to_halo_buffer_*`, а затем данные извлекаются из буфера с помощью

`unpack_slice_from_halo_buffer_*`, что позволяет минимизировать количество копирований между GPU и хостом.

Резюме обновлённых и новых методов

Вычисление временного слоя на GPU (шаг солвера)

```
__global__ void solver_step(double *p_next, double *p_curr, double
*p_prev, int n, solver_params params)
```

Назначение: функция-ядро вычисляет следующий временной слой u^{n+1} на GPU согласно основной разностной схеме.

Аргументы:

- p_next – указатель на массив для хранения вычисленных значений u^{n+1}
- p_curr – указатель на массив, содержащий значения u^{n}
- p_prev – указатель на массив, содержащий значения u^{n-1}
- n – номер текущего временного слоя
- params – структура с параметрами задачи

Использование: вызывается на каждом временном шаге для продвижения решения во времени. Для n=0 вычисляет начальные условия, для n=1 – первый временной слой, для n>1 – последующие временные слои.

Вычисление среднеквадратичной ошибки

```
__global__ void compute_mse_error(double *err, const double *data,
solver_params params, int n)
```

Назначение: функция-ядро вычисляет квадраты разностей между численным и аналитическим решениями для MSE.

Аргументы:

- err – указатель на массив для хранения результатов
- data – указатель на массив численного решения
- params – структура с параметрами задачи
- n - номер текущего временного слоя

Использование: вызывается на каждом временном шаге для подготовки данных к вычислению MSE через thrust.

Вычисление максимальной ошибки

```
__global__ void compute_max_error(double *err, const double *data,
solver_params params, int n)
```

Назначение: функция-ядро вычисляет абсолютные разности между численным и аналитическим решениями для определения максимальной ошибки.

Аргументы:
- err – указатель на массив для хранения результатов
- data – указатель на массив численного решения
- params – структура с параметрами задачи
- n – номер текущего временного слоя

Использование: вызывается на каждом временном шаге для подготовки данных к вычислению максимальной ошибки через thrust.

Обновление граничных условий по оси X

```
__global__ void update_halo_x(double *data, solver_params params, int
i_dst, int i_src)
```

Назначение: функция-ядро обновляет граничные значения по оси X для периодических условий.

Аргументы:
- data – указатель на массив данных текущего временного слоя
- params – структура с параметрами задачи
- i_dst – индекс назначения
- i_src – индекс источника

Использование: вызывается после коммуникации для установки значений на границах локальной области согласно периодическим условиям по оси X.

Обновление граничных условий по осям Y и Z

```
__global__ void update_halo_y/z(double *data, solver_params params, int
j_dst/k_dst, int j_src/k_src)
```

Назначение: аналогично `update_halo_x`, но для осей Y и Z соответственно.

Аргументы: аналогичны `update_halo_x`

Использование: вызывается последовательно с `update_halo_x` после коммуникации для полной установки периодических граничных условий.

Упаковка данных для обмена по оси X

```
__global__ void pack_slice_to_halo_buffer_x(double *buffer, double *data,
solver_params params, int i)
```

Назначение: функция-ядро копирует данные из указанного слоя по оси X в буфер для передачи другим процессам.

Аргументы:
- buffer – указатель на буфер для передачи
- data – указатель на массив данных
- params – структура с параметрами задачи
- i – индекс слоя по оси X

Использование: вызывается перед отправкой данных соседним процессам по оси X.

Распаковка данных после обмена по оси X

```
__global__ void unpack_slice_from_halo_buffer_x(double *buffer, double
*data, solver_params params, int i)
```

Назначение: функция-ядро копирует принятые данные из буфера в указанный слой по оси X.

Аргументы:
- buffer – указатель на буфер с принятыми данными
- data – указатель на массив данных
- params – структура с параметрами задачи
- i – индекс слоя по оси X

Использование: вызывается после приема данных от соседних процессов по оси X.

Упаковка и распаковка данных по осям Y и Z

```
__global__ void
pack_slice_to_halo_buffer_y/z/unpack_slice_to_halo_buffer_y/z(...)
```

Назначение: аналогично функциям для оси X, но для осей Y и Z соответственно.

Аргументы: аналогичны функциям для оси X

Использование: вызываются последовательно с функциями для оси X при обмене данными между процессами по всем трем осям.

Обмен данными по оси X

```
void send_recv_forward_x/backward_x(double *data, MPI_Comm& comm_cart, int rank_prev, int rank_next, bool is_first, bool is_last)
```

Назначение: функция выполняет обмен данными по оси X между процессами с использованием GPU-памяти.

Аргументы:

- data – указатель на массив данных на GPU
- comm_cart – коммуникатор декартовой топологии
- rank_prev – ранг предыдущего процесса
- rank_next – ранг следующего процесса
- is_first – флаг, является ли процесс первым по оси X
- is_last – флаг, является ли процесс последним по оси X

Использование: вызывается на каждом временном шаге для передачи данных между процессами по оси X с использованием GPU-буферов.

Оценка корректности параллельного решения

В рамках выполнения задания по суперкомпьютерному моделированию для оценки корректности гибридной реализации MPI+GPU (и других параллельных версий) была реализована комплексная система сравнения численного решения с аналитическим, основанная на **вычислении стандартных метрик погрешности**. Этот подход позволяет объективно оценить точность и надежность параллельного алгоритма, включая корректность межпроцессорного взаимодействия и обработки граничных условий.

Основой проверки является **аналитическое решение**, соответствующее индивидуальному варианту задания (*вариант 8*), которое задается формулой:

$$u_{analytical} = \sin(2\pi x/L_x) * \sin(4\pi y/L_y) * \sin(6\pi z/L_z) * \cos(a_t * t),$$

где $a_t = \pi*\sqrt{4/L_x^2+16/L_y^2+36/L_z^2}$, а $a^2 = 1$

Это решение используется как эталон для сравнения с результатами численного расчета на каждом временном шаге.

Для количественной оценки отклонения численного решения от аналитического применяются две ключевые метрики:

1) **Среднеквадратичная ошибка (MSE)** вычисляется как среднее значение квадратов разностей между соответствующими узлами сетки. Эта величина дает представление о среднем уровне погрешности по всей области. Для получения значения в физических единицах из MSE извлекается квадратный корень, что дает корень среднеквадратичной ошибки (RMSE).
2) Параллельно с этим вычисляется **максимальная абсолютная ошибка (MAX**), которая представляет собой наибольшее по модулю отклонение в любом узле сетки и характеризует худший случай погрешности. Эти метрики рассчитываются на каждом временном шаге, что позволяет отслеживать эволюцию ошибки во времени.

Особое внимание уделено корректной реализации этой проверки в условиях гибридной архитектуры. Вычисления производятся непосредственно на GPU: ядра `compute_mse_error` и `compute_max_error` параллельно рассчитывают частичные значения ошибок для всех узлов локальной подобласти каждого процесса. Для эффективной агрегации этих частичных результатов внутри GPU используется библиотека `Thrust`, которая обеспечивает оптимальную редукцию данных с использованием всех доступных ресурсов графического процессора. Затем, с помощью коллективных операций `MPI_Reduce`, полученные на каждом процессе частичные суммы объединяются в глобальные значения RMSE и MAX, которые становятся доступны на корневом процессе. Такой подход гарантирует, что конечные значения ошибок являются полными и корректными для всей расчетной области, несмотря на ее распределение между множеством процессов и GPU.

Корректность всей системы оценки была подтверждена несколькими факторами. Во-первых, при запуске с одним MPI-процессом и минимальным размером сетки

результаты должны совпадать (и совпадает) с последовательной реализацией. Во-вторых, для варианта с периодическими граничными условиями (П П П) наблюдаемые значения ошибок должны оставаться стабильными и находиться в пределах, допустимых для выбранной разностной схемы второго порядка аппроксимации по времени и пространству (был выбран допустимый диапазон порядка 10^{-11}-10^{-5}). Небольшие значения RMSE и MAX, сохраняющиеся на протяжении всех 20 временных шагов, свидетельствуют о том, что численная схема устойчива, межпроцессорные коммуникации выполняются корректно, и граничные условия обрабатываются без ошибок. Таким образом, представленная система оценки использовалась для контроля точности решения и позволила доказать корректность параллельной реализации

Используемая в решении явная схема для волнового уравнения, как было сказано выше, имеет вид:

$$(u^{n+1} - 2u^n + u^{n-1})/\tau^2 = a^2\Delta u$$

Откуда видим, что схема имеет как второй порядок по времени ($O(\tau^2)$), так и второй порядок по пространству ($O(h^2)$), а это значит, что при уменьшении шага, например, в 2 раза ошибка должна уменьшиться в 4 раза (таблица ниже, полученная при запусках MPI+CUDA путём усреднения, подтверждает данный факт)

Таблица 0. Скорость убывания ошибки (20 шагов,N=512,CUDA+MPI)

N	RMSE	MAX	Скорость убывания RMSE	Скорость убывания MAX
128	2,42E-10	1,40E-09	1,00	1,00
256	6,17E-11	3,53E-10	3,92	3,97
512	1,55E-11	8,96E-11	15,61	15,63
1024	3,80E-12	2,21E-11	63,68	63,35

Таблица со скоростью убывания ошибки подтверждает, что при увеличении числа узлов сетки в N раза, погрешность решения уменьшается в N^2 раз, что соответствует теоретическому второму порядку точности используемой разностной схемы. Это подтверждает корректность реализации как последовательной, так и всех параллельных версий программы (для других версий это было также проверено в процессе выполнения работы и проверки корректности, что можно видеть в таблицах ниже). Также для проверки корректности были сравнены получившиеся ошибки при разных способах распараллеливания. Для наглядности построены графики убывания ошибок.

График 1. Скорость убывания RMSE

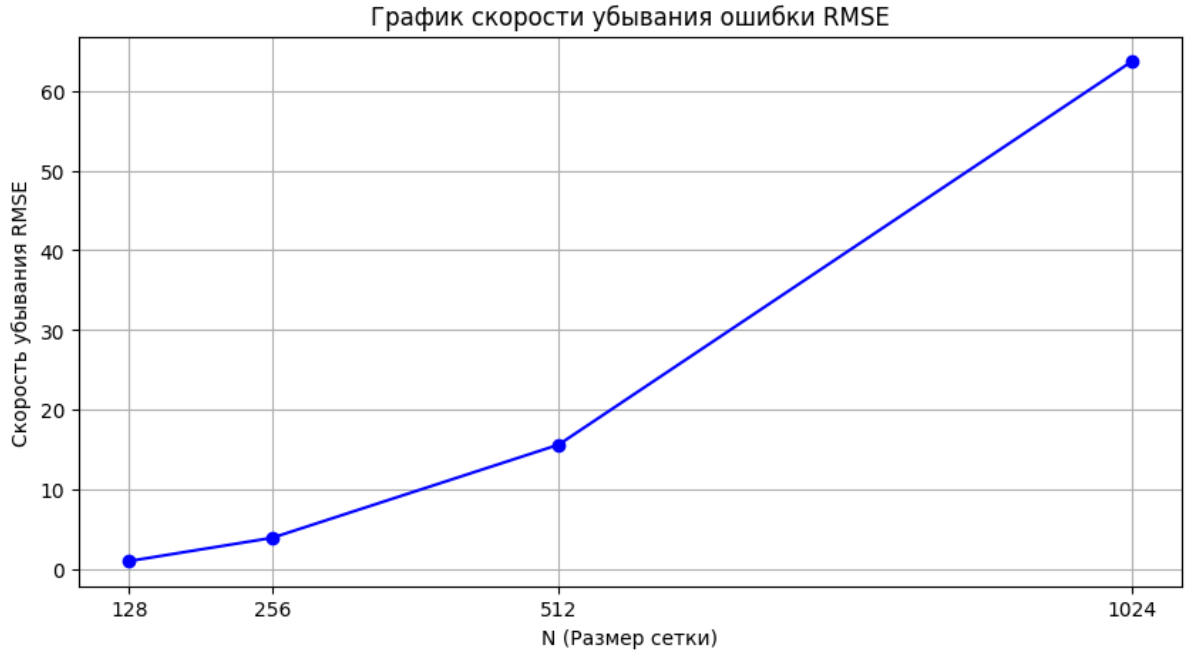


График 2. Скорость убывания MAX

6. Результаты запусков на ПВС Polus

Запуски были проведены в промежутках:

- 23.10.25 – 30.10.25
- 22.11.25 – 10.12.25

В каждый из дней внутри данных промежутков была работа над заданием, осуществлялись очередные запуски с целью усреднения полученных данных и получения более точного результата, испытывались различные способы ускорения, а также замерялись различные конфигурации. Все полученные результаты представлены в виде таблиц и графиков ниже.

Выводы на основании представленных ниже результатов изложены в главе 7.

6.1. OpenMP

Таблица 1. Lx=Ly=Lz=1, N=128

Число OpenMP нитей	N	Время решения T, с	Ускорение S	Ошибка MAX
1	128	11,9746	1	3,31E-09
2	128	6,21604	1,926403305	3,31E-09
4	128	3,1788	3,767019001	3,31E-09
8	128	1,71784	6,970730685	3,31E-09
16	128	1,3185	9,081987107	3,31E-09
32	128	1,11423	10,74697325	3,31E-09

Таблица 2. Lx=Ly=Lz=1, N=256

Число OpenMP нитей	N	Время решения T, с	Ускорение S	Ошибка MAX
1	256	86,974	1	8,27E-10
2	256	44,859	1,938830558	8,27E-10
4	256	22,7998	3,814682585	8,27E-10
8	256	12,2386	7,106531793	8,27E-10
16	256	7,08264	12,27988434	8,27E-10
32	256	5,98331	14,53610125	8,27E-10

Таблица 3. Lx=Ly=Lz=1, N=512

Число OpenMP нитей	N	Время решения T, с	Ускорение S	Ошибка MAX
1	512	661,683	1	2,07E-10
2	512	339,754	1,94753557	2,07E-10
4	512	170,71	3,876064671	2,07E-10
8	512	88,1233	7,508604421	2,07E-10
16	512	55,1862	11,99000837	2,07E-10
32	512	32,3481	20,45508082	2,07E-10

Таблица 4. Lx=Ly=Lz=Pi, N=128

Число OpenMP нитей	N	Время решения T, с	Ускорение S	Погрешность δ
1	128	11,9856	1	3,26E-08
2	128	6,22467	1,925499665	3,26E-08
4	128	3,17319	3,777145396	3,26E-08
8	128	1,71712	6,980059635	3,26E-08
16	128	1,34024	8,942875903	3,26E-08
32	128	1,183654	10,12593207	3,26E-08

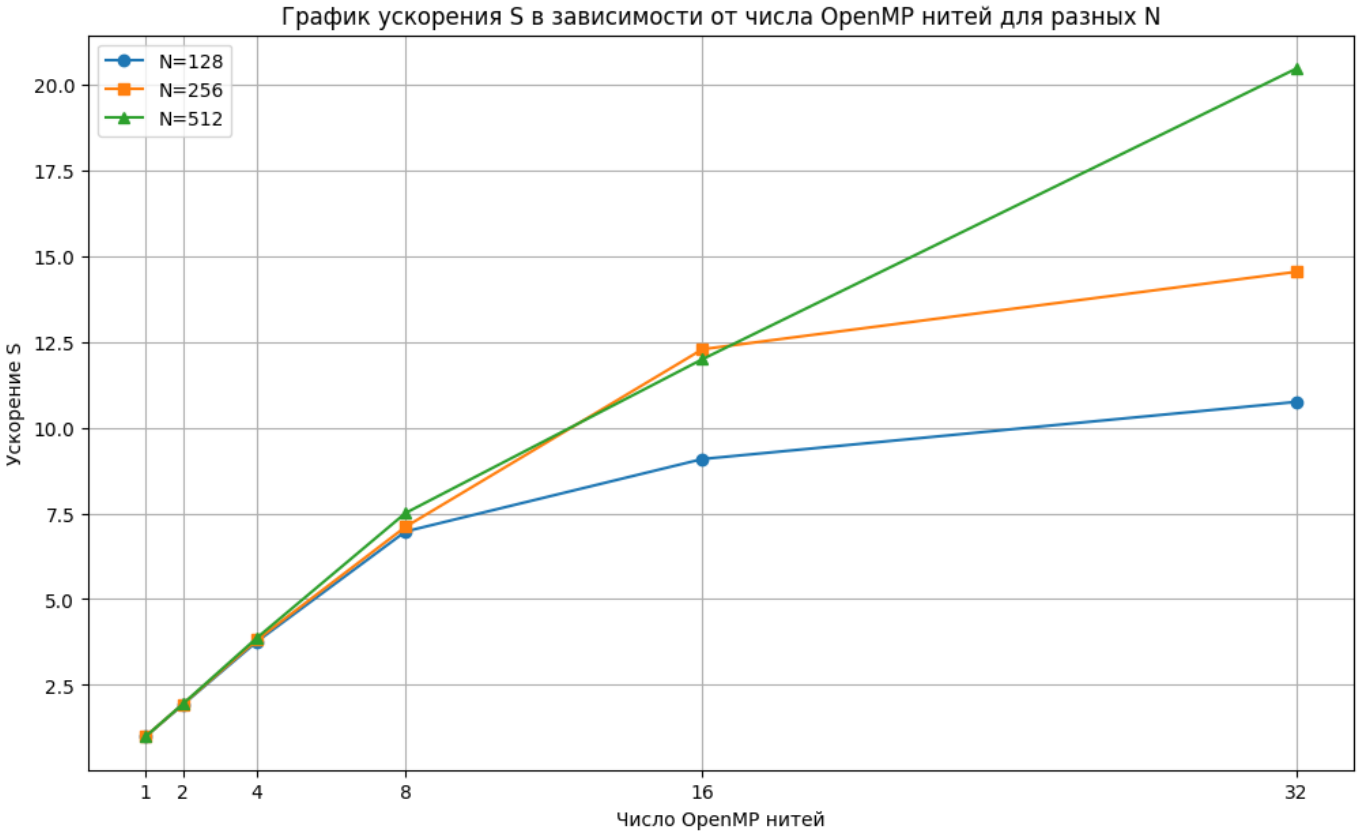
Таблица 5. Lx=Ly=Lz=Pi, N=256

Число OpenMP нитей	N	Время решения T, с	Ускорение S	Погрешность δ
1	256	86,9132	1	8,17E-09
2	256	44,83	1,93872853	8,17E-09
4	256	22,5781	3,849447031	8,17E-09
8	256	12,0508	7,212234872	8,17E-09
16	256	7,19546	12,07889419	8,17E-09
32	256	5,65074	15,38085277	8,17E-09

Таблица 6. Lx=Ly=Lz=Pi, N=512

Число OpenMP нитей	N	Время решения T, с	Ускорение S	Погрешность δ
1	512	659,755	1	2,04E-09
2	512	338,489	1,949117992	2,04E-09
4	512	173,625	3,799884809	2,04E-09
8	512	88,9311	7,418720785	2,04E-09
16	512	65,0372	10,14427128	2,04E-09
32	512	34,1883	19,29768371	2,04E-09

График 3. Ускорение S в зависимости от числа OpenMP нитей для разных N



6.2. MPI

Таблица 7. Lx=Ly=Lz=1, N=128

Число MPI-процессов	N	Общее время T, с	Ускорение S	Погрешность δ
1	128	6,067607	1	1,42E-08
4	128	1,618993	3,74776605	1,42E-08
8	128	0,931247	6,515572131	1,42E-08
16	128	0,618354	9,812513544	1,42E-08
32	128	0,677825	8,951583373	1,42E-08

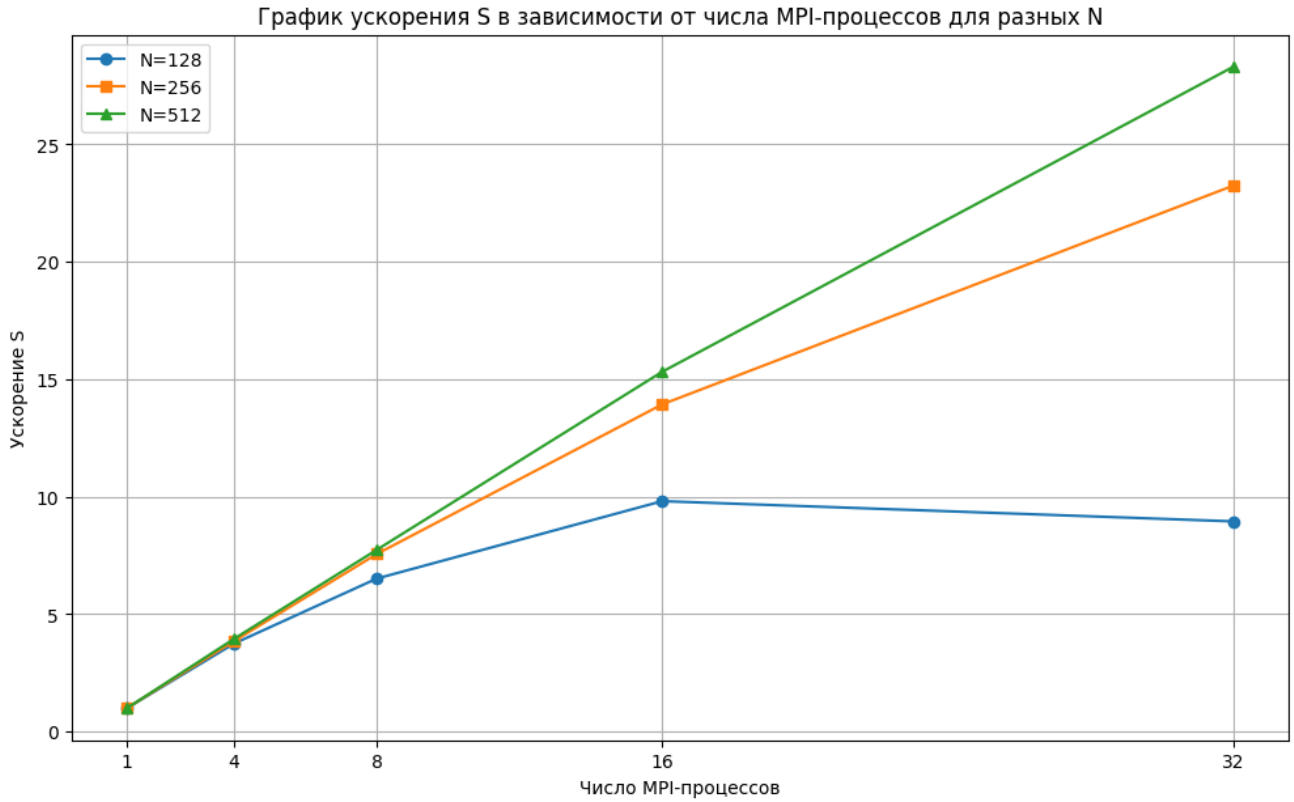
Таблица 8. Lx=Ly=Lz=1, N=256

Число MPI-процессов	N	Общее время T, с	Ускорение S	Погрешность δ
1	256	48,247569	1	3,55E-09
4	256	12,516575	3,854694195	3,55E-09
8	256	6,381031	7,561093027	3,55E-09
16	256	3,463204	13,93148339	3,55E-09
32	256	2,075954	23,24115515	3,55E-09

Таблица 9. Lx=Ly=Lz=1, N=512

Число MPI-процессов	N	Общее время T, с	Ускорение S	Погрешность δ
1	512	387,7	1	8,75E-10
4	512	98,2	3,95	8,75E-10
8	512	50,1	7,74	8,75E-10
16	512	25,3	15,32	8,75E-10
32	512	13,7	28,3	8,75E-10

График 4. Ускорение S в зависимости от числа MPI процессов для разных N



6.3. MPI+OpenMP

Таблица 10. Lx=Ly=Lz=1

Число MPI-процессов	Число OpenMP-нитей	N	Общее время T, с	Ускорение S	Погрешность δ
4	1	128	1,627109	1	7,55E-08
4	2	128	1,110271	1,465506169	7,58E-08
4	4	128	0,879587	1,849855671	7,61E-08
4	8	128	1,115274	1,458932065	7,61E-08
8	1	256	6,682747	1	2,23E-08
8	2	256	4,630854	1,443091706	2,23E-08
8	4	256	3,518603	1,89926144	2,23E-08
8	8	256	4,133705	1,616648261	2,23E-08
8	1	512	53,463234	1	4,81E-09
8	2	512	27,893421	1,92884128	4,81E-09
8	4	512	14,327395	3,73543376	4,81E-09
8	8	512	8,750012	6,12426414	4,81E-09

Таблица 11. Макс. конфигурация (20x8)

Число MPI-процессов	Число OpenMP-нитей	N	Общее время T, с	Погрешность δ
8	20	128	3,02	1,15E-07
8	20	256	4,45	2,91E-08
8	20	512	9,02	7,33E-09

6.4. CUDA + MPI

Таблица 12. Поэтапное сравнение CUDA+MPI (1GPU, 1proc & 2GPU,2procs) vs MPI+OpenMP (20x8=160 threads)

N	Время работы, T (сек,)	MPI+CUDA (1xGPU, 1xProc)	MPI+CUDA (2xGPU, 2 procs)	MPI+OpenMP (160 threads)
128	Инициализация	0,107533	0,089313	0,120956
	Копирование	0,001827	0,003955	0,001686
	Операции обмена	0,000000	0,001328	0,002445
	Решение уравнения	0,007871	0,022434	0,032484
	Освобождение ресурсов	0,000456	0,000652	0,015930
256	Инициализация	0,071638	0,088430	0,124718
	Копирование	0,003054	0,010024	0,008618
	Операции обмена	0,000000	0,003279	0,003777
	Решение уравнения	0,033293	0,042102	0,245274
	Освобождение ресурсов	0,001200	0,001369	0,054064
512	Инициализация	0,086713	0,098885	0,140902
	Копирование	0,009083	0,036945	0,094437
	Операции обмена	0,000000	0,009002	0,036343
	Решение уравнения	0,234688	0,166957	1,952858
	Освобождение ресурсов	0,004538	0,006337	0,219895

Таблица 13. Ошибки RMSE и MAX для CUDA+MPI (1GPU, 1proc & 2GPU,2procs) и MPI+OpenMP (20x8=160 threads)

N	RMSE (1xGPU)	RMSE (2xGPU)	RMSE (160 threads)	MAX (1xGPU)	MAX (2xGPU)	MAX (160 threads)
128	2,34E-10	2,34E-10	5,20E-09	1,44E-09	1,44E-09	1,15E-07
256	5,85E-11	5,85E-11	1,33E-09	3,62E-10	3,62E-10	2,91E-08
512	1,46E-11	1,46E-11	3,35E-10	9,01E-11	9,01E-11	7,33E-09

Таблица 14. Ускорение на ГПУ относительно ЦПУ (1GPU, 1proc & 2GPU,2procs vs MPI+OpenMP, 20x8=160 threads)

N	Время работы, T (сек,)	MPI+CUDA (1xGPU, 1xProc)	MPI+CUDA (2xGPU, 2 procs)
	Инициализация	1,12	1,35
	Копирование	0,92	0,43
	Операции обмена	-	1,84
	Решение уравнения	4,13	1,45
128	Освобождение ресурсов	34,93	24,44
	Инициализация	1,74	1,41
	Копирование	2,82	0,86
	Операции обмена	-	1,15
	Решение уравнения	7,37	5,83
256	Освобождение ресурсов	45,06	39,49
	Инициализация	1,62	1,42
	Копирование	10,40	2,56
	Операции обмена	-	4,04
	Решение уравнения	8,32	11,70
512	Освобождение ресурсов	48,46	34,70

Таблица 15. Ускорение на ГПУ относительно ЦПУ (1GPU, 1proc & 2GPU,2procs vs MPI+OpenMP, 20x8=160 threads)

N	Инициализация		Копирование		Операции обмена		Решение уравнения		Освобождение ресурсов	
	1 ГПУ	2 ГПУ	1 ГПУ	2 ГПУ	1 ГПУ	2 ГПУ	1 ГПУ	2 ГПУ	1 ГПУ	2 ГПУ
128	1,12	1,35	0,92	0,43	-	1,84	4,13	1,45	34,93	24,44
256	1,74	1,41	2,82	0,86	-	1,15	7,37	5,83	45,06	39,49
512	1,62	1,42	10,4	2,56	-	4,04	8,32	11,7	48,46	34,7

График 5. Время для разных этапов в зависимости от N для CUDA+MPI (1 GPU, 1 proc)

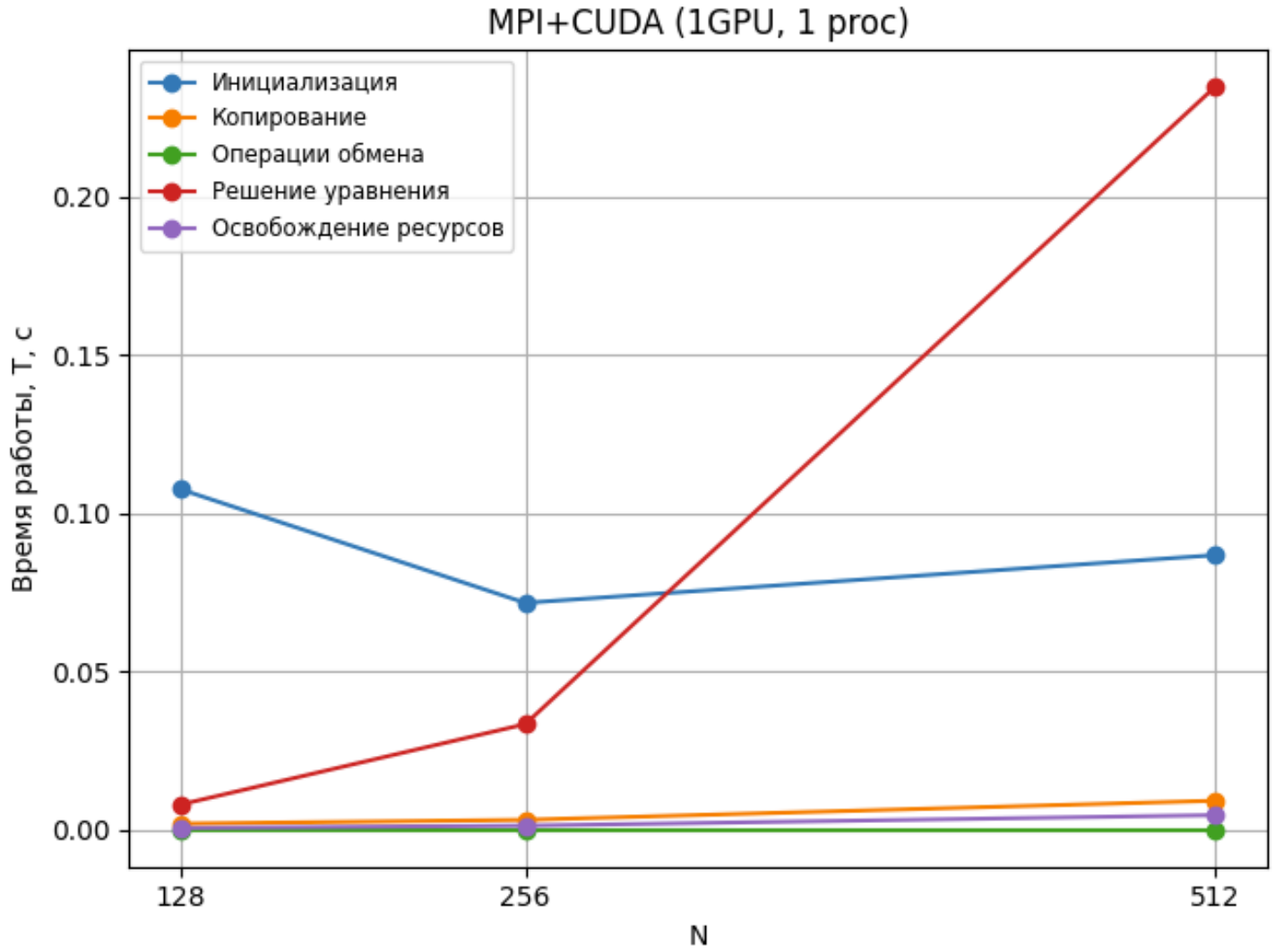


График 6. Время для разных этапов в зависимости от N для CUDA+MPI (2 GPU, 2 proc)

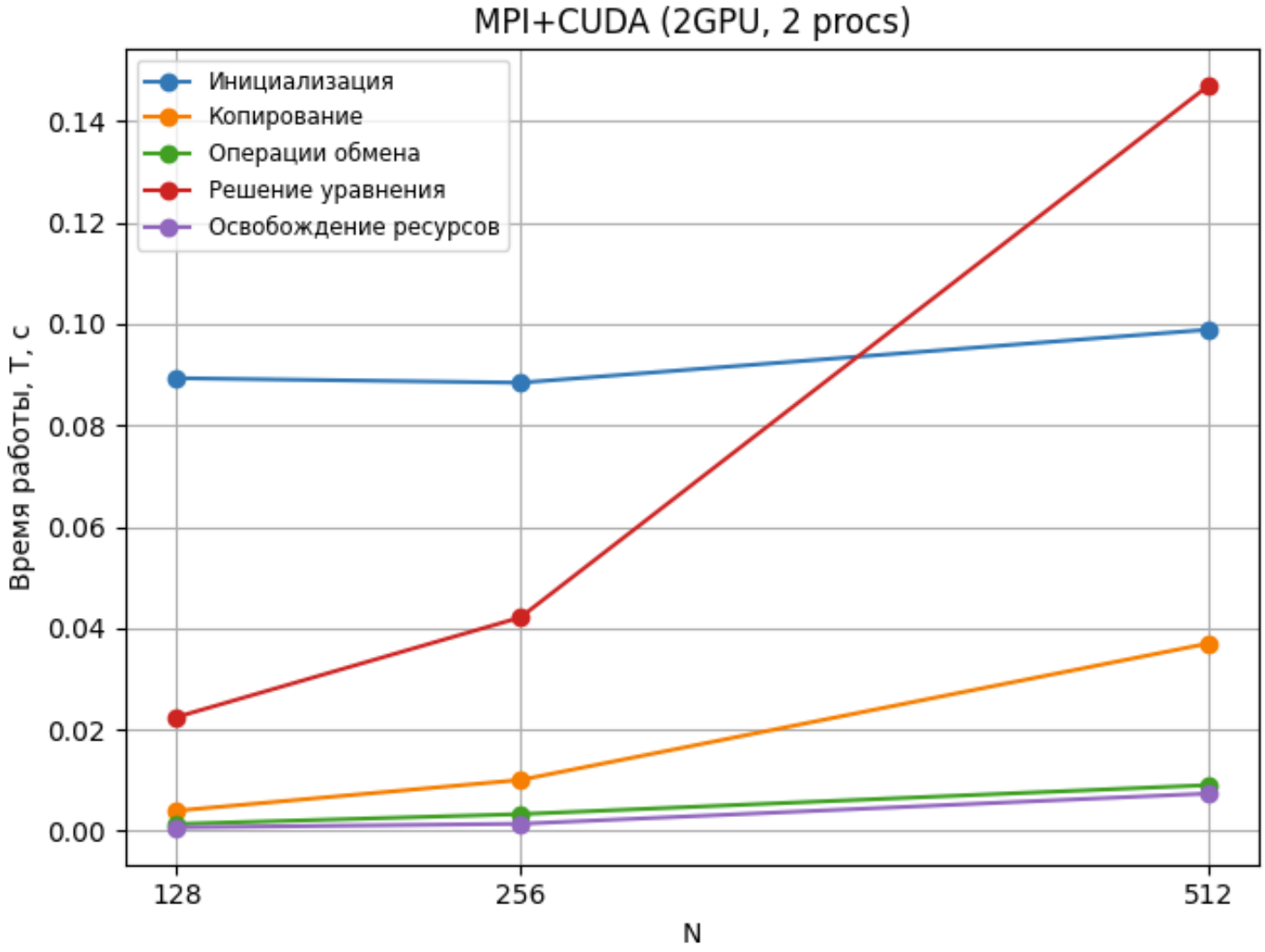
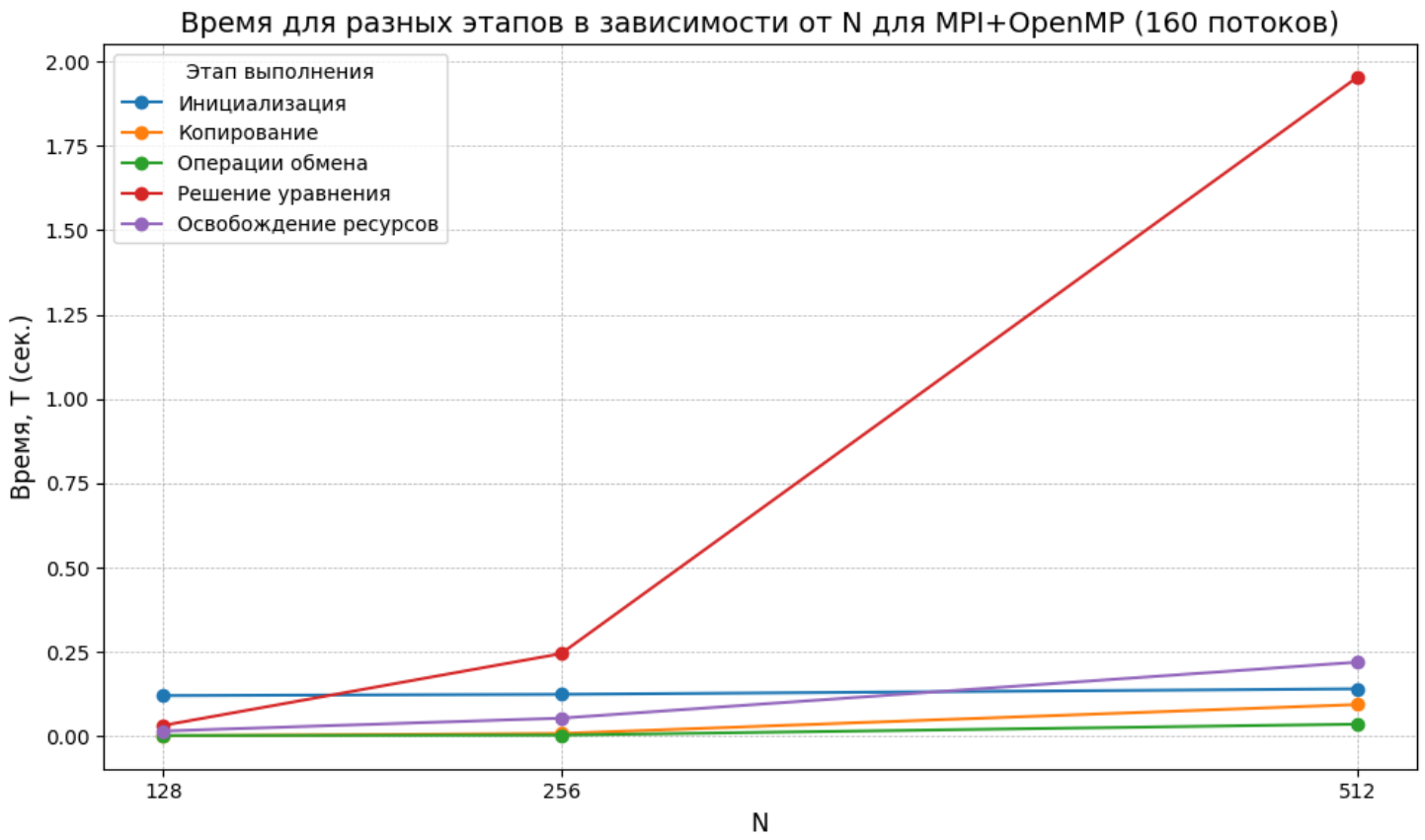


График 7. Время для разных этапов в зависимости от N для MPI+OpenMP (160 потоков)



Сравнение конфигураций CUDA+MPI и MPI+OpenMP между собой

График 8. Сравнение времени решения уравнения для трех конфигураций в зависимости от N

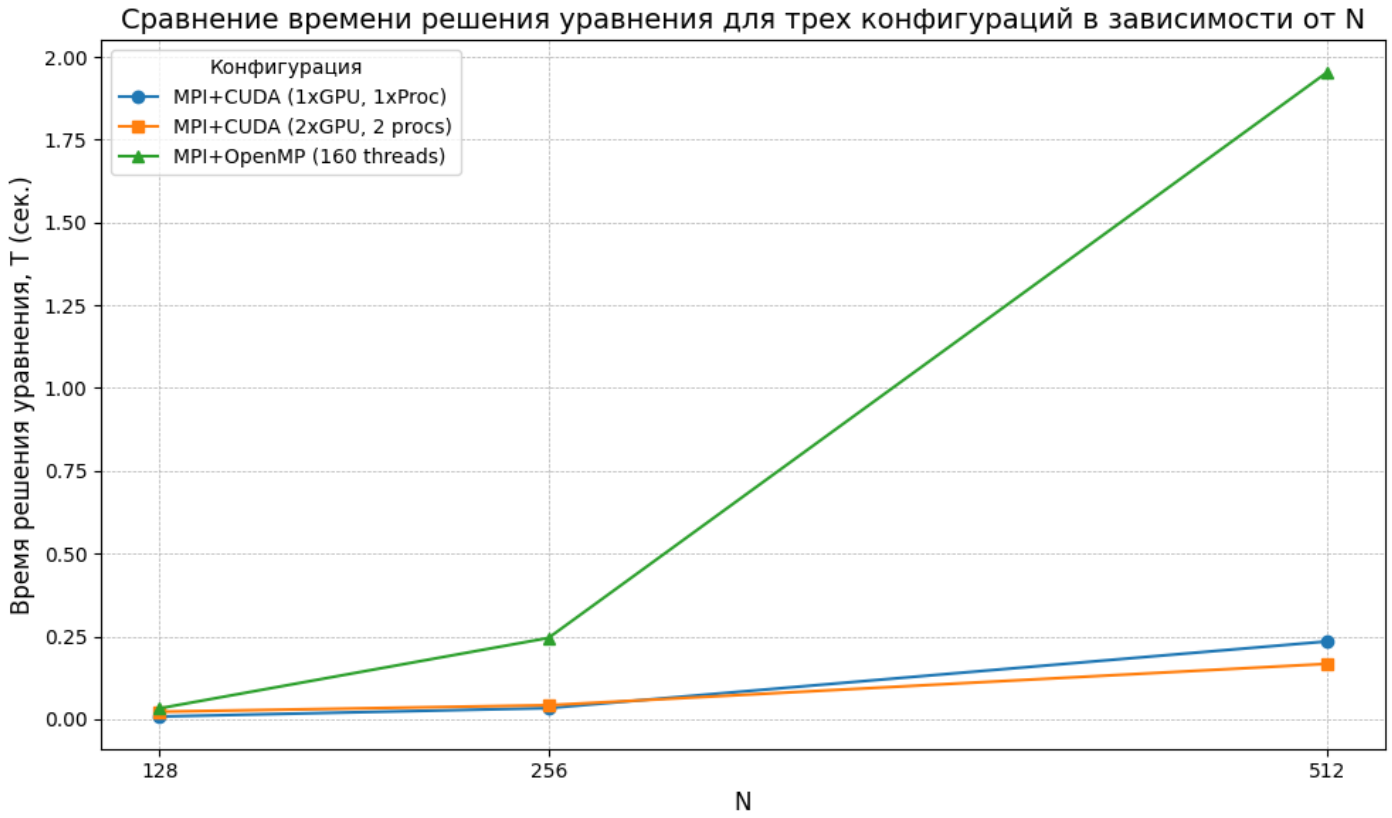


График 9. Сравнение времени решения уравнения в зависимости от N для CUDA+MPI методов

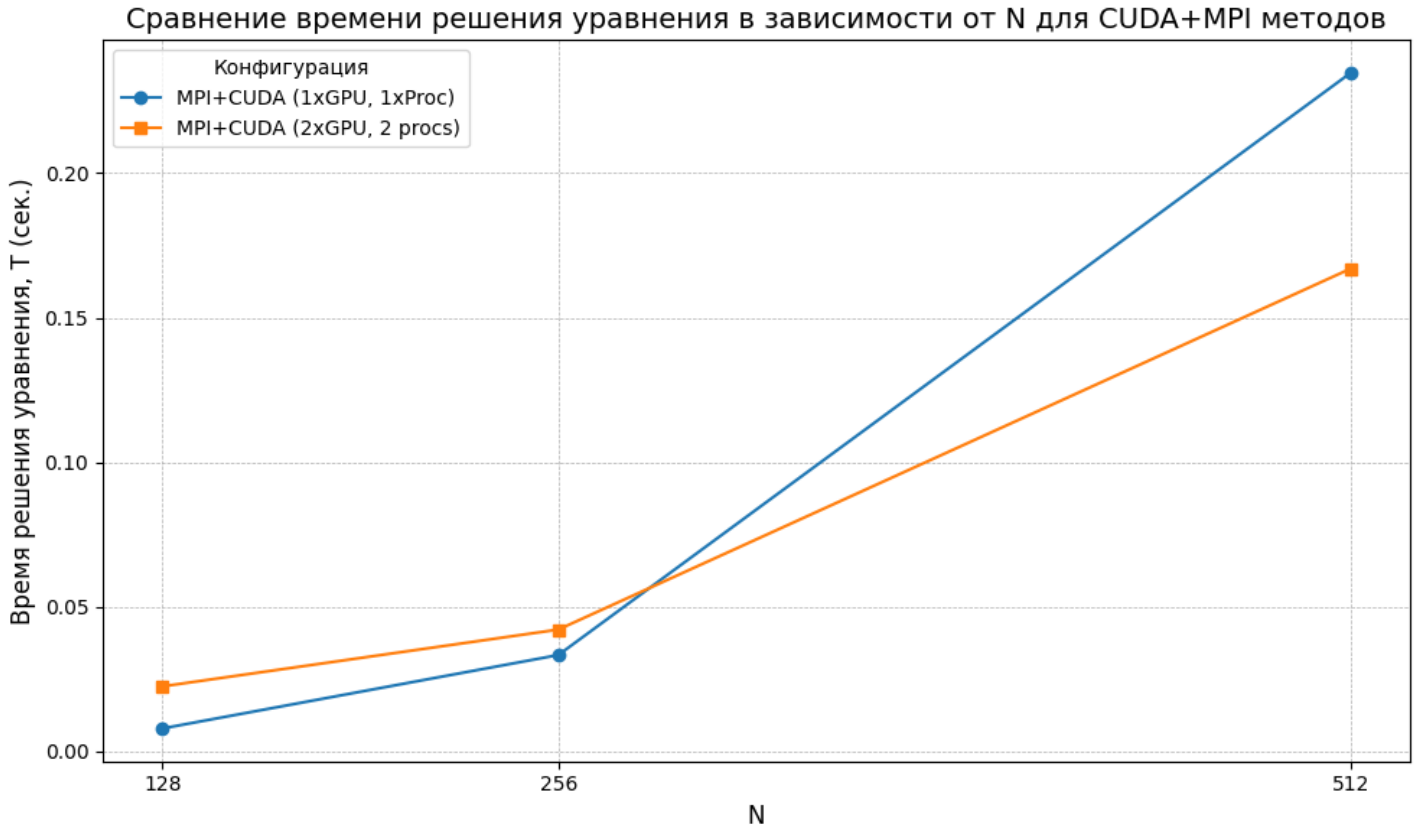


График 10. Сравнение методов по этапам для N=128

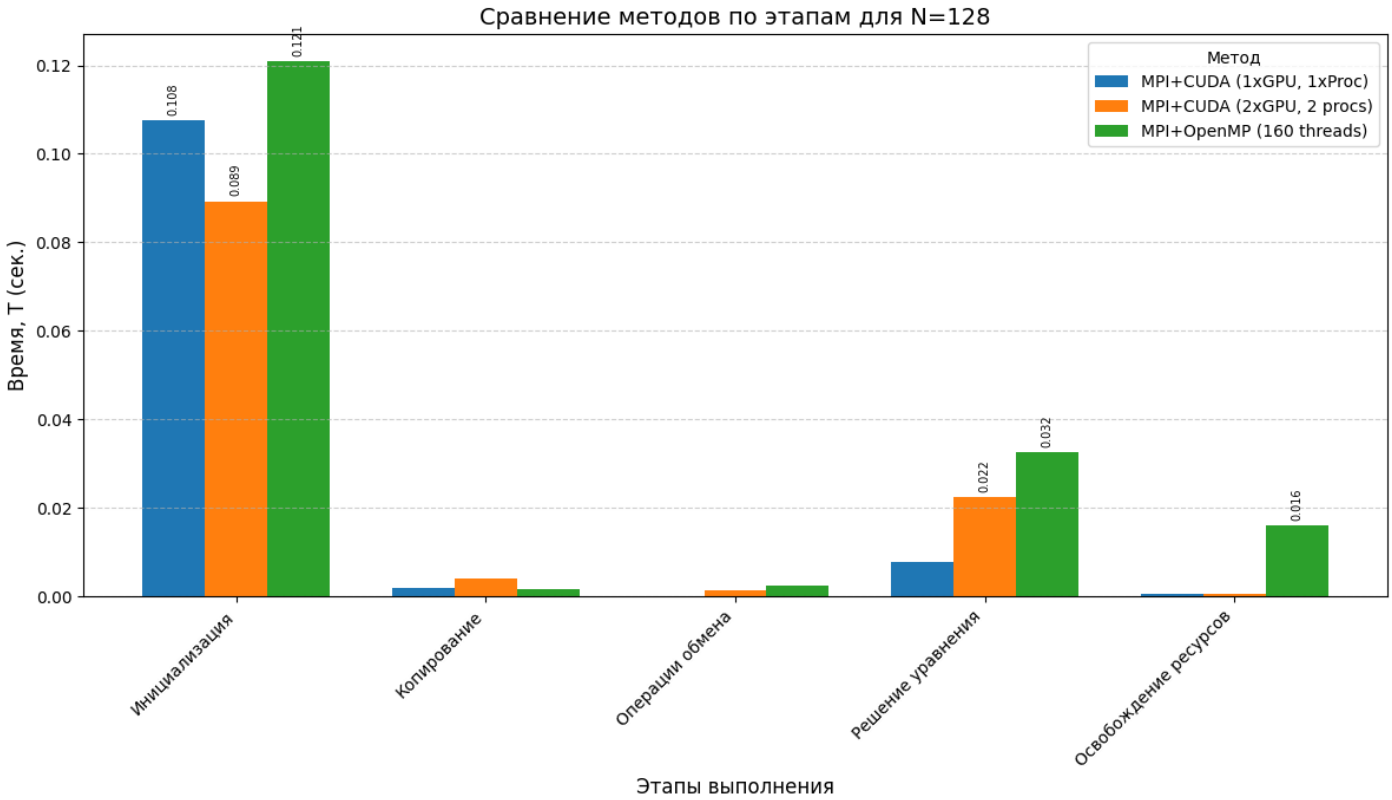


График 11. Сравнение методов по этапам для N=256

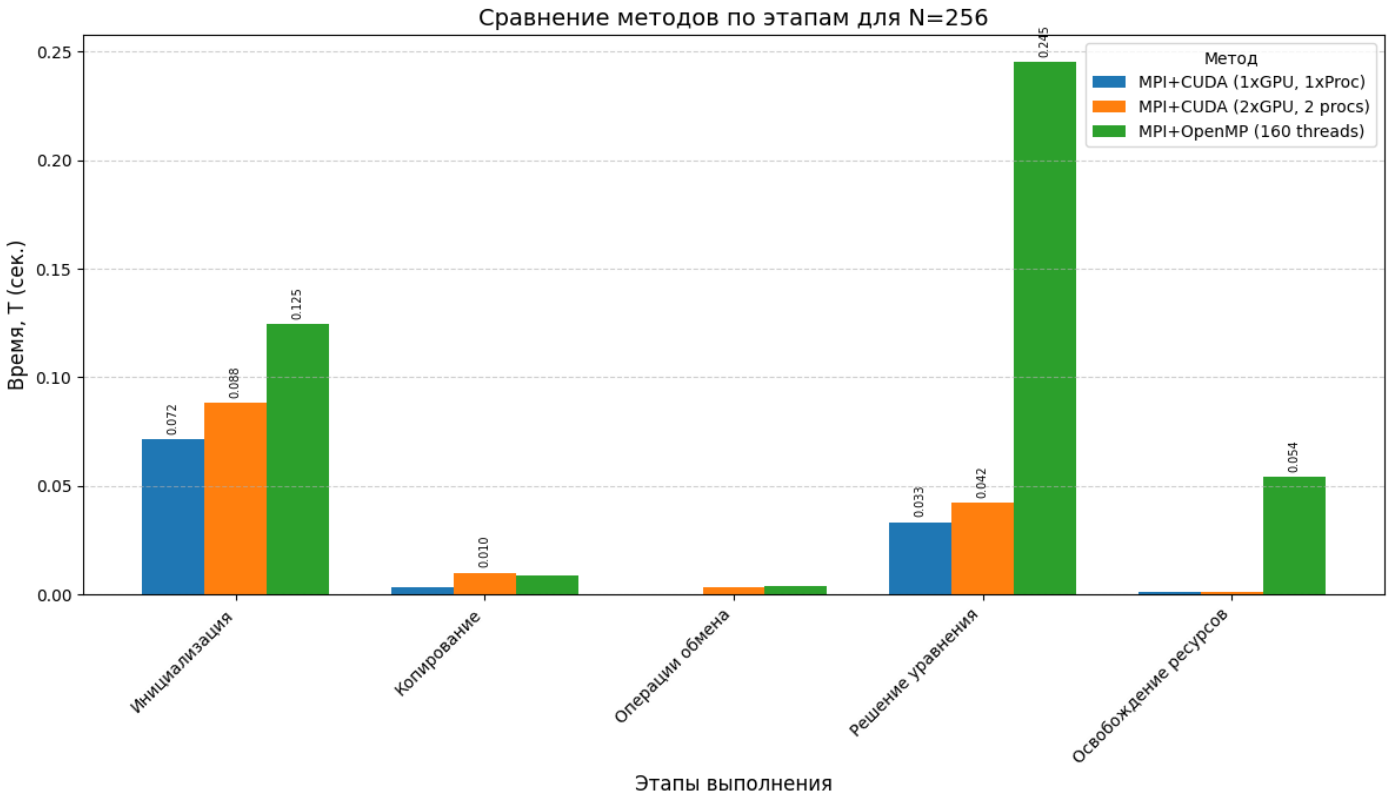
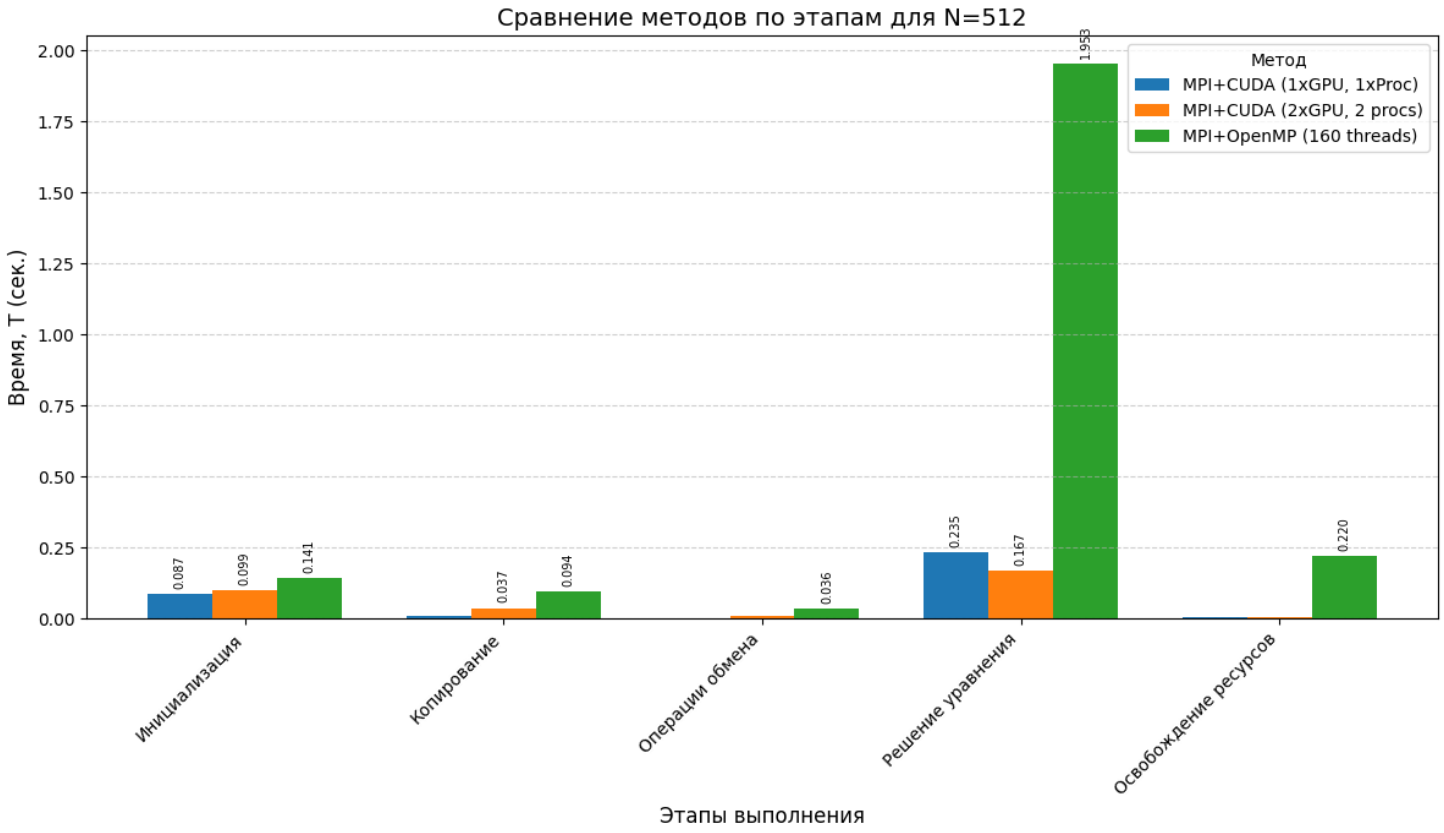


График 12. Сравнение методов по этапам для N=512



7. Выводы

Таблицы 1-6 данные демонстрируют корректное поведение параллельной реализации решения уравнения на **OpenMP**. Анализ трех наборов данных для разных размеров сетки (N=128, N=256, N=512) позволяет сделать несколько важных выводов о масштабируемости и эффективности параллельного алгоритма.

Во-первых, одним из ключевых наблюдений является **увеличение ускорения с ростом размера сетки** при фиксированном числе потоков. Это явление соответствует теоретическому ожиданию, поскольку его можно объяснить через призму закона Густавсона-Барсиса (мы проходили его ещё год назад назад в рамках темы про закон Амдала на курсе "*Высокопроизводительные вычисления и параллельная обработка данных*", читающийся Якобовским М.В., а также два года назад, на 4-м курсе, в рамках курса "*Параллельная обработка данных*", который читает Воеводин В.В.), который утверждает, что при увеличении размера задачи доля последовательной части алгоритма уменьшается относительно общей вычислительной нагрузки. Формально, если обозначить последовательную часть как `p`, а параллельную как `1-p`, то ускорение определяется как:

```
S(n) = (1 - p + p * n) / (1 - p + p * n / n) = n - (n - 1) * p
```

При увеличении размера задачи `p` уменьшается, что приводит к росту ускорения `S(n)` при фиксированном `n` (числе потоков) – что полностью соответствует полученным результатам. Другими словами, в данном случае, для малых N (128) доля последовательных операций (инициализация, коммуникация, синхронизация) составляет значительную часть общего времени выполнения. При увеличении N до 512 объем вычислительной работы растет как $O(N^3)$, тогда как накладные расходы на параллельное выполнение остаются относительно постоянными или растут значительно медленнее. Это приводит к тому, что относительная доля параллельной части алгоритма увеличивается, что отражается в росте ускорения.

Эффективность параллельного выполнения (отношение ускорения к числу потоков) демонстрирует следующую динамику:

- для N=128 при 32 потоках: 0.336
- для N=256 при 32 потоках: 0.454
- для N=512 при 32 потоках: 0.639

Этот рост эффективности с увеличением размера задачи является характерным для вычислительно-интенсивных алгоритмов. Причина аналогична описанной выше: накладные расходы на создание и управление потоками, синхронизацию и разделение данных становятся менее значительными по сравнению с растущим объемом чисто вычислительной работы.

Для N=128 ускорение практически перестает расти при переходе от 16 к 32 потокам (с 9.08 до 10.74), что указывает на достижение предела эффективного использования

потоков для этого размера задачи. Для больших N этот предел сдвигается, позволяя эффективно использовать большее количество потоков.

Соотношение времени выполнения для одного потока при увеличении N:
- N=128 → N=256: 86.97 / 11.97 ≈ 7.26
- N=256 → N=512: 661.68 / 86.97 ≈ 7.61

Эти значения близки к теоретическому ожиданию 8 (поскольку объем данных растет как N^3), что подтверждает корректность реализации алгоритма и его ожидаемую вычислительную сложность $O(N^3)$. Небольшое отклонение от 8 может быть объяснено различиями в использовании кэш-памяти и других архитектурных особенностях при обработке данных разного размера.

Погрешность (MAX) демонстрирует ожидаемое поведение для метода второго порядка аппроксимации:
- N=128: 3.3077e-09
- N=256: 8.2733e-10 (уменьшение в ~4.00 раза)
- N=512: 2.0685e-10 (уменьшение в ~4.00 раза)

Такое уменьшение погрешности ровно в 4 раза при увеличении N в 2 раза является характерным для методов второго порядка точности (как уже доказано и объяснено выше) и подтверждает корректность реализации разностной схемы. Формально, для метода второго порядка погрешность δ связана с шагом сетки h соотношением $\delta = O(h^2)$. Поскольку $h = L/(N-1)$, то при увеличении N в 2 раза h уменьшается примерно в 2 раза, а погрешность уменьшается в 4 раза, т.е. всё соответствует теории

Таблицы 7-10 также демонстрируют корректное поведение параллельной **MPI-реализации** решения уравнения и позволяют провести сравнение с предыдущими результатами OpenMP-версии показывает некоторые различия в поведении. Одним из ключевых наблюдений является существенное различие во времени выполнения на одном процессоре/потоке (при этом важно отметить, что оно всё равно больше последовательной версии из-за затрат на инициализацию). Это свидетельствует о том, что MPI-реализация более оптимизирована для вычислений на отдельном узле (тут важно отметить, что это не противоречит теории, так как различные параллельные парадигмы могут иметь разные накладные расходы даже на одном узле)

В остальном можно заметить схожий характер поведения, который описан и объяснен выше (поэтому ниже краткое резюме):
- ускорение растет с увеличением размера задачи при фиксированном числе процессов, что соответствует закону Густавсона-Барсиса и объясняется уменьшением относительной доли последовательных операций
- эффективность параллельного выполнения значительно выше для больших N, что указывает на правильную архитектуру распределенной реализации, где коммуникационные накладные расходы становятся менее значительными по мере роста вычислительной нагрузки

- соотношение времени выполнения для разных N близко к теоретическому ожиданию для алгоритма сложности $O(N^3)$, подтверждая корректность реализации
- погрешность остается на приемлемом уровне и демонстрирует незначительный рост с увеличением числа процессов (и корректно увеличивается с ростом N) , что характерно для распределенных реализаций

Ещё одно важное замечание в том, что сравнение с OpenMP-реализацией показывает, что MPI-реализация имеет лучшую масштабируемость на большом числе процессов, особенно для крупных задач, что является ожидаемым результатом, так как MPI лучше подходит для распределенных вычислений на кластере.

Анализ поведения гибридной **MPI+OpenMP** (далее – “гибридная реализация”) реализации показывает, что результаты в целом соответствуют теоретическим ожиданиям, однако наблюдаются некоторые особенности, требующие подробного рассмотрения. Гибридная реализация сохраняет преимущества распределенной памяти MPI при одновременном использовании преимуществ общей памяти OpenMP, но накладные расходы на коммуникацию между процессами и синхронизацию внутри процессов приводят к увеличению времени выполнения по сравнению с чистой MPI реализацией.

Одной из ключевых особенностей является наблюдаемое уменьшение ускорения при увеличении числа OpenMP-нитей в пределах одного MPI-процесса:

- для N=128, 4 MPI-процессов:
    - 1 OpenMP-нить: ускорение 1.755
    - 8 OpenMP-нитей: ускорение 1.458
- для N=256, 8 MPI-процессов:
    - 1 OpenMP-нить: ускорение 1.223
    - 8 OpenMP-нитей: ускорение 1.616

Это поведение не является ожидаемым, но в то же время не является и парадоксальным, а отражает сложную динамику накладных расходов в гибридной архитектуре. При малых размерах задачи (N=128) дополнительные накладные расходы на синхронизацию и управление OpenMP-нитями могут превышать выгоду от параллельных вычислений. Для более крупных задач (N=256) увеличение числа нитей дает положительный эффект, так как доля вычислительной работы возрастает относительно накладных расходов (а при N=512 можно видеть преобладание параллельной доли в работе программы, о чём свидетельствует более высокое и более близкое к ожидаемому ускорение)

Таким образом, общая тенденция показывает, что гибридная реализация обеспечивает лучшую масштабируемость по сравнению с чистыми MPI или OpenMP реализациями при больших размерах задач, что является ключевым преимуществом гибридного подхода. Эффективность гибридной реализации возрастает с увеличением размера задачи, что подтверждает правильность выбора архитектуры для решения вычислительно-интенсивных задач на современных кластерных системах с многоядерными узлами.

Полученные данные по гибридной **CUDA+MPI** реализации демонстрируют корректное поведение, соответствующее теоретическим ожиданиям. Анализ показывает, что результаты согласуются с фундаментальными принципами параллельных вычислений и законами Амдала и Густавсона-Барсиса (далее подробные обоснования будут опущены дабы не дублировать уже имеющиеся выше аналогичные мысли)

Корректность реализации разностной схемы второго порядка подтверждается *таблицей 13*: характер убывания RMSE и MAX-ошибок соответствует ожиданию, уменьшаясь в ~4 раза при увеличении N в два раза (т.к. h = L/(N-1) ⇒ увеличении N в 2 раза h уменьшается в ~2 раза, ⇒ погрешность должна уменьшиться в 4 раза), при этом сами ошибки находятся в допустимом диапазоне

Как видно из *таблиц 12* и *14-15*, CUDA+MPI обеспечивает значительное ускорение решения по сравнению с MPI+OpenMP (даже с условием, что используем все ресурсы узла на ЦПУ – суммарно 160 потоков), что объясняется высоким потенциалом векторных вычислений, а также видно, что при “малой” сетке (N=128) 1 ГПУ немного выигрывает в скорости решения (ускорение , тогда как для “большой” (N=512) – ускорение больше с двумя ГПУ, что соответствует ожиданиям (поскольку, как выше было обосновано, доля параллельной части увеличивается ⇒ увеличивается и ускорение), при этом, как видно на *графиках 5-7*, время других этапов (инициализация, копирование, освобождение ресурсов) невелико и значительно не меняется ни при росте N, ни при смене конфигурации между 1 ГПУ и 2-мя ГПУ (увеличивается только время копирования, что кажется логичным).

По таблице 14-15 заметно, что для “малых” сеток конфигурация с 1 ГПУ даёт большее ускорение, чем с 2 ГПУ. Полагаю, что это можно объяснить тем, что для N=128 и N=256 вычислительная нагрузка недостаточна для эффективной загрузки двух ГПУ. В данных случаях реализация с 1 ГПУ лучше использует доступные ресурсы, избегая дополнительных накладных расходов на управление и синхронизацию двух ГПУ. Но вот для большой сетки при N=512 у нас уже заметно больше вычислений, что позволяет эффективнее загрузить 2 ГПУ, и теперь уже этот вариант позволяет получить бОльшее ускорение относительно гибридной версии со 160 потоками, чем вариант с 1 ГПУ

Таким образом, методы на основе CUDA+MPI демонстрируют корректную реализацию разностной схемы решения уравнения и показывают заметное превосходство над другими методами распараллеливания в производительности для данной задачи (*таблицы 12 и 14-15*, *графики 8-12*)

Приложение 1. Компиляция и запуск

Для сдачи CUDA+MPI части задания был создан github-репозиторий, включающий исходный код всех версий описанных в отчёте программ::

`https://github.com/Bikmish/ParallelComputing2025`

Проект содержит 4 папки с реализациями для разных технологий параллельных вычислений:

- `task1_OMP` - многопоточная реализация, используя OpenMP
- `task2_MPI` - распределенная реализация с передачей сообщений с использованием MPI
- `task3_MPI_OMP` - гибридная реализация MPI+OpenMP
- `task4_MPI_CUDA` - реализация, используя гибридную версию CUDA+MPI (вычисления перенесены на ГПУ)

Для каждого проекта написан `Makefile`, в котором прописаны команды компиляции и запусков индивидуально для каждого задания.Запуски проводились во всевозможных конфигурациях, статистика которых в отчёте к заданию отражена на соответствующих таблицах и графиках (таблицы 1-15 и графики 1-12 отчёта).

(ниже для каждой версии программы перечислены всевозможные конфигурации запусков, а также способы компиляции)

OpenMP

1) **компиляция**

- локально: `clang++ -O3 -std=c++11 -fopenmp solver.cpp main.cpp -o solver`
  - или через Makefile: `make build` (компиляция локально с `clang++`)
- на Polus: `g++ -O3 -std=c++11 -fopenmp solver.cpp main.cpp -o solver`
- или через Makefile: `make build_polus` (компиляция на Polus с `g++`)

2) **тестирование и сбор статистики для отчёта**

   `make test_n256_l1_thAll` (N=256, L=1, потоки: 1,2,4,8,16,32)
   `make test_n256_lPi_thAll` (N=256, L=Pi, потоки: 1,2,4,8,16,32)
   `make test_full_config` (N:128,256,512; L:1,Pi; потоки:1,2,4,8,16,32)
   `make test_n256_l1_thAll_polus` (N=256, L=1, потоки:1,2,4,8,16,32)
   `make test_full_config_polus` (N:128,256,512; L:1,Pi; потоки:1,2,4,8,16,32)

Для полноты отчёта и корректности выводов для данной версии и всех последующих были проведены всевозможные запуски с разными конфигурациями параметров и разной привязкой с последующем усреднением статистики

MPI

1) **компиляция**:
   - `mpixlC -O3 -std=c++11 -o solverMPI_only main.cpp`
   - через Makefile: `make build` или просто `make`
2) **тестирование и сбор статистики для отчёта**:
   `make run_all` (N:128,256,512; L:1,Pi; процессы:1,2,4,8,16,32)
   `make clean` (очистка результатов)
   `make help` (справка по командам)

MPI+OpenMP

1) **компиляция**:
   - `mpixlC -O3 -qsmp=omp -std=c++11 -o solverMPI main.cpp`
   - или через `Makefile`: `make build` или просто `make`
2) **тестирование и сбор статистики для отчёта**:
   `make run_all` (N:128,256,512; L:1,Pi; MPI:4,8 проц; OMP:1,2,4,8 потоков)
   `make run_all_1` (N:128,256,512; L:1,Pi; MPI:1,4,8,16,32 проц; OMP:1 поток)
   `make run_all_max` (N:128,256,512; L:1,Pi; MPI:8 проц; OMP:20 потоков)
   `make clean` (очистка результатов)
   `make help` (справка по командам)

CUDA+MPI

1) **компиляция**:
   - `nvcc -O3 -arch=sm_35 -ccbin mpic++ -o solverCUDA main.cu -x cu -Xcompiler -Wall -std=c++11`
   - или через Makefile: `make build`
2) **тестирование и сбор статистики для отчёта**:
   `make run_test` (N=256, L=Pi, MPI:8 проц, 1 GPU shared) - тестовый запуск
   `make run_all_single_gpu` (N:128,256,512; L=Pi; MPI:1,2,4,8,16 проц; 1 GPU shared) - shared-режим
   `make run_all_single_gpu_ex` (N:128,256,512; L=Pi; MPI:1,2,4,8,16 проц; 1 GPU exclusive) - эксклюзивный режим
   `make run_all_double_gpu` (N:128,256,512; L=Pi; MPI:1,2,4,8,16 проц; 2 GPU shared) - shared-режим
   `make run_all_double_gpu_ex` (N:128,256,512; L=Pi; MPI:1,2,4,8,16 проц; 2 GPU exclusive) - эксклюзивный режим
   `make clean` (очистка)

Для CUDA-реализации главным образом сравнивались конфигурации "1 ГПУ + 1 процесс" и "2 ГПУ + 2 процесс", остальные конфигурации запускались чисто из научного интереса.

Подробный анализ статистики, выводы, таблицы, графики и подробности реализаций каждой из версий программ описаны выше в отчёте.
Также для сдачи CUDA+MPI части задания был создан github-репозиторий, включающий исходный код всех версий описанных в отчёте программ:
`https://github.com/Bikmish/ParallelComputing2025`