

Отчёт по заданию: Решение двумерного уравнения Пуассона методом фиктивных областей с использованием MPI+Cuda

Вариант: №9

Студент: Ма Синьюэ 617

Цель работы

Целью работы является исследование производительности параллельного алгоритма решения двумерного уравнения Пуассона в области сложной формы методом фиктивных областей с использованием предобусловленного метода сопряжённых градиентов. Необходимо реализовать и сравнить несколько вариантов параллельной реализации (MPI, OpenMP, MPI+CUDA), оценить их ускорение и эффективность относительно последовательной версии, а также проанализировать вклад различных составляющих времени исполнения (инициализация, вычисления на CPU/GPU, обмены данными, завершение программы).

Постановка задачи

Рассматривается уравнение Пуассона

$$-\Delta u(x, y) = f(x, y)$$

в области

$$D = \{(x, y) : x^2 + 4y^2 < 1\},$$

вложенной в прямоугольник

$$\Omega = [A_1, B_1] \times [A_2, B_2],$$

с граничным условием Дирихле

$$u(x, y) = 0 \quad \text{на } \partial D.$$

В работе используется аналитическое решение вида

$$u(x, y) = \frac{1}{10}(1 - x^2 - 4y^2),$$

что позволяет контролировать точность численного решения.

Для аппроксимации задачи используется метод фиктивных областей: исходная область D расширяется до прямоугольника Ω , а коэффициент $k(x, y)$ в операторе

$$-\nabla \cdot (k(x, y) \nabla u(x, y)) = f(x, y)$$

принимает значение $k = 1$ внутри эллипса и $k = \varepsilon \ll 1$ вне эллипса. На прямоугольнике Ω строится равномерная сетка размера $M \times N$ с шагами h_1, h_2 и выписывается разностная схема, приводящая к большой разреженной СЛАУ.

Для решения разностной задачи используется предобусловленный метод сопряжённых градиентов (PCG) с диагональным предобуславливателем. Требуется:

- реализовать алгоритм в последовательном варианте (без MPI/OpenMP/GPU) и получить базовое время T_{seq} ;
- реализовать параллельный вариант на MPI (разбиение области по процессам), а также вариант с OpenMP (распараллеливание по нитям) и измерить времена $T_{\text{MPI}}(p)$ и $T_{\text{OMP}}(t)$;
- реализовать гибридный вариант MPI+CUDA, в котором операции применения матрицы, предобуславливателя и скалярные произведения выполняются на GPU, а обмен граничными слоями осуществляется через MPI;
- для каждой версии вычислить ускорение и эффективность по отношению к последовательной программе:

$$S(p) = \frac{T_{\text{seq}}}{T(p)}, \quad E(p) = \frac{S(p)}{p},$$

где p — число процессов (для MPI) или нитей (для OpenMP);

- для MPI+CUDA дополнительно проанализировать вклады во время выполнения: суммарное время работы GPU-ядер, время копирования данных между CPU и GPU, время коммуникаций MPI, время работы предобуславливателя на CPU, время вычисления скалярных произведений;
- сравнить полученные численные решения с аналитическим решением, оценив погрешность, и объяснить характер полученных значений ускорения и эффективности.

Метод и реализация

Область, сетка и правая часть

Рассматривается эллиптическая область

$$D = \{(x, y) : x^2 + 4y^2 < 1\},$$

вложенная в прямоугольный контейнер

$$A_1 = -1, \quad B_1 = 1, \quad A_2 = -0.6, \quad B_2 = 0.6.$$

Сетка равномерная:

$$x_i = A_1 + i h_x, \quad i = 0, \dots, M; \quad y_j = A_2 + j h_y, \quad j = 0, \dots, N,$$

где $h_x = \frac{B_1 - A_1}{M}$, $h_y = \frac{B_2 - A_2}{N}$. Правая часть строится как индикатор области:

$$B_{ij} = \begin{cases} F_{\text{VAL}} = 1, & (x_i, y_j) \in D, \\ 0, & (x_i, y_j) \notin D. \end{cases}$$

Метод фиктивных областей и выбор ε

Коэффициент теплопроводности задаётся по правилу

$$k(x, y) = \begin{cases} 1, & (x, y) \in D, \\ \frac{1}{\varepsilon}, & (x, y) \notin D, \end{cases} \quad \boxed{\varepsilon = (\max\{h_x, h_y\})^2}.$$

(Именно такая формула ε используется в коде: `eps = max(h1, h2)*max(h1, h2);`).

Коэффициенты схемы a_{ij} , b_{ij}

Для ячейки (i, j) вычисляются доли пересечения её граней с областью D . Длины пересечения отрезков с эллипсом:

$$x = \text{const} = x_0 : \quad y \in \left[-\sqrt{\frac{1-x_0^2}{4}}, \sqrt{\frac{1-x_0^2}{4}} \right], \quad y = \text{const} = y_0 : \quad x \in \left[-\sqrt{1-4y_0^2}, \sqrt{1-4y_0^2} \right].$$

Пусть $\ell_{ij}^{(v)}$ — часть вертикального ребра длиной h_y , лежащая в D , а $\ell_{ij}^{(h)}$ — часть горизонтального ребра длиной h_x , лежащая в D . Тогда (ровно как в функции `fictitious_regions_setup`):

$$a_{ij} = \begin{cases} 1, & \ell_{ij}^{(v)} = h_y, \\ \frac{1}{\varepsilon}, & \ell_{ij}^{(v)} = 0, \\ \frac{\ell_{ij}^{(v)}}{h_y} + \frac{1 - \frac{\ell_{ij}^{(v)}}{h_y}}{\varepsilon}, & \text{иначе,} \end{cases} \quad b_{ij} = \begin{cases} 1, & \ell_{ij}^{(h)} = h_x, \\ \frac{1}{\varepsilon}, & \ell_{ij}^{(h)} = 0, \\ \frac{\ell_{ij}^{(h)}}{h_x} + \frac{1 - \frac{\ell_{ij}^{(h)}}{h_x}}{\varepsilon}, & \text{иначе.} \end{cases}$$

Действие оператора A

Разностный оператор для внутренних узлов ($i = 1..M - 1, j = 1..N - 1$) реализован как

$$(Aw)_{ij} = -\frac{1}{h_x} \left(a_{i+1,j} \frac{w_{i+1,j} - w_{i,j}}{h_x} - a_{i,j} \frac{w_{i,j} - w_{i-1,j}}{h_x} \right) - \frac{1}{h_y} \left(b_{i,j+1} \frac{w_{i,j+1} - w_{i,j}}{h_y} - b_{i,j} \frac{w_{i,j} - w_{i,j-1}}{h_y} \right).$$

(См. функцию `apply_A`).

Диагональный предобусловливатель D^{-1}

Используется диагональное предобусловливание (см. `apply_Dinv`):

$$D_{ij} = \frac{a_{i+1,j} + a_{i,j}}{h_x^2} + \frac{b_{i,j+1} + b_{i,j}}{h_y^2}, \quad z_{ij} = (D^{-1}r)_{ij} = \frac{r_{ij}}{D_{ij}}.$$

Метод сопряжённых градиентов (PCG)

Итерационный процесс в `solve`:

$$\begin{aligned} r^{(0)} &= B, \quad z^{(0)} = D^{-1}r^{(0)}, \quad p^{(1)} = z^{(0)}, \quad \langle u, v \rangle = \sum_{i,j} u_{ij}v_{ij} h_x h_y; \\ \text{на шаге } k : \quad \alpha_k &= \frac{\langle z^{(k-1)}, r^{(k-1)} \rangle}{\langle Ap^{(k)}, p^{(k)} \rangle}, \quad w^{(k)} = w^{(k-1)} + \alpha_k p^{(k)}, \\ r^{(k)} &= r^{(k-1)} - \alpha_k A p^{(k)}, \quad z^{(k)} = D^{-1}r^{(k)}, \quad \beta_{k+1} = \frac{\langle z^{(k)}, r^{(k)} \rangle}{\langle z^{(k-1)}, r^{(k-1)} \rangle}, \\ p^{(k+1)} &= z^{(k)} + \beta_{k+1} p^{(k)}. \end{aligned}$$

$$\|w^{(k+1)} - w^{(k)}\|_E < \delta,$$

где $\|\cdot\|_E$ — евклидова норма, а δ задаётся пользователем (в тестах $\delta = 10^{-6}$).

Параллельная реализация OpenMP

Параллелизация выполняется с помощью директив:

```
#pragma omp parallel for collapse(2)
```

Они применяются в циклах:

- вычисление коэффициентов a_{ij}, b_{ij}, F_{ij} ;
- операция Aw ;
- обновление векторов r, z, p, w .

Время измеряется с помощью `omp_get_wtime()`.

Параллельная реализация MPI

Для распараллеливания разностной схемы используется библиотека MPI. Разбиение прямоугольника по условию задания (пункт 4) осуществляется по переменной x на несколько подобластей, каждая из которых обрабатывается отдельным MPI-процессом.

Двумерное разбиение области

Пусть всего P процессов и $(M - 1) \times (N - 1)$ внутренних узлов по координатам x и y . В программе сначала выбирается двумерная решётка процессов $P_x \times P_y$ (функция `choose_process_grid(P, Px, Py)`), где $P_x P_y = P$ и решётка по возможности близка к квадратной.

Затем функция `decompose_2d(M, N, Px, Py, rank, i_start, i_end, j_start, j_end)` делит множество внутренних индексов

$$i = 1, \dots, M - 1, \quad j = 1, \dots, N - 1$$

между процессами так, что каждому процессу с координатами (p_x, p_y) в решётке соответствует прямоугольный блок

$$i_{\min}^{(p)} \leq i \leq i_{\max}^{(p)}, \quad j_{\min}^{(p)} \leq j \leq j_{\max}^{(p)},$$

где

$$n_x^{(p)} = i_{\max}^{(p)} - i_{\min}^{(p)} + 1, \quad n_y^{(p)} = j_{\max}^{(p)} - j_{\min}^{(p)} + 1$$

— число внутренних узлов по x и y на данном процессе. Размеры блоков по каждому направлению отличаются не более чем на единицу:

$$|n_x^{(p)} - n_x^{(q)}| \leq 1, \quad |n_y^{(p)} - n_y^{(q)}| \leq 1$$

для любых процессов p, q . Тем самым выполняется требование задания о двумерном разбиении прямоугольника: каждый MPI-процесс отвечает за собственный подпрямоугольник сетки.

Для удобства используется локальная нумерация

$$l_i = i - i_{\min}^{(p)} + 1, \quad l_j = j - j_{\min}^{(p)} + 1,$$

где $l_i = 1, \dots, n_x^{(p)}$, $l_j = 1, \dots, n_y^{(p)}$. Дополнительно по каждому направлению вводятся *призрачные* (halo) слои с индексами $l_i = 0$, $l_i = n_x^{(p)} + 1$ и $l_j = 0$, $l_j = n_y^{(p)} + 1$, в которых хранятся значения граничных узлов, полученные от соседних процессов при обмене данными.

Локальные операции и обмен граничной информацией

Коэффициенты a_{ij}, b_{ij} и правая часть B_{ij} вычисляются локально в процедуре `fictitious_regions_se` только для принадлежащих процессу индексов $i = i_{\min}^{(p)}, \dots, i_{\max}^{(p)}$ (плюс один слой по x для корректного вычисления потоков через границы). Формулы полностью совпадают с последовательным случаем.

Действие оператора A над вектором w реализовано в функции `apply_A_local`: во внутренних узлах ($l_i = 1..n_p, j = 1..N - 1$) используется та же разностная формула, что и в последовательной версии, но все вычисления ведутся только по локальным индексам. Перед каждым вызовом `apply_A_local` выполняется обмен граничными слоями между соседними процессами (`exchange_halos()`), где используются пары вызовов `MPI_Sendrecv`. На крайних процессах граничные слои, соответствующие границе контейнера, задаются равными нулю, что соответствует граничным условиям Дирихле.

Диагональный предобусловливатель D^{-1} также строится локально (`apply_Dinv_local`): для каждого внутреннего узла процесс вычисляет

$$D_{ij} = \frac{a_{i+1,j} + a_{i,j}}{h_x^2} + \frac{b_{i,j+1} + b_{i,j}}{h_y^2}, \quad z_{ij} = \frac{r_{ij}}{D_{ij}},$$

используя только локальные данные и призрачные слои.

Параллельный метод сопряжённых градиентов

Векторные операции метода сопряжённых градиентов выполняются локально, а глобальные скалярные произведения вычисляются в два шага:

1. каждый процесс считает свою локальную сумму $\sum_{l_i,j} u_{ij}v_{ij} h_x h_y$ (функция `dot_local`);
2. все локальные суммы суммируются при помощи `MPI_Allreduce` с операцией `MPI_SUM`, формируя глобальное скалярное произведение.

Аналогично собирается глобальная норма $\|w^{(k+1)} - w^{(k)}\|_E$, которая используется в критерии остановки:

$$\|w^{(k+1)} - w^{(k)}\|_E < \delta, \quad \delta = 10^{-6}.$$

После каждого шага все процессы получают одинаковое значение нормы, поэтому решение синхронно прекращает итерации.

Вся логика метода PCG вынесена в функцию `gradient_solver_mpi`, в которой используются описанные выше локальные операции и глобальные коллективные вызовы MPI.

Гибридная реализация MPI+OpenMP

Гибридный вариант объединяет распределение данных по MPI и внутрипроцессную распараллелизацию по OpenMP. Разбиение области по переменной x , локальная нумерация

узлов, вычисление коэффициентов a_{ij}, b_{ij}, B_{ij} , обмен граничными слоями и все коллективные операции (`MPI_Allreduce`, `MPI_Sendrecv`) полностью совпадают с чисто MPI-реализацией, описанной выше.

Внутри каждого MPI-процесса наиболее трудоёмкие циклы распараллеливаются директивами OpenMP вида

```
#pragma omp parallel for
```

а при накоплении локальных норм и скалярных произведений — с использованием редукции:

```
#pragma omp parallel for reduction(+:local_sum)
```

К таким циклам относятся:

- вычисление локальных коэффициентов a_{ij}, b_{ij}, B_{ij} в процедуре `fictitious_regions_setup_local`;
- применение оператора A к вектору w (`apply_A_local`);
- применение диагонального предобуславливателя (`apply_Dinv_local`);
- обновление векторов w, r, z, p и вычисление локального вклада в норму $\|w^{(k+1)} - w^{(k)}\|_E$ в основной итерации метода.

Число нитей OpenMP в каждом процессе задаётся переменной среды `OMP_NUM_THREADS` или опцией `affinity[core(N)]` в командном файле LSF. Временные характеристики гибридного варианта измеряются на процессе с рангом 0 с помощью функции `MPI_Wtime()`, что позволяет корректно сравнивать полученное ускорение с последовательной и чисто MPI-реализацией на сетках $(M, N) = (40, 40), (400, 600)$ и $(800, 1200)$.

Реализация MPI+CUDA

В варианте MPI+CUDA сохраняется то же двумерное разбиение прямоугольника по MPI, что и в предыдущей реализации: область по внутренним индексам $i = 1, \dots, M - 1$, $j = 1, \dots, N - 1$ делится на подпрямоугольники, каждый из которых обрабатывается отдельным MPI-процессом. Локальная нумерация узлов, введение призрачных слоёв и схема разбиения области по осям x и y полностью совпадают с чисто MPI-версией (функции `choose_process_grid`, `decompose_2d`).

Коэффициенты a_{ij}, b_{ij} и правая часть B_{ij} вычисляются на CPU в процедуре `fictitious_regions_s` для внутренних узлов данного процесса (с учётом одного дополнительного слоя по каждому направлению для корректного вычисления потоков). После этого двумерные массивы a , b и B однократно «сплющиваются» в одномерные массивы `h_a`, `h_b`, `h_B` размера $(n_x^{(p)} + 2) \times (n_y^{(p)} + 2)$ с шагом `pitch = n_y^{(p)} + 2` и копируются на устройство в массивы `d_a`, `d_b`, `d_B` с помощью `cudaMemcpy`. На GPU также выделяются массивы `d_w`, `d_r`, `d_z`, `d_p`, `d_Ap` для решения в методе сопряжённых градиентов.

В текущей реализации на GPU переносится практически весь основной цикл PCG. Начальное приближение $w \equiv 0$ задаётся на устройстве ядром `zero_kernel`, после чего остаток инициализируется как $\mathbf{d_r} = \mathbf{d_B}$. Диагональный предобуславливатель D^{-1} применяется на GPU ядром `apply_Dinv_kernel`, которое для каждого внутреннего узла вычисляет

$$D_{ij} = \frac{a_{i+1,j} + a_{i,j}}{h_x^2} + \frac{b_{i,j+1} + b_{i,j}}{h_y^2}, \quad z_{ij} = \frac{r_{ij}}{D_{ij}},$$

формируя вектор $\mathbf{d_z}$. Направление p инициализируется на устройстве ядром `copy_kernel` ($\mathbf{d_p} = \mathbf{d_z}$).

Обмен призрачными слоями для вектора направления p реализуется в функции `exchange_halos_2`. Граничные значения соответствующих строк и столбцов массива $\mathbf{d_p}$ копируются на CPU с помощью вызовов `cudaMemcpy`/`cudaMemcpy2D` во временные буферы `send_left`, `send_right`, `send_up`, `send_down`, после чего межпроцессный обмен выполняется через `MPI_Sendrecv`. Полученные значения записываются обратно в призрачные слои массива $\mathbf{d_p}$ операциями `device ← host`; на глобальных границах компоненты призрачных слоёв заполняются нулями, реализуя граничное условие Дирихле $u = 0$.

Применение оператора A к вектору направления p выполняется на GPU ядром `apply_A_kernel`, которое вычисляет значения $(Ap)_{ij}$ во всех внутренних узлах данного процесса по той же разностной формуле, что и в последовательной версии. Результат записывается в массив $\mathbf{d_Ap}$ на устройстве и дальше используется полностью на GPU без копирования на CPU.

Скалярные произведения (Ap, p) и (z, r) также вычисляются на GPU. Для этого используется ядро `dot_kernel`, в котором каждый поток обходит несколько внутренних узлов, накапливая частичную сумму $\sum x_{ij}y_{ij}h_xh_y$ в массив $\mathbf{d_partial}$. После завершения работы ядра массив $\mathbf{d_partial}$ копируется на CPU, где частичные суммы суммируются в локальное значение скалярного произведения. Глобальное значение получается вызовом `MPI_Allreduce` с операцией `MPI_SUM`.

Обновление решения и остатка выполняется на GPU ядром `update_w_r_kernel`, которое для каждого внутреннего узла вычисляет

$$w_{ij}^{(k+1)} = w_{ij}^{(k)} + \alpha p_{ij}^{(k)}, \quad r_{ij}^{(k+1)} = r_{ij}^{(k)} - \alpha (Ap)_{ij}^{(k)},$$

и одновременно накапливает локальный вклад в норму $\|w^{(k+1)} - w^{(k)}\|_E^2$ во вспомогательном массиве $\mathbf{d_partial}$. После копирования $\mathbf{d_partial}$ на CPU и суммирования по процессам с помощью `MPI_Allreduce` получается глобальное значение нормы, по которому проверяется критерий остановки $\|w^{(k+1)} - w^{(k)}\|_E < \delta$.

После применения предобуславливателя D^{-1} на GPU и вычисления нового скалярного произведения $(z, r)^{(k+1)}$ по тому же шаблону определяется коэффициент $\beta^{(k)}$, и направление обновляется на устройстве ядром `update_p_kernel` по стандартной формуле метода сопряжённых градиентов:

$$p^{(k+1)} = z^{(k+1)} + \beta^{(k)} p^{(k)}.$$

Таким образом, внутри одного шага PCG выполняется следующий цикл:

1. Обмен призрачными слоями вектора направления p между MPI-процессами с помощью `exchange_halos_2d_gpu`, включающей операции `cudaMemcpy`/`cudaMemcpy2D` и `MPI_Sendrecv`.
2. Запуск ядра `apply_A_kernel` для вычисления Ap на GPU.
3. Вычисление скалярного произведения (Ap, p) ядром `dot_kernel`, копирование частичных сумм на CPU и сборка глобального знаменателя через `MPI_Allreduce`, определение шага α .
4. Обновление векторов w и r ядром `update_w_r_kernel` и вычисление нормы $\|w^{(k+1)} - w^{(k)}\|_E$ с последующей глобальной проверкой критерия остановки.
5. Применение диагонального предобуславливателя D^{-1} на GPU (`apply_Dinv_kernel`), вычисление скалярного произведения (z, r) с помощью `dot_kernel` и `MPI_Allreduce`, определение коэффициента β .
6. Обновление направления p на устройстве ядром `update_p_kernel`.

Измерение времени в MPI+CUDA-версии выполняется с помощью `MPI_Wtime()`. Для каждого процесса накапливаются отдельные счётчики: суммарное время работы всех вычислительных ядер на GPU (ядра `apply_A_kernel`, `apply_Dinv_kernel`, `update_w_r_kernel`, `update_p_kernel` и инициализирующие ядра), время копирования данных `host↔device` (включая обмен призрачных слоёв между устройством и хостом), время, затраченное на коммуникацию по MPI (обмен halo и вызовы `MPI_Allreduce`), время, связанное с применением предобуславливателя D^{-1} (T_{prec}), и время вычисления скалярных произведений (T_{dot}). В конце работы все эти величины собираются на процессе с рангом 0 при помощи `MPI_Reduce` с операцией `MPI_MAX` и выводятся вместе с общим временем инициализации, временем работы решателя T_{solver} и временем завершения программы. Это позволяет подробно проанализировать, какая доля общего ускорения связана с использованием GPU, а какая ограничивается затратами на обмен данными и коммуникацию между процессами.

Результаты расчётов(первые три этапа)

1. Последовательная версия

Размер сетки	Число итераций	Время решения
10×10	17	0.0001
20×20	31	0.0005
40×40	60	0.0034

2. OpenMP

Размер сетки	Число нитей	Число итераций	Время решения
40×40	1	60	0.005
40×40	4	60	0.004
40×40	16	60	0.009

Размер сетки ($M \times N$)	Число нитей	Число итераций	Время решения	Ускорение
400×600	2	546	3.098	1.59
400×600	4	546	1.776	2.78
400×600	8	546	1.170	4.21
400×600	16	546	0.977	5.05
800×1200	4	989	12.214	3.19
800×1200	8	989	7.554	5.15
800×1200	16	989	6.034	6.45
800×1200	32	989	6.216	6.27

3. MPI

Размер сетки	Число процессов	Число итераций	Время с MPI	Время с Openmp
40×40	1	60	0.00328	0.005
40×40	2	60	0.00186	0.004
40×40	4	60	0.00189	0.004

Размер сетки	Число процессов	Число итераций	Время решения	Ускорение
400×600	2	546	2.646	1.86
400×600	4	546	1.432	3.44
400×600	8	546	0.905	5.45
400×600	16	546	0.438	11.25
800×1200	4	989	9.689	4.02
800×1200	8	989	5.022	7.75
800×1200	16	989	2.906	13.40
800×1200	32	989	2.382	16.35

4. MPI+OpenMP

Размер сетки	Число процессов	Число нитей	Число итераций	Время с Openmp и MPI	Время с MPI
40×40	1	1	60	0.0125	0.00328
40×40	1	4	60	0.0100	0.00186
40×40	2	4	60	0.1204	0.00189

Размер сетки	Число процессов	Число нитей	Число итераций	Время решения	Ускорение
400×600	2	1	546	1.911	2.58
400×600	2	2	546	1.151	4.28
400×600	2	4	546	0.597	8.26
400×600	2	8	546	0.313	15.75
800×1200	4	1	989	7.636	5.10
800×1200	4	2	989	4.205	9.26
800×1200	4	4	989	2.138	18.21
800×1200	4	8	989	1.103	35.30

5. Выводы

По результатам численных экспериментов можно сделать следующие выводы.

Во-первых, для небольшой сетки 40×40 выигрыш от распараллеливания ограничен. Как в OpenMP, так и в MPI при увеличении числа нитей/процессов время решения меняется слабо или даже начинает расти, что связано с тем, что накладные расходы на создание нитей, синхронизацию и обмен граничными слоями сопоставимы с собственно вычислениями.

Во-вторых, при переходе к более крупным сеткам 400×600 и 800×1200 параллельные методы демонстрируют устойчивый рост ускорения. В OpenMP при увеличении числа нитей для сетки 800×1200 достигается ускорение порядка 6–7 раз по сравнению с последовательной программой. MPI-реализация на тех же сетках даёт более высокий выигрыш: до ≈ 11 раз для 400×600 и до ≈ 16 раз для 800×1200 при 16–32 процессах. Это показывает, что для задачи с относительно дорогим обменом границкой информацией 2D-разбиение области по MPI хорошо масштабируется при увеличении числа узлов.

В-третьих, гибридный вариант MPI+OpenMP позволяет заметно лучше использовать вычислительные ресурсы каждого узла. Для крупной сетки 800×1200 сочетание разбиения области по MPI и внутрипроцессного распараллеливания по OpenMP даёт ускорение до ≈ 35 раз по сравнению с последовательной версией, что существенно превосходит как чисто MPI-, так и чисто OpenMP-подход. Таким образом, гибридная схема оказывается наиболее эффективной на больших размерах задачи.

Результаты расчётов (этап 4: MPI+CUDA)

Поскольку с декабря время выполнения одной и той же программы на кластере «Полюс» заметно увеличилось, часть экспериментов была повторно проведена для корректного сравнения с Cuda-версией.

Количество процессов MPI	Количество нитей OpenMP	Количество GPU	Число точек сетки ($M \times N$)	Число итер.	Время	Ускорение
20	8	—	800×1200	989	7.64s	1.00
1	—	1	800×1200	989	0.83s	9.21
2	—	2	800×1200	989	0.64s	11.94
20	8	—	1600×2400	1858	54.33s	1.00
1	—	1	1600×2400	1858	4.85s	11.20
2	—	2	1600×2400	1858	3.19s	17.03
20	8	—	2400×3200	2449	117.39	1.00
1	—	1	2400×3200	2449	13.24s	8.87
2	—	2	2400×3200	2449	7.67s	15.31

Таблица 1: Сравнение производительности конфигураций MPI+OpenMP (20 MPI, 8 OpenMP) и MPI+CUDA

MPI proc.	GPU	Сетка ($M \times N$)	Итер.	T_{gpu}	T_{copy}	T_{mpi}	T_{prec}	T_{dot}	T_{total}
1	1	800×1200	989	0.526	0.0208	0.00194	0.1288	0.1668	0.839
2	2	800×1200	989	0.2726	0.0544	0.01372	0.0683	0.1454	0.643
1	1	1600×2400	1858	3.911	0.0518	0.00369	0.8932	0.6293	4.858
2	2	1600×2400	1858	2.333	0.1485	0.4191	0.5239	0.4480	3.195
1	1	2400×3200	2449	11.306	0.0815	0.00485	2.7939	1.4237	13.249
2	2	2400×3200	2449	6.168	0.2527	0.9145	1.3275	0.8877	7.677

Таблица 2: Декомпозиция времени выполнения для MPI+CUDA (в секундах). T_{gpu} — вычисления на GPU, T_{copy} — копирование Host \leftrightarrow Device, T_{mpi} — обмен halo, T_{prec} — CPU-часть предобуславливателя, T_{dot} — скалярные произведения.

Анализ полученных ускорений и эффективности

На основе данных, представленных в таблицах 1 и 2, можно сделать следующие выводы о производительности и эффективности реализации MPI+CUDA.

Во всех рассмотренных конфигурациях использование GPU приводит к существенному ускорению по сравнению с базовой реализацией MPI+OpenMP (20 MPI, 8 OpenMP). Уже при использовании одного GPU достигается ускорение от 8.9 до 11.2 раз, а при использовании двух GPU ускорение возрастает до 15–17 раз в зависимости от размера задачи. Это подтверждает, что вычислительная часть задачи эффективно перенесена на GPU и масштабируется с увеличением числа ускорителей.

Анализ декомпозиции времени выполнения показывает, что основная доля времени в версии MPI+CUDA приходится на вычисления на GPU (T_{gpu}), что является желаемым режимом работы для гибридных CPU–GPU приложений. При увеличении числа GPU наблюдается почти двукратное уменьшение T_{gpu} , особенно для более крупных сеток (1600×2400 и 2400×3200), что свидетельствует о хорошем распараллеливании вычислений.

В то же время при использовании нескольких MPI-процессов возрастает вклад коммуникационных накладных расходов. В частности, время обмена граничными слоями (T_{mpi}) и время копирования данных между host и device (T_{copy}) заметно увеличиваются при переходе от одной GPU к двум. Этот эффект наиболее выражен для самой крупной сетки 2400×3200 , где T_{mpi} становится сопоставимым с временем вычислений на GPU. Тем не менее, даже с учетом этих накладных расходов общее ускорение продолжает расти, что говорит о том, что выигрыш от параллельных вычислений превышает потери на коммуникацию.

Отдельно следует отметить вклад CPU-частей алгоритма, таких как предобуславливатель (T_{prec}) и вычисление скалярных произведений (T_{dot}). Их абсолютное время уменьшается при увеличении числа MPI-процессов, однако относительная доля этих операций в общем времени выполнения возрастает. Это указывает на потенциальное ограничение масштабируемости и соответствует закону Амдала: при дальнейшем увеличении числа GPU ускорение будет все в большей степени ограничиваться последовательными или сла-

бо параллелизуемыми участками кода, выполняемыми на CPU.

В целом полученные результаты подтверждают, что реализация MPI+CUDA является эффективной и обеспечивает значительное ускорение по сравнению с MPI+OpenMP. Для задач достаточно большого размера GPU используется эффективно, а наблюдаемое отклонение от идеального линейного ускорения объясняется ростом коммуникационных и CPU-накладных расходов, что полностью согласуется с теоретическими ожиданиями.

Выводы

В рамках работы была реализована и исследована гибридная версия алгоритма с использованием MPI+CUDA. Проведённые вычислительные эксперименты показали, что перенос основной вычислительной нагрузки на GPU обеспечивает существенный прирост производительности по сравнению с базовой реализацией MPI+OpenMP.

Для всех рассмотренных размеров сетки уже при использовании одного GPU достигается ускорение порядка 9–11 раз, а при использовании двух GPU ускорение возрастает до 15–17 раз. Это свидетельствует о корректной организации вычислений на GPU и хорошем распараллеливании вычислительной части задачи.

Декомпозиция времени выполнения показала, что доминирующим компонентом в реализации MPI+CUDA является время вычислений на GPU, тогда как накладные расходы, связанные с обменами по MPI и копированием данных между host и device, остаются вторичными. При увеличении числа GPU вклад коммуникационных операций возрастает, особенно для крупных сеток, однако он не приводит к деградации общей производительности.

Наблюдаемое отклонение от идеального линейного ускорения объясняется ростом коммуникационных затрат и наличием частей алгоритма, выполняемых на CPU (предобуславливатель и скалярные произведения), что соответствует закону Амдала и является ожидаемым для гибридных CPU-GPU приложений.

В целом результаты подтверждают, что реализация MPI+CUDA является эффективной и целесообразной для задач большого размера. Полученные характеристики ускорения и эффективности согласуются с теоретическими ожиданиями и подтверждают корректность выполненной параллельной реализации.

Рисунки

1. Рисунок приближенного решения на сетке 800×1200

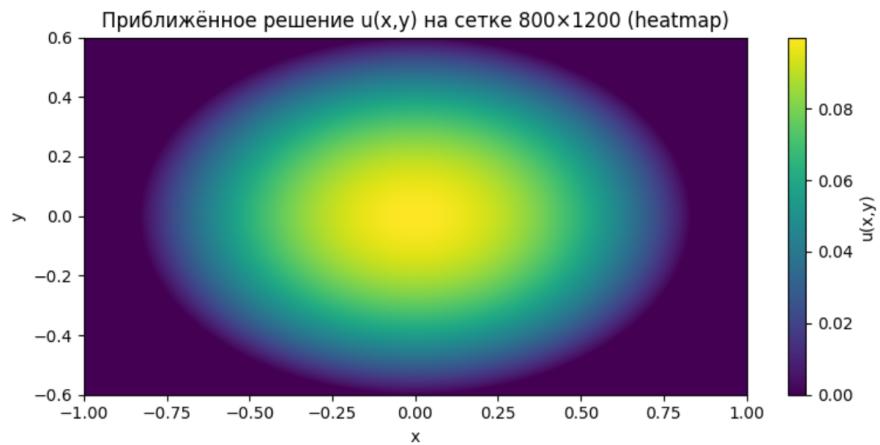


Рис. 1: Приближённое решение на сетке 800×1200 (вид сверху, 2D-представление).

Приближённое решение на сетке 800×1200

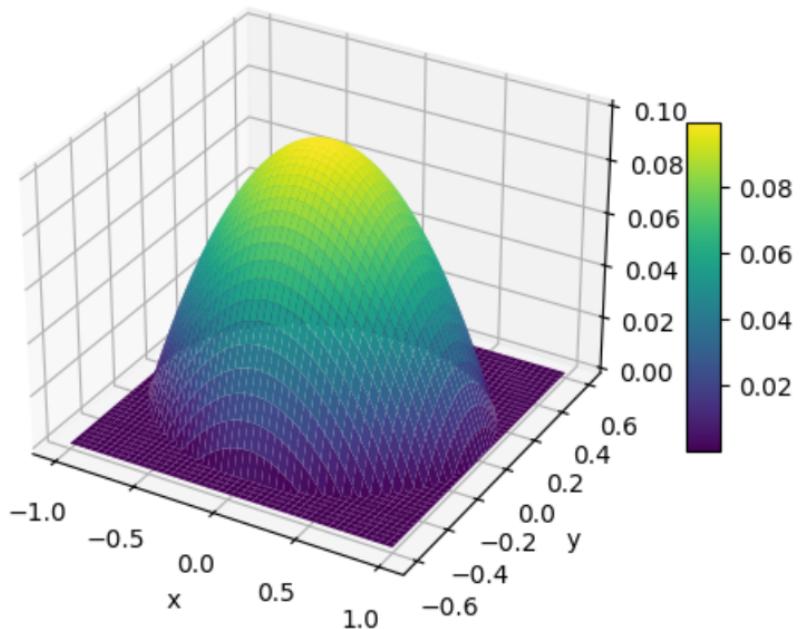


Рис. 2: Приближённое решение на сетке 800×1200 (3D-график).

2. Рисунки ускорений

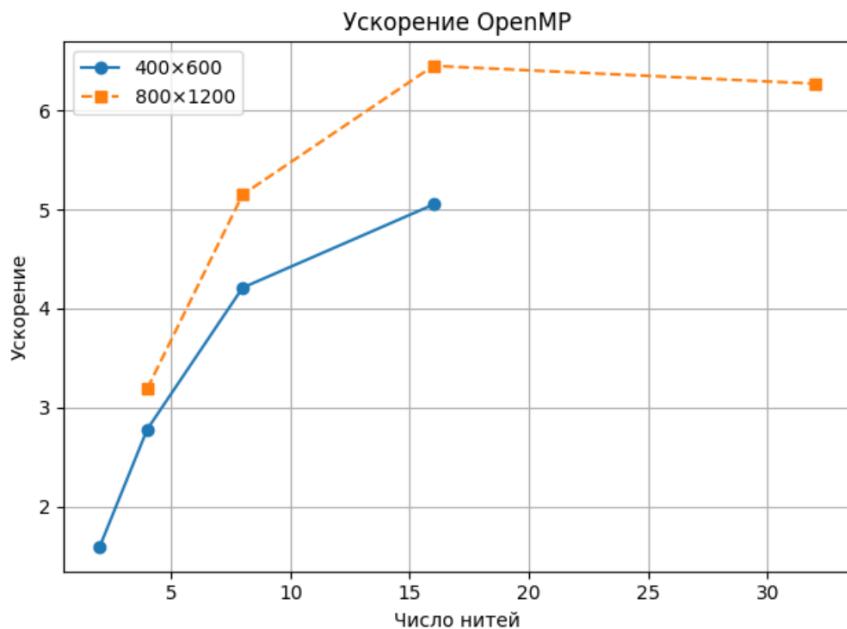


Рис. 3: Ускорение OpenMP-программы для сеток 400×600 и 800×1200 .

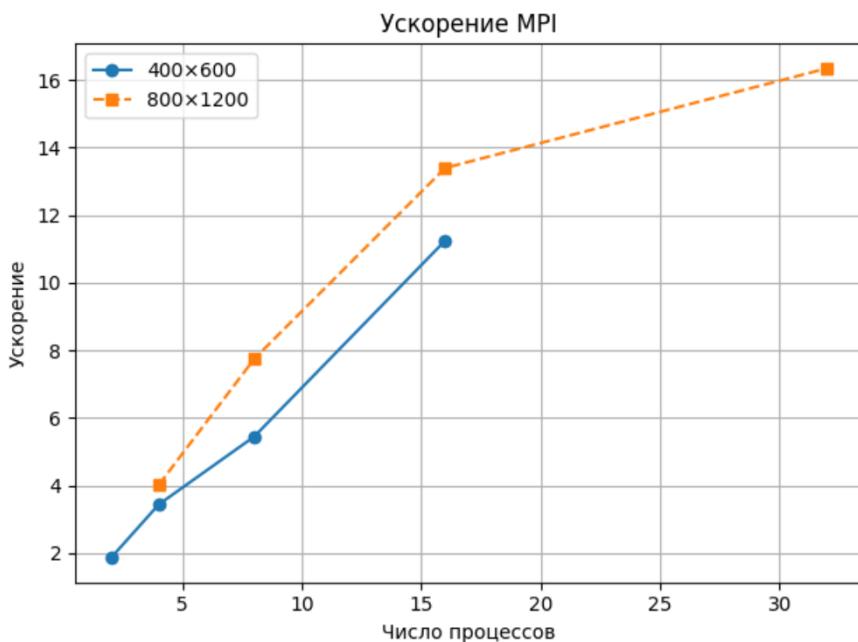


Рис. 4: Ускорение MPI-программы для сеток 400×600 и 800×1200 .

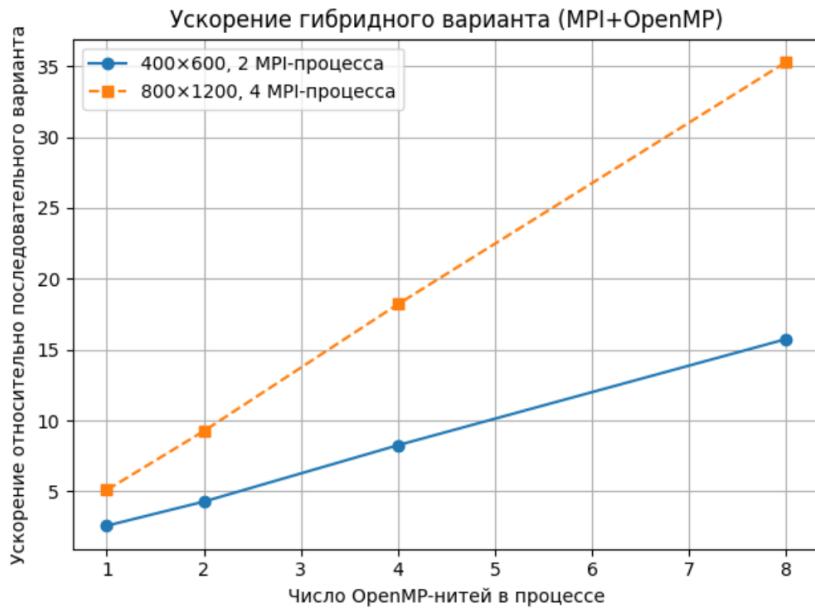


Рис. 5: Ускорение MPI и Openmp-программы для сеток 400×600 и 800×1200 .

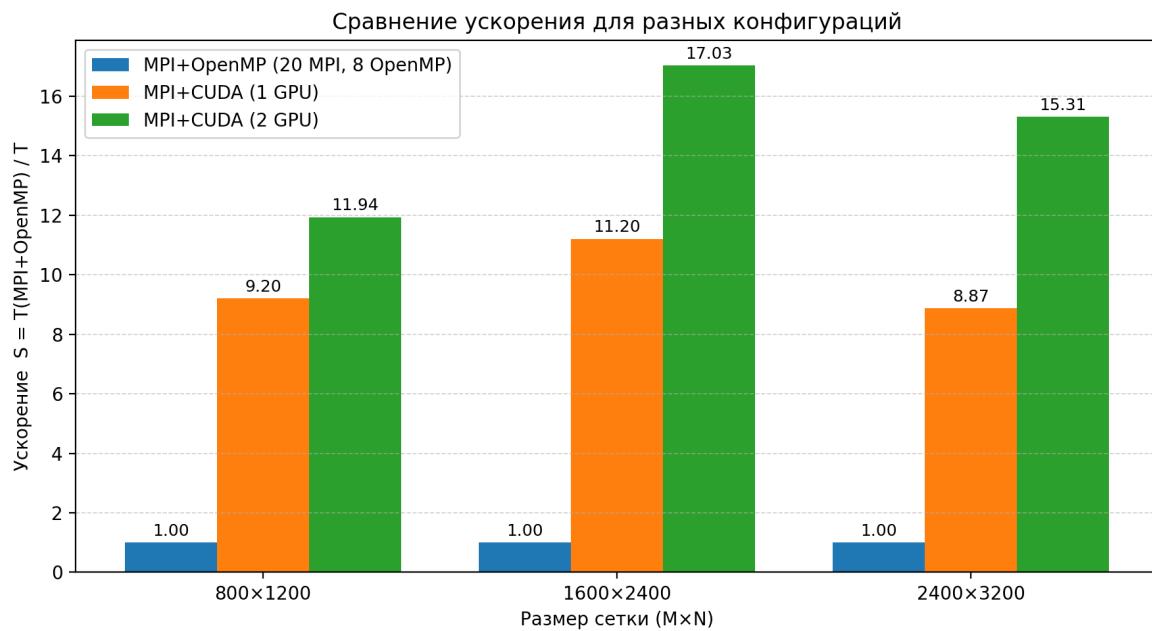


Рис. 6: Сравнение ускорения для MPI+OpenMP (20 MPI, 8 OpenMP) и MPI+CUDA (1/2 GPU) при разных размерах сетки.