Московский государственный университет
имени М.В.Ломоносова

ЗАДАНИЕ ПО КУРСУ

«Суперкомпьютерное моделирование и технологии»

Вариант: 4

Студент: Семеняк Г.А.
Группа: 628

Сентябрь 2025 - Декабрь 2025

Москва 2025

[1] Код решения лежит на Гитхабе: https://github.com/twist13227/sctm

Содержание

1 Математическая постановка дифференциальной задачи

В трехмерной замкнутой области

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

для $(0 < t \le T]$ требуется найти решение $u(x, y, z, t)$ уравнения в частных производных

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

с начальными условиями

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

$$\left.\frac{\partial u}{\partial t}\right|_{t=0} = 0, \tag{3}$$

при условии, что на границах области заданы однородные граничные условия первого рода

$$u(0, y, z, t) = 0, \quad u(L_x, y, z, t) = 0, \tag{4}$$

$$u(x, 0, z, t) = 0, \quad u(x, L_y, z, t) = 0, \tag{5}$$

$$u(x, y, 0, t) = 0, \quad u(x, y, L_z, t) = 0, \tag{6}$$

либо периодические граничные условия

$$u(0, y, z, t) = u(L_x, y, z, t), \quad u_x(0, y, z, t) = u_x(L_x, y, z, t), \tag{7}$$

$$u(x, 0, z, t) = u(x, L_y, z, t), \quad u_y(x, 0, z, t) = u_y(x, L_y, z, t), \tag{8}$$

$$u(x, y, 0, t) = u(x, y, L_z, t), \quad u_z(x, y, 0, t) = u_z(x, y, L_z, t). \tag{9}$$

Конкретная комбинация граничных условий определяется индивидуальным вариантом задания (см. п. 5).

2 Численный метод решения задачи

Содержание данного пункта основано на материале книги [2]. Для численного решения задачи введем на Ω сетку $\omega_{h\tau} = \bar{\omega}_h \times \omega_\tau$, где

$$T = T_0,$$

$$L_x = L_{x_0},\ L_y = L_{y_0},\ L_z = L_{z_0}$$

$$\bar{\omega}_h = \{(x_i = ih_x, y_j = jh_y, z_k = kh_z), i, j, k = 0, 1, \ldots, N, h_x N = L_x, h_y N = L_y, h_z N = L_z\},$$

$$\omega_\tau = \{t_n = n\tau, n = 0, 1, \ldots, K, \tau K = T\}.$$

Через ω_h обозначим множество внутренних, а через γ_h — множество граничных узлов сетки $\bar{\omega}_h$.

Для аппроксимации исходного уравнения (1) с однородными граничными условиями (4)-(6) и начальными условиями (2)-(3) воспользуемся следующей системой уравнений:

$$\frac{u_{ijk}^{n+1} - 2u_{ijk}^n + u_{ijk}^{n-1}}{\tau^2} = a^2 \Delta_h u^n, \quad (x_i, y_j, z_k) \in \omega_h, \quad n = 1, 2, \ldots, K-1,$$

Здесь Δ_h — семиточечный разностный аналог оператора Лапласа:

$$\Delta_h u^n = \frac{u_{i-1,j,k}^n - 2u_{i,j,k}^n + u_{i+1,j,k}^n}{h^2} + \frac{u_{i,j-1,k}^n - 2u_{i,j,k}^n + u_{i,j+1,k}^n}{h^2} + \frac{u_{i,j,k-1}^n - 2u_{i,j,k}^n + u_{i,j,k+1}^n}{h^2}.$$

Приведенная выше разностная схема является явной — значения u_{ijk}^{n+1} на $(n+1)$-м шаге можно явным образом выразить через значения на предыдущих слоях.

Для начала счета (т.е. для нахождения u_{ijk}^2) должны быть заданы значения $u_{ijk}^0, u_{ijk}^1, (x_i, y_j, z_k) \in \omega_h$. Из условия (2) имеем

$$u_{ijk}^0 = \varphi(x_i, y_j, z_k), \quad (x_i, y_j, z_k) \in \omega_h. \tag{10}$$

Простейшая замена начального условия (3) уравнением $(u_{ijk}^1 - u_{ijk}^0)/\tau = 0$ имеет лишь первый порядок аппроксимации по τ. Аппроксимацию второго порядка по τ и h дает разностное уравнение

$$\frac{u_{ijk}^1 - u_{ijk}^0}{\tau} = a^2 \frac{\Delta_h \varphi(x_i, y_j, z_k)}{2}, \quad (x_i, y_j, z_k) \in \omega_h. \tag{11}$$

$$u_{ijk}^1 = u_{ijk}^0 + a^2 \frac{\Delta_h \varphi(x_i, y_j, z_k)}{2}. \tag{12}$$

Разностная аппроксимация для периодических граничных условий выглядит следующим образом

$$u^{n+1}_{0jk} = u^{n+1}_{Njk}, \quad u^{n+1}_{1jk} = u^{n+1}_{N+1jk},$$
$$u^{n+1}_{i0k} = u^{n+1}_{iNk}, \quad u^{n+1}_{i1k} = u^{n+1}_{iN+1k},$$
$$u^{n+1}_{ij0} = u^{n+1}_{ijN}, \quad u^{n+1}_{ij1} = u^{n+1}_{ijN+1},$$

$i, j, k = 0, 1, \ldots, N$.

Для вычисления значений $u^0, u^1 \in \gamma_h$ допускается использование аналитического значения u, которое задается в программе еще для вычисления погрешности решения задачи.

Вычисления далее проводятся для следующей аналитической функции:

$$u(x, y, z, t) = \sin\left(\frac{3\pi}{L_x}x\right) \cdot \sin\left(\frac{2\pi}{L_y}y\right) \cdot \sin\left(\frac{2\pi}{L_z}z\right) \cdot \cos(a_t\, t + 4\pi),$$

$$a_t = 2\pi, \quad a^2 = \frac{1}{\frac{9}{L_x^2} + \frac{4}{L_y^2} + \frac{4}{L_z^2}}, \quad L_x = L_y = L_z = 1$$

Со следующими граничными условиями:

$$u(0, y, z, t) = 0, \quad u(L_x, y, z, t) = 0, \tag{4}$$
$$u(x, 0, z, t) = u(x, Ly, z, t), \quad u(x, L_y, z, t) = u(x, Ly, z, t), \tag{5}$$
$$u(x, y, 0, t) = u(x, y, Lz, t), \quad u(x, y, 0, t) = u(x, y, Lz, t), \tag{6}$$

3 Программная реализация (OpenMP)

Для хранения трёхмерной сеточной функции используется одномерный вектор, а обращение к элементу выполняется через функцию вычисления индекса. Такое хранение экономит память и ускоряет доступ. Размер сетки по каждой координате задаётся переменной `N_PLUS_1 = N+1`.

Аналитическая функция $u(x, y, z, t)$ реализована отдельно и используется как для начальных условий, так и для оценки погрешности. По координате x задаются граничные условия Дирихле ($u = 0$ на границе), а по y и z реализованы периодические условия — значения на противоположных границах совпадают.

Вычисление лапласиана выполняется через центральные разности с учётом соответствующих граничных условий. Эволюция во времени реализована в функции `time_step_evolution`, где используется схема второго порядка:

$$u^{n+1} = 2u^n - u^{n-1} + a^2\tau^2\Delta u^n.$$

Для первого шага применяется модифицированный коэффициент 0.5 для корректного старта по времени.

Параллелизация реализована с помощью OpenMP. Используется директива `#pragma omp parallel for collapse(3)`, что позволяет эффективно распараллелить трёхмерные циклы по узлам сетки. Вычисление максимальной ошибки производится с редукцией `reduction(max:error)`, чтобы избежать гонок при поиске глобального максимума.

В основном цикле по времени сначала выполняется первый шаг, затем на каждом шаге вызывается функция эволюции, после чего массивы переставляются и при необходимости выводится текущая погрешность. В результате программа реализует явную схему второго порядка с периодическими условиями по y, z и Дирихле по x, эффективно распараллеленную средствами OpenMP.

3.1 Результаты OpenMP ($L = 1$)

Вычисления производились с числом временных шагов $K = 20$.

Таблица 1: Результаты OpenMP

Число OpenMP нитей	Число узлов сетки N^3	Время решения T	Ускорение S	Погрешность δ
1	128^3	7.871	1.00	7.6104e-03
2	128^3	4.161	1.89	7.6104e-03
4	128^3	2.262	3.48	7.6104e-03
8	128^3	1.217	6.47	7.6104e-03
16	128^3	0.958	8.22	7.6104e-03
32	128^3	0.704	11.18	7.6104e-03
1	256^3	59.800	1.00	3.8078e-03
2	256^3	30.937	1.93	3.8078e-03
4	256^3	16.866	3.55	3.8078e-03
8	256^3	10.891	5.49	3.8078e-03
16	256^3	6.839	8.74	3.8078e-03
32	256^3	4.526	13.22	3.8078e-03

3.2 Результаты OpenMP ($L = \pi$)

Вычисления производились с числом временных шагов $K = 20$.

4 Программная реализация (MPI)

MPI-реализация использует трёхмерную декомпозицию вычислительной области. Сетка процессов автоматически формируется в виде трёхмерного тора с помощью `MPI_Dims_creat` что обеспечивает сбалансированное распределение нагрузки. Каждый процесс хранит

Таблица 2: Результаты OpenMP

Число OpenMP нитей	Число узлов сетки N^3	Время решения T	Ускорение S	Погрешность δ
1	128^3	8.019	1.000	8.3247e-04
2	128^3	4.109	1.952	8.3247e-04
4	128^3	2.250	3.565	8.3247e-04
8	128^3	1.223	6.559	8.3247e-04
16	128^3	0.922	8.701	8.3247e-04
32	128^3	0.709	11.314	8.3247e-04
1	256^3	59.939	1.000	4.1637e-04
2	256^3	31.778	1.887	4.1637e-04
4	256^3	19.505	3.073	4.1637e-04
8	256^3	10.463	5.728	4.1637e-04
16	256^3	7.225	8.295	4.1637e-04
32	256^3	4.674	12.818	4.1637e-04

локальный блок данных с ghost-слоями толщиной в один узел на границах своей подобласти по всем декомпозированным направлениям.

Граничные условия задаются следующим образом:

- По оси x: непериодические условия (Дирихле $u = 0$ на внешних границах $x = 0$ и $x = L$)
- По осям y и z: периодические условия

Декомпозиция по периодическим направлениям (y и z) всегда требует обмена ghost-слоями между соседними процессами. Для непериодического направления (x) обмен выполняется только если декомпозиция затрагивает эту ось (при числе процессов > 1 по x), но значения на внешних границах всегда фиксируются как $u = 0$.

Хранение данных и вычисление индекса

Трёхмерная сеточная функция хранится в одномерном массиве в порядке $i \to j \to k$. Локальный вектор содержит $(N_i^{\text{local}} + 2) \cdot (N_j^{\text{local}} + 2) \cdot (N_k^{\text{local}} + 2)$ элементов, где дополнительные слои используются для ghost-элементов. Индексация выполняется по формуле:

$$\text{idx}(i, j, k) = i \cdot (N_j^{\text{local}} + 2) \cdot (N_k^{\text{local}} + 2) + j \cdot (N_k^{\text{local}} + 2) + k,$$

где i, j, k — локальные индексы в пределах подобласти процесса.

Граничные условия и обмен данными

Для поддержания корректных вычислений лапласиана на границах подобластей используется механизм ghost-слоёв:

- По периодическим осям (y, z) выполняется полный обмен границами между соседними процессами
- По непериодической оси (x) ghost-слои обмениваются только между внутренними границами процессов, а внешние границы ($x = 0$ и $x = L$) явно устанавливаются в ноль
- Для обмена используются неблокирующие операции `MPI_Isend` и `MPI_Irecv`, что позволяет перекрыть коммуникации с вычислениями
- Циклический обмен для периодических границ реализуется автоматически через топологию коммуникатора `cart_comm`

На каждом временном шаге выполняется следующая последовательность операций:

1. Обмен ghost-слоями по всем декомпозированным периодическим направлениям
2. Вычисление дискретного лапласиана $\Delta_h u^n$ на внутренних узлах:
$$\Delta_h u^n_{i,j,k} = \frac{u^n_{i+1,j,k} - 2u^n_{i,j,k} + u^n_{i-1,j,k}}{h_x^2} + \frac{u^n_{i,j+1,k} - 2u^n_{i,j,k} + u^n_{i,j-1,k}}{h_y^2} + \frac{u^n_{i,j,k+1} - 2u^n_{i,j,k} + u^n_{i,j,k-1}}{h_z^2}$$
3. Обновление решения по схеме второго порядка точности:
$$u^{n+1} = 2u^n - u^{n-1} + a^2\tau^2\Delta_h u^n$$
4. Принудительная установка $u = 0$ на внешних границах по оси x после каждого шага

Для оценки точности решения на каждом шаге:

1. Каждый процесс вычисляет локальную максимальную ошибку по своей подобласти:
$$\varepsilon_{\text{local}} = \max_{i,j,k} |u^{\text{calc}}_{i,j,k} - u^{\text{exact}}_{i,j,k}|$$
2. С помощью коллективной операции `MPI_Reduce` с операцией `MPI_MAX` вычисляется глобальный максимум ошибки
3. Результат доступен только на корневом процессе (rank 0) для минимизации накладных расходов

Таким образом, MPI-версия обеспечивает масштабирование по числу процессов и позволяет распределить объём вычислений и памяти по кластерам.

Ниже приведены усреднённые по пяти запускам значения времени, погрешности и ускорения.

4.1 Результаты MPI ($L = 1$)

Таблица 3: Результаты MPI при $L = 1$

MPI	N^3	Время T	Ускорение S	Погрешность δ	Var(T)
1	128^3	9.814	1.00	7.610e−03	0.653
2	128^3	4.015	2.44	7.610e−03	0.119
4	128^3	2.097	4.68	7.610e−03	0.00125
8	128^3	1.415	6.93	7.610e−03	0.0373
16	128^3	0.831	11.81	7.610e−03	0.00385
32	128^3	0.562	17.45	7.610e−03	0.00178
1	256^3	81.274	1.00	3.808e−03	30.026
2	256^3	37.036	2.19	3.808e−03	8.199
4	256^3	15.577	5.22	3.808e−03	0.0904
8	256^3	9.703	8.37	3.808e−03	1.592
16	256^3	6.059	13.41	3.808e−03	0.0196
32	256^3	3.024	26.88	3.808e−03	0.0069

4.2 Результаты MPI ($L = \pi$)

Таблица 4: Результаты MPI при $L = \pi$

MPI	N^3	Время T	Ускорение S	Погрешность δ	Var(T)
1	128^3	9.250	1.00	8.325e−04	2.279
2	128^3	4.769	1.94	8.325e−04	0.532
4	128^3	2.066	4.48	8.325e−04	0.00141
8	128^3	1.421	6.51	8.325e−04	0.0602
16	128^3	1.072	8.63	8.325e−04	0.0477
32	128^3	0.580	15.95	8.325e−04	0.000671
1	256^3	66.829	1.00	4.164e−04	14.733
2	256^3	36.675	1.82	4.164e−04	0.100
4	256^3	16.132	4.14	4.164e−04	0.945
8	256^3	10.218	6.54	4.164e−04	2.891
16	256^3	5.843	11.44	4.164e−04	0.0246
32	256^3	3.555	18.80	4.164e−04	0.416

Отметим, что MPI-версия демонстрирует почти линейное ускорение до 16 процессов, после чего влияние коммуникаций начинает снижать эффективность.

5 Программная реализация (MPI+OpenMP)

Гибридная реализация использует двухуровневый параллелизм: распределение данных между узлами кластера через MPI и параллельные вычисления на многоядерных процессорах узла через OpenMP. Трёхмерная декомпозиция области сохраняется (формируемая `MPI_Dims_create`), но каждый MPI-процесс дополнительно использует

несколько OpenMP-потоков для обработки своей локальной подобласти. Это позволяет эффективно использовать современные вычислительные архитектуры с иерархией памяти и десятками ядер на узел.

Граничные условия и структура декомпозиции полностью соответствуют MPI-версии:

- По оси x: условия Дирихле (u=0 на внешних границах x=0 и x=L)
- По осям y и z: периодические условия
- Обмен ghost-слоями выполняется только через MPI (в основном потоке)
- Все OpenMP-потоки работают с уже синхронизированными данными

Параллельная обработка данных

Вычислительно-ёмкие участки кода распараллеливаются с помощью OpenMP:

- Инициализация начальных условий и аналитического решения
- Вычисление дискретного лапласиана на внутренних узлах
- Обновление решения по временной схеме
- Заполнение локальных ghost-слоёв при отсутствии соседей по декомпозиции
- Подсчёт локальной максимальной ошибки

Для вложенных циклов используется директива `collapse(3)`, объединяющая три уровня вложенности в единое пространство итераций:

```
#pragma omp parallel for collapse(3)
for (int i = 1; i <= local_ni; ++i) {
    for (int j = 1; j <= local_nj; ++j) {
        for (int k = 1; k <= local_nk; ++k) {
        }
    }
}
```

Это обеспечивает равномерное распределение нагрузки даже при небольших размерах локальных блоков.

Оптимизация локальных операций

Для заполнения ghost-слоёв в случае отсутствия декомпозиции по некоторой оси (dims[i]=1) используются параллельные циклы:

```
if (dims[1] == 1) {
    #pragma omp parallel for collapse(2)
    for (int i = 0; i < local_ni + 2; ++i) {
        for (int k = 0; k < local_nk + 2; ++k) {
            u[local_index(i, 0, k)] = u[local_index(i, local_nj, k)];
            u[local_index(i, local_nj + 1, k)] = u[local_index(i, 1, k)];
        }
    }
}
```

Такой подход эффективно использует внутренний параллелизм узла для операций, которые в чистой MPI-версии выполнялись бы последовательно.

Синхронизация между уровнями параллелизма

Ключевой принцип гибридной реализации — строгое разделение коммуникаций и вычислений:

1. Все MPI-коммуникации (`MPI_Isend`, `MPI_Irecv`, `MPI_Waitall`) выполняются в последовательной части кода до входа в OpenMP-регионы
2. OpenMP-параллелизм активируется только после полной синхронизации данных через MPI
3. Для агрегации максимальной ошибки используется комбинация локальной OpenMP-редукции и глобального `MPI_Reduce`:

```
double local_error = 0.0;
#pragma omp parallel for collapse(3) reduction(max:local_error)
for (int i = 1; i <= local_ni; ++i) {
}
MPI_Reduce(&local_error, &global_error, 1, MPI_DOUBLE, MPI_MAX, 0, cart_comm);
```

Это исключает гонки данных и гарантирует корректность результатов.

Управление ресурсами

Перед началом вычислений настраивается среда OpenMP:

```
omp_set_dynamic(0);
omp_set_num_threads(omp_get_max_threads());
```

Отключение динамического распределения потоков (`omp_set_dynamic(0)`) обеспечивает предсказуемую производительность, а установка максимального числа потоков позволяет полностью использовать вычислительные ресурсы узла.

Таким образом, гибридная MPI+OpenMP реализация обеспечивает:

- Снижение накладных расходов на коммуникации за счёт уменьшения числа MPI-процессов
- Эффективное использование кэш-памяти за счёт локальности данных в рамках одного узла
- Масштабируемость как на уровне кластера (MPI), так и на уровне многоядерного процессора (OpenMP)
- Гибкость в настройке соотношения MPI-процессов и OpenMP-потоков под конкретную архитектуру

Ниже приведены усреднённые по пяти запускам значения времени, погрешности и ускорения для гибридной реализации.

5.1 Результаты MPI+OpenMP ($L = 1$)

Таблица 5: Результаты MPI+OpenMP, $L = 1$

MPI	OMP	N^3	Время T	Погрешность δ	Var(T)
4	1	128^3	0.863	7.610e-03	0.00678
4	2	128^3	0.490	7.610e-03	0.00476
4	4	128^3	0.410	7.610e-03	0.00826
4	8	128^3	0.239	7.610e-03	0.000736
8	1	256^3	3.466	3.808e-03	0.121
8	2	256^3	2.441	3.808e-03	0.00369
8	4	256^3	1.767	3.808e-03	0.00675
8	8	256^3	1.240	3.808e-03	0.00262

5.2 Результаты MPI+OpenMP ($L = \pi$)

Таблица 6: Результаты MPI+OpenMP, $L = \pi$

MPI	OMP	N^3	Время T	Погрешность δ	Var(T)
4	1	128^3	0.945	8.325e-04	0.00151
4	2	128^3	0.652	8.325e-04	0.00661
4	4	128^3	0.333	8.325e-04	0.00189
4	8	128^3	0.293	8.325e-04	0.00197
8	1	256^3	3.620	4.164e-04	0.01585
8	2	256^3	2.452	4.164e-04	0.00733
8	4	256^3	1.682	4.164e-04	0.00102
8	8	256^3	1.227	4.164e-04	0.000182

6 Программная реализация (MPI+CUDA)

Гибридная реализация MPI+CUDA следует той же архитектуре, что и MPI+OpenMP: распределение данных между узлами через MPI с дополнительным внутренним параллелизмом на уровне вычислительных устройств.

Граничные условия и структура декомпозии полностью соответствуют предыдущим версиям:

- По оси x: условия Дирихле (отражение)
- По осям y и z: периодические условия
- Обмен ghost-слоями выполняется через неблокирующие MPI-операции

- Все вычисления на GPU работают с данными, уже синхронизированными через MPI

Управление памятью и данными

Ключевое отличие от CPU-реализации — явное управление памятью между хостом и устройством:

- Каждый MPI-процесс управивает тремя основными массивами на GPU: `d_prev`, `d_curr`, `d_next`
- Для обмена гало-слоями используются специальные буферы `cudaHostAlloc`, обеспечивающие максимальную пропускную способность при копировании между CPU и GPU
- Структура `HaloBuffers` объединяет все промежуточные буферы (устройство + хост).

Память выделяется с учётом 1-слойного гало по каждой границе: `alloc_ni = local_ni + 2`.

CUDA-ядра и вычислительные этапы

Вся вычислительная логика реализована через специализированные CUDA-ядра:

- `core_kernel` — обрабатывает внутреннюю область (`[1, local_ni]`) по трёхуровневой схеме
- `halo_kernel` — обновляет гало-слои с условиями отражения на внешних границах
- `initial_step_kernel` — вычисляет второй временной слой на GPU
- `pack_*/unpack_*/apply_periodic_y/z` — утилитарные ядра для подготовки данных

Особое внимание уделено периодическим условиям по осям Y и Z:

- При однопроцессной декомпозиции по оси (`dims[i] = 1`) периодичность реализуется через специальные ядра `apply_periodic_y/z`
- При многопроцессной декомпозиции периодичность обеспечивается через MPI-обмен

Гало-обмен и перекрытие коммуникаций

Процедура `do_time_step` реализует следующий пайплайн вычислений:

1. Запуск ядер `pack_*/unpack_*/apply_periodic` для подготовки данных
2. Асинхронное копирование гало-слоёв
3. Запуск неблокирующих MPI-операций обмена
4. Запуск `core_kernel` без ожидания завершения MPI — обеспечивает перекрытие вычислений и коммуникаций
5. Синхронизация MPI через `MPI_Waitall`
6. Копирование полученных данных на GPU и распаковка в гало-слои
7. Обновление гало через `halo_kernel`

Этот подход максимально эффективно использует ресурсы GPU и сети, минимизируя простои.

Обработка самосвязи

Дополнительно обрабатываются самосвязи (`neighbor == rank`) при периодических условиях в однопроцессной декомпозиции по осям Y и Z:

- При `neighbor == rank` обмен выполняется через локальную копию `memcpy` вместо MPI
- Все MPI-вызовы защищены проверкой `neighbor != rank`
- Это исключает тупик, возникающий при попытке бмена с самим собой

Оптимизация вычислений и редукция

Для вычисления максимальной ошибки используется thrust с `counting_iterator`, что исключает необходимость аллокации промежуточных буферов:

```
thrust::counting_iterator<size_t> first(0);
thrust::counting_iterator<size_t> last(total);
thrust::transform_reduce(first, last, f, 0.0, thrust::maximum<double>());
```

Это обеспечивает оптимальное использование памяти и производительности GPU.

Управление ресурсами GPU

- Проверяется наличие CUDA-устройств (`cudaGetDeviceCount`)
- Каждый MPI-процесс привязывается к отдельному GPU
- Используется макрос `SAFE_CALL` (взят из лекций) для всех CUDA-вызовов

Это гарантирует корректную работу на узлах с несколькими GPU и позволяет эффективно распределять вычислительную нагрузку.

Ниже приведены результаты производительности и анализа эффективности для данной реализации.

6.1 Результаты MPI+CUDA ($L = 1$)

В таблице 7 представлены результаты запусков для трёх размеров задачи: $N = 256, 512, 768$ (1024 не используется, поскольку не пролазит по памяти) . Все времена измерены в секундах. Ускорение рассчитано относительно MPI-версии на 20 процессов.

Таблица 7: Сравнение производительности различных реализаций

Версия	Процессы	Потоки/GPU	N	Время (с)	Ускорение
Serial	1	1	256	62.42	—
MPI	20	1	256	4.53	1.00
MPI+OpenMP	20	8	256	1.05	4.32
MPI+CUDA	1	1 GPU	256	0.23	19.87
MPI+CUDA	2	2 GPU	256	0.17	27.29
Serial	1	1	512	475.67	—
MPI	20	1	512	28.48	1.00
MPI+OpenMP	20	8	512	6.96	4.10
MPI+CUDA	1	1 GPU	512	1.34	21.32
MPI+CUDA	2	2 GPU	512	0.68	41.75
Serial	1	1	768	1587.71	—
MPI	20	1	768	86.55	1.00
MPI+OpenMP	20	8	768	19.52	4.43
MPI+CUDA	1	1 GPU	768	4.65	18.61
MPI+CUDA	2	2 GPU	768	2.31	37.43

6.2 Детализация времени выполнения для MPI+CUDA

В таблице 8 приведено распределение времени по компонентам для CUDA-версий. Указаны усреднённые значения времени одного шага (в миллисекундах).

Таблица 8: Детализация времени выполнения MPI+CUDA

Компонент	$N = 256$		$N = 512$		$N = 768$	
	1 GPU	2 GPU	1 GPU	2 GPU	1 GPU	2 GPU
pack & post	0.004	0.085	0.005	0.257	0.005	0.525
core kernel	**7.475**	**2.671**	**40.230**	**19.325**	**149.885**	**70.433**
wait MPI	0.000	1.894	0.000	0.929	0.000	2.641
recv & unpack	0.165	0.115	0.386	0.394	0.896	0.843
halo kernel	**0.579**	**0.240**	**1.987**	**1.167**	**5.574**	**3.084**
error calc	5.547	1.878	24.225	13.335	77.265	39.067
single step	8.241	5.023	42.632	22.094	156.385	77.577

6.3 Расчёт эффективности

6.3.1 Расчёт теоретической ограниченной производительности (TBP)

Для оценки потенциальной производительности использовалась модель теоретически ограниченной производительности (TBP). Исходные характеристики системы:

- Пиковая производительность (DP): TPP = 4.7 TFLOPs = 4700 GFLOPs
- Пропускная способность памяти: BW = 700 GBs = 700×10^9 байт/с

Арифметическая интенсивность (AI) для каждого вычислительного ядра:

- `core_kernel`: 12 FLOP, 64 байта, AI = $12/64 = 0.1875$ FLOP/byte
- `halo_kernel`: 12 FLOP, 72 байта, AI = $12/72 \approx 0.1667$ FLOP/byte
- `pack/unpack`: 0 FLOP, 16 байт, AI = 0 FLOP/byte

Расчёт TBP по формуле $TBP = \min(TPP, BW \times AI)$:

- `core_kernel`: $\min(4700, 700 \times 0.1875) = 131.25$ GFLOPs
- `halo_kernel`: $\min(4700, 700 \times 0.1667) \approx 116.69$ GFLOPs
- `pack/unpack`: $\min(4700, 0) = 0$ GFLOPs (операции копирования памяти)

Таблица 9: Теоретическая ограниченная производительность (TBP) для компонент системы

Компонент	AI (FLOP/byte)	TBP (GFLOPs)
`core_kernel`	0.1875	131.25
`halo_kernel`	0.1667	116.69
`pack/unpack`	0.0000	0.00

6.3.2 Результаты расчётов эффективности

Фактическая производительность и эффективность относительно TBP для `core_kernel`:

- $N = 256$: 27.25 GFLOPs (20.76% от TBP)
- $N = 512$: 40.27 GFLOPs (30.68% от TBP)
- $N = 768$: 36.41 GFLOPs (27.74% от TBP)

Фактическая производительность и эффективность относительно TBP для `halo_kernel`:

- $N = 256$: 18.00 GFLOPs (15.42% от TBP)
- $N = 512$: 22.06 GFLOPs (18.91% от TBP)
- $N = 768$: 20.13 GFLOPs (17.25% от TBP)

Таблица 10: Эффективность компонентов системы относительно TBP

Компонент	$N = 256$	$N = 512$	$N = 768$
`core_kernel`	20.76%	30.68%	27.74%
`halo_kernel`	15.42%	18.91%	17.25%
`pack/unpack`	—	—	—

6.4 Анализ результатов

1. **Ускорение MPI+CUDA**: Версия с 2 GPU демонстрирует ускорение до 41.75x относительно MPI (для $N = 512$).
2. **Масштабируемость**: При увеличении размера задачи (N от 256 до 512) ускорение растёт, что указывает на хорошую масштабируемость. Однако при $N = 768$ наблюдается небольшое снижение эффективности из-за роста доли накладных расходов на обмен данными.
3. **Временные компоненты**: В версии с 1 GPU доминирующим компонентом является `core_kernel` (91% времени для $N = 512$). При переходе к 2 GPU доля коммуникационных операций (`wait MPI`) возрастает до 4% для $N = 768$, но остаётся приемлемой.
4. **Эффективность TBP**: Наблюдаемая эффективность 20–30% от TBP по моему мнению обоснована ограниченной пропускной способностью памяти. Рост эффективности с $N = 256$ до $N = 512$ объясняется лучшим соотношением вычислений к объёму данных.

5. **Сравнение с MPI+OpenMP**: MPI+CUDA показывает ускорение в 4–5 раз по сравнению с MPI+OpenMP, что демонстрирует преимущества архитектуры GPU для данной вычислительной задачи.

Таким образом, реализация MPI+CUDA удовлетворяет всем требованиям задания: она работает быстрее MPI и MPI+OpenMP, демонстрирует хорошую масштабируемость и достигает разумной эффективности относительно теоретических пределов производительности.