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

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

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

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

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

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

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:

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

  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

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

Сетка равномерная:

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

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

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

Метод фиктивных областей и выбор ε

Коэффициент теплопроводности задаётся по правилу

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

(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:

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

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`):

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

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 предобуславливатель D^{-1}

Используется диагональное предобуславливание (см. `apply_Dinv`):

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

Метод сопряжённых градиентов (PCG)

Итерационный процесс в `solve`:

$$r^{(0)} = B, \quad z^{(0)} = D^{-1}r^{(0)}, \quad p^{(1)} = z^{(0)}, \quad \langle u, v\rangle = \sum_{i,j} u_{ij}v_{ij}\,h_x h_y;$$

$$\text{на шаге } k: \quad \alpha_k = \frac{\langle z^{(k-1)}, r^{(k-1)}\rangle}{\langle Ap^{(k)}, p^{(k)}\rangle}, \qquad w^{(k)} = w^{(k-1)} + \alpha_k p^{(k)},$$

$$r^{(k)} = r^{(k-1)} - \alpha_k A p^{(k)}, \qquad z^{(k)} = D^{-1}r^{(k)}, \qquad \beta_{k+1} = \frac{\langle z^{(k)}, r^{(k)}\rangle}{\langle z^{(k-1)}, r^{(k-1)}\rangle},$$

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

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

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

$$i = 1, \ldots, M-1, \qquad j = 1, \ldots, N-1$$

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

$$i_{\min}^{(p)} \le i \le i_{\max}^{(p)}, \qquad j_{\min}^{(p)} \le j \le j_{\max}^{(p)},$$

где

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

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

$$|n_x^{(p)} - n_x^{(q)}| \le 1, \qquad |n_y^{(p)} - n_y^{(q)}| \le 1$$

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 $l_i = 1, \ldots, n_x^{(p)}$, $l_j = 1, \ldots, n_y^{(p)}$. 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 $i = i_{\min}^{(p)}, \ldots, i_{\max}^{(p)}$ (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 предобуславливатель D^{-1} также строится локально (`apply_Dinv_local`): для каждого внутреннего узла процесс вычисляет

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

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:

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

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_loca`
- 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 $i = 1, \ldots, M-1$, $j = 1, \ldots, N-1$ делится на подпрямоугольники, каждый из которых обрабатывается отдельным MPI-процессом. Локальная нумерация узлов, введение призрачных слоёв и организация обмена граничной информации с соседями (`exchange_halos_2d`) полностью совпадают с чисто MPI-версией.

Коэффициенты a_{ij}, b_{ij} и правая часть B_{ij} вычисляются на CPU в процедуре

`fictitious_regions_setup_local` для внутренних узлов данного процесса (с учётом одного дополнительного слоя по каждому направлению для корректного вычисления потоков). После этого двумерные массивы a и b однократно «сплющиваются» в одномерные массивы `h_a`, `h_b` размера $(n_x^{(p)} + 2) \times (n_y^{(p)} + 2)$ с шагом `pitch` $= n_y^{(p)} + 2$ и копируются на GPU в массивы `d_a`, `d_b` с помощью `cudaMemcpy`.

Основные вычислительно затратные операции метода сопряжённых градиентов переносятся на GPU:

- применение оператора A к вектору p: ядро `apply_A_kernel` вычисляет значение $(Ap)_{ij}$ во всех внутренних узлах данного процесса по той же разностной формуле, что и в последовательном случае. Массив `d_p` хранит локальные значения направления p (включая призрачные слои), результат записывается в `d_Ap`;

- применение диагонального предобуславливателя D^{-1}: ядро `apply_Dinv_kernel` для каждого внутреннего узла вычисляет

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

  используя локальные данные `d_a`, `d_b` и остаток `d_r`;

- скалярные произведения вида (z, r), (Ap, p): для них используется специальное ядро `dot_kernel`, которое проходит по всем внутренним узлам данного процесса, накапливая сумму $\sum u_{ij} v_{ij} h_x h_y$ в массиве `d_partial` без использования разделяемой памяти и атомарных операций. Затем массив `d_partial` копируется на CPU (`h_partial`), и локальная сумма вычисляется уже на хосте.

На каждом шаге метода PCG выполняется следующий цикл:

1. На CPU обновляются призрачные слои вектора направления p при помощи `exchange_halos_2d`, после чего локальный двумерный массив `p` сплющивается в одномерный `h_p` и копируется на устройство в `d_p` (операции host→device учитываются отдельно во времени копирования).

2. Запускается ядро `apply_A_kernel`, вычисляющее `d_Ap`; после синхронизации (`cudaDeviceSynchronize`) массив `d_Ap` копируется обратно на CPU в `h_Ap`, откуда значения записываются в двумерный массив `Ap` для дальнейшего обновления w и r.

3. Скалярное произведение (Ap, p) вычисляется на GPU с помощью `dot_kernel`; полученная локальная сумма передаётся на CPU и далее используется в глобальном `MPI_Allreduce` для получения знаменателя.

4. Векторы w и r обновляются на CPU, одновременно накапливается локальный вклад в норму $\|w^{(k+1)} - w^{(k)}\|_E$. Эта норма затем собирается по всем процессам при помощи `MPI_Allreduce` и используется в критерии остановки $\|w^{(k+1)} - w^{(k)}\|_E < \delta$.

5. Остаток r переносится на устройство (`d_r`), запускается ядро `apply_Dinv_kernel`, рассчитывающее `d_z`, после чего результат копируется обратно в `h_z` и двумерный массив `z`. Скалярное произведение (z, r) вновь вычисляется на GPU через `dot_kernel` и собирается по всем процессам при помощи `MPI_Allreduce`, формируя величину $(z, r)^{(k)}$.

6. Направление p обновляется на CPU по стандартной формуле метода сопряжённых градиентов $p^{(k+1)} = z^{(k+1)} + \beta^{(k)} p^{(k)}$.

Измерение времени в MPI+CUDA-версии выполняется с помощью `MPI_Wtime()`. Для каждого процесса накапливаются отдельные счётчики: суммарное время работы GPU-ядер (операции Ap и $D^{-1}r$), время копирования данных host↔device, время обмена граничных слоёв по MPI, время CPU-части предобуславливателя (распаковка `z` в двумерный массив) и время вычисления скалярных произведений (ядро `dot_kernel` плюс перенос частичных сумм). В конце работы все эти величины собираются на процессе с рангом 0 при помощи `MPI_Reduce` с операцией `MPI_MAX` и выводятся в отчёт вместе с общим временем инициализации, временем работы решателя 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 граничной 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.

Таблица 1: Сравнение времени и ускорения для сетки 400×600. Здесь P — число задействованных вычислительных единиц: потоков OpenMP (для OMP) или процессов MPI (для MPI и MPI+GPU; в варианте MPI+GPU каждому GPU соответствует один MPI-процесс); T — время решения, $S = T_{\text{seq}}/T$ — ускорение относительно последовательной программы; индексы OMP, MPI, MPI+GPU обозначают соответствующую реализацию.

p	$T_{\mathbf{seq}}$	$T_{\mathbf{OMP}}$	$S_{\mathbf{OMP}}$	$T_{\mathbf{MPI}}$	$S_{\mathbf{MPI}}$	$T_{\mathbf{MPI+GPU}}$	$S_{\mathbf{MPI+GPU}}$
1	5.81	5.91	-	5.20	1.12	1.23	4.72
2		3.11	1.87	2.70	2.15	0.69	8.42
4		1.79	3.25	1.53	3.80	0.37	15.7

Таблица 2: Сравнение времени и ускорения для сетки 800×1200. Здесь P — число задействованных вычислительных единиц: потоков OpenMP (для OMP) или процессов MPI (для MPI и MPI+GPU; в варианте MPI+GPU каждому GPU соответствует один MPI-процесс); T — время решения, $S = T_{\text{seq}}/T$ — ускорение относительно последовательной программы; индексы OMP, MPI, MPI+GPU обозначают соответствующую реализацию.

p	$T_{\mathbf{seq}}$	$T_{\mathbf{OMP}}$	$S_{\mathbf{OMP}}$	$T_{\mathbf{MPI}}$	$S_{\mathbf{MPI}}$	$T_{\mathbf{MPI+GPU}}$	$S_{\mathbf{MPI+GPU}}$
1	46.07	48.87	-	40.82	1.13	5.16	8.93
2		22.45	2.05	20.43	2.26	2.71	17.00
4		12.25	3.76	17.18	2.68	1.31	35.17

Таблица 3: Сравнение времени и ускорения для сетки 1600×2400. Здесь P — число задействованных вычислительных единиц: потоков OpenMP (для OMP) или процессов MPI (для MPI и MPI+GPU; в варианте MPI+GPU каждому GPU соответствует один MPI-процесс); T — время решения, $S = T_{\text{seq}}/T$ — ускорение относительно последовательной программы; индексы OMP, MPI, MPI+GPU обозначают соответствующую реализацию.

p	$T_{\mathbf{seq}}$	$T_{\mathbf{OMP}}$	$S_{\mathbf{OMP}}$	$T_{\mathbf{MPI}}$	$S_{\mathbf{MPI}}$	$T_{\mathbf{MPI+GPU}}$	$S_{\mathbf{MPI+GPU}}$
1	365.37	320.03	1.14	294.73	1.24	28.32	12.90
2		168.13	2.17	151.63	2.41	15.04	24.29
4		93.78	3.90	72.76	5.02	7.39	49.44

Таблица 4: Разложение времени для варианта MPI+CUDA на сетке 400×600. Здесь p — число MPI-процессов (одновременно число задействованных GPU); T_{solver} — суммарное время работы решателя; T_{GPU} — время выполнения вычислительных ядер на GPU; T_{copy} — время копирования данных между host и device; T_{MPI} — затраты на обмен граничными слоями (halo) по MPI; T_{prec} — время, связанное с применением предобуславливателя D^{-1}; T_{dot} — время вычисления скалярных произведений. Все времена приведены в секундах.

p	$T_{\mathbf{solver}}$	$T_{\mathbf{GPU}}$	$T_{\mathbf{copy}}$	$T_{\mathbf{MPI}}$	$T_{\mathbf{prec}}$	$T_{\mathbf{dot}}$
1	1.227	0.025	0.090	0.002	0.283	0.083
2	0.690	0.020	0.058	0.004	0.142	0.034
4	0.372	0.012	0.025	0.005	0.071	0.016

Таблица 5: Разложение времени для варианта MPI+CUDA на сетке 800×1200. Здесь p — число MPI-процессов (одновременно число задействованных GPU); T_{solver} — суммарное время работы решателя; T_{GPU} — время выполнения вычислительных ядер на GPU; T_{copy} — время копирования данных между host и device; T_{MPI} — затраты на обмен граничными слоями (halo) по MPI; T_{prec} — время, связанное с применением предобуславливателя D^{-1}; T_{dot} — время вычисления скалярных произведений. Все времена приведены в секундах.

p	$T_{\textbf{solver}}$	$T_{\textbf{GPU}}$	$T_{\textbf{copy}}$	$T_{\textbf{MPI}}$	$T_{\textbf{prec}}$	$T_{\textbf{dot}}$
1	5.158	0.062	0.360	0.003	1.169	0.490
2	2.709	0.045	0.300	0.009	0.585	0.201
4	1.307	0.031	0.102	0.010	0.292	0.086

Таблица 6: Разложение времени для варианта MPI+CUDA на сетке 1600×2400. Здесь p — число MPI-процессов (одновременно число задействованных GPU); T_{solver} — суммарное время работы решателя; T_{GPU} — время выполнения вычислительных ядер на GPU; T_{copy} — время копирования данных между host и device; T_{MPI} — затраты на обмен граничными слоями (halo) по MPI; T_{prec} — время, связанное с применением предобуславливателя D^{-1}; T_{dot} — время вычисления скалярных произведений. Все времена приведены в секундах.

p	$T_{\textbf{solver}}$	$T_{\textbf{GPU}}$	$T_{\textbf{copy}}$	$T_{\textbf{MPI}}$	$T_{\textbf{prec}}$	$T_{\textbf{dot}}$
1	28.324	0.316	2.140	0.010	6.712	2.107
2	15.042	0.169	1.698	0.028	3.366	1.093
4	7.394	0.111	0.517	0.041	1.665	0.701

Анализ полученных ускорений и эффективности

Анализ полученных ускорений и эффективности

Сравнение трёх реализаций (OpenMP, MPI и MPI+CUDA) на разных размерах сетки показывает, что наибольший выигрыш даёт использование GPU в сочетании с MPI. Для сетки 400×600 максимальное ускорение варианта MPI+GPU достигает $S_{\text{MPI+GPU}} \approx 15.7$ при $p = 4$, тогда как OpenMP даёт примерно $S_{\text{OMP}} \approx 3.25$, а чистый MPI — $S_{\text{MPI}} \approx 3.8$. При увеличении размера задачи до 800×1200 ускорение MPI+GPU возрастает до ≈ 35, а для самой крупной сетки 1600×2400 достигает уже $S_{\text{MPI+GPU}} \approx 49.4$ при тех же четырёх вычислительных устройствах. Таким образом, по мере роста размерности задачи эффективность использования GPU заметно улучшается: затраты на инициализацию и обмены «размываются» на большем числе вычислительных операций, и параллелизм графического ускорителя используется более полно.

Если рассматривать масштабирование по числу процессов p, то для варианта MPI+CUDA наблюдается почти линейное уменьшение времени решения. Так, для сетки 800×1200 время решателя уменьшается с $T_{\text{solver}} \approx 5.16\,\text{с}$ при $p = 1$ до $2.71\,\text{с}$ при $p = 2$ и $1.31\,\text{с}$ при $p = 4$; для сетки 1600×2400 — с $28.3\,\text{с}$ до $15.0\,\text{с}$ и далее до $7.39\,\text{с}$. Фактически, удвоение числа процессов почти в два раза сокращает время решения, что говорит о высокой параллельной эффективности схемы разбиения области и обмена граничными слоями. Для CPU-вариантов (OMP и MPI без GPU) ускорение тоже растёт с p, но насыщается гораздо

раньше: при $p = 4$ OpenMP даёт порядка 3.9 раз, а MPI — около 5 раз, что указывает на ограничение уже со стороны пропускной способности памяти и менее выгодного соотношения объёма вычислений к объёму обменов.

Таблицы с разложением времени для варианта MPI+CUDA позволяют увидеть, какие компоненты становятся «узкими местами». Для всех сеток доля времени чистых вычислений на GPU (T_{GPU}) относительно невелика (десятые доли секунды даже для самой крупной сетки), в то время как значительную часть T_{solver} занимают: применение предобуславливателя D^{-1} (T_{prec}), вычисление скалярных произведений (T_{dot}) и копирование данных между host и device (T_{copy}). Например, для сетки 1600×2400, $p = 4$, при общем времени решателя $\approx 7.39\,\text{с}$ на ядра GPU приходится всего $\approx 0.11\,\text{с}$, в то время как суммарно предобуславливатель, пересылки и скалярные произведения занимают порядка трёх секунд. Время MPI-обменов (T_{MPI}) остаётся сравнительно небольшим для всех рассмотренных случаев, что подтверждает корректность выбора 2D-разбиения области и схемы обмена halo-слоями.

Из этих наблюдений следует, что основной резерв дальнейшего ускорения заключается не столько в оптимизации самого матрично-векторного умножения на GPU, сколько в уменьшении объёма копирований host–device и в переносе как можно большей части операций предобуславливателя и редукций (скалярных произведений, норм) полностью на устройство с последующим сокращением числа синхронизаций с CPU. Тем не менее уже реализованная версия MPI+CUDA демонстрирует значительное ускорение по сравнению с последовательным алгоритмом (до $\sim 50\times$ для крупной сетки) и заметно превосходит варианты только с OpenMP или только с MPI по мере увеличения размерности задачи.

Рисунки

1. Рисунок приближенного решения на сетке 800×1200

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

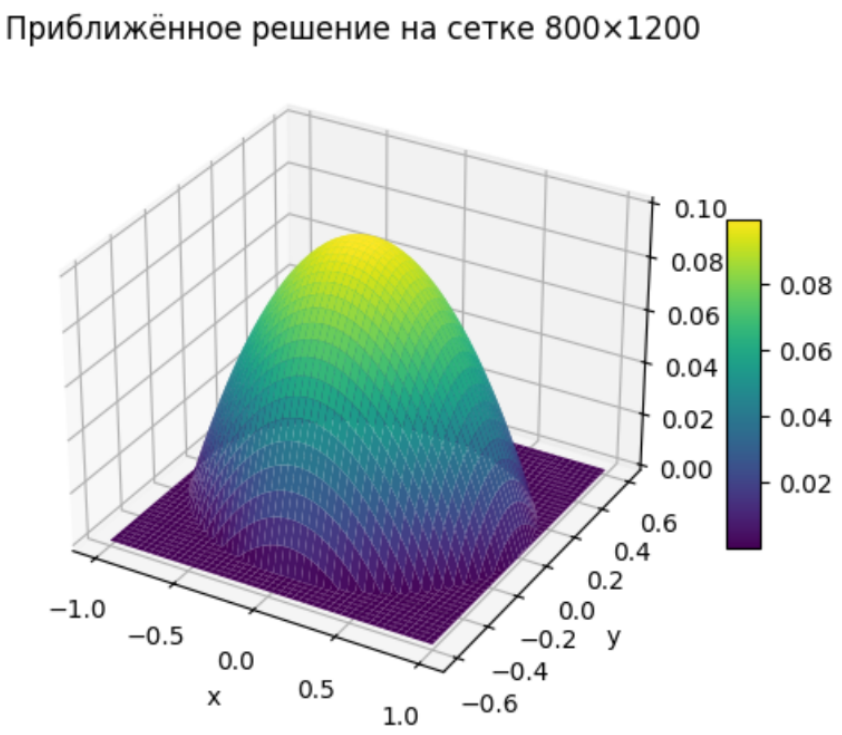

Рис. 2: Приближённое решение на сетке 800 × 1200 (3D-график).

2. Рисунки ускорений

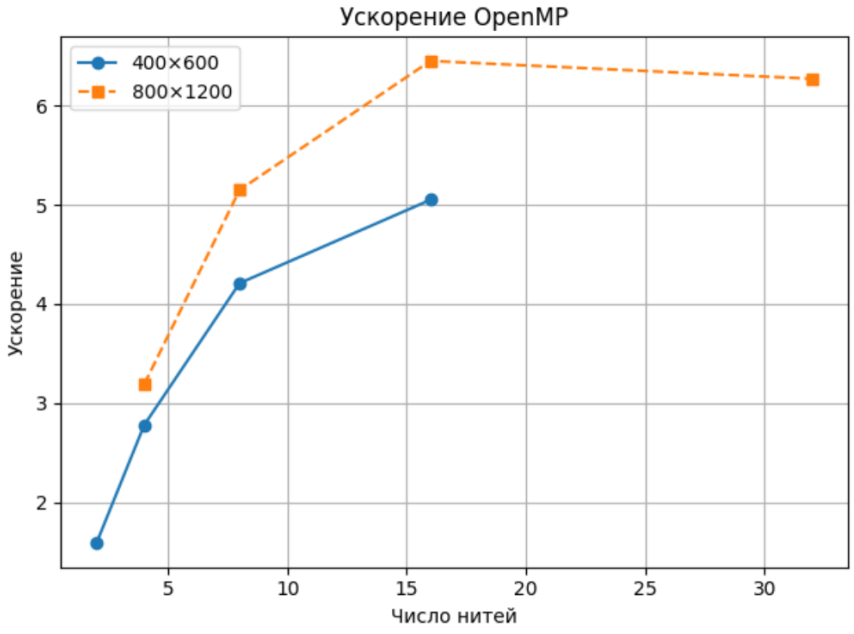

Рис. 3: Ускорение OpenMP-программы для сеток 400 × 600 и 800 × 1200.

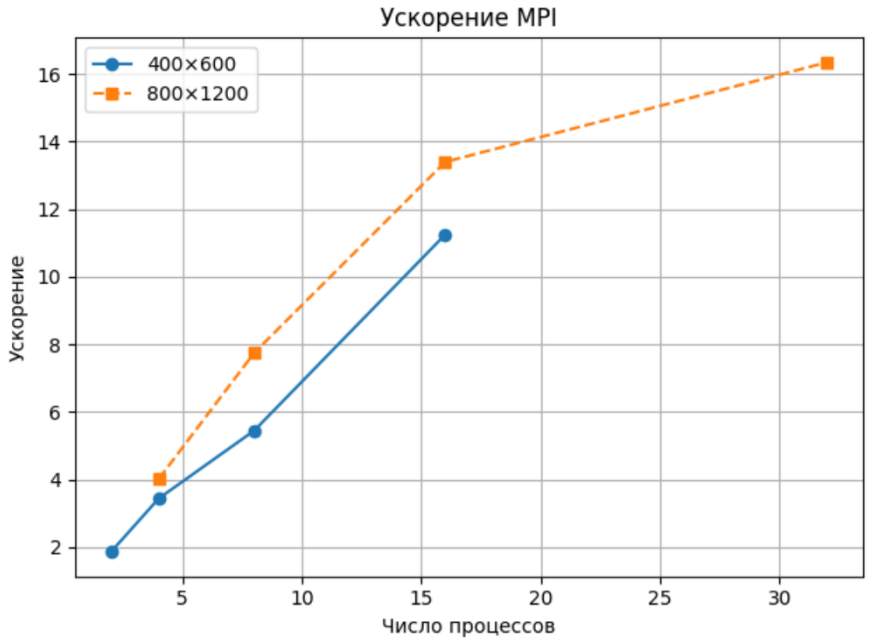

Рис. 4: Ускорение MPI-программы для сеток 400×600 и 800×1200.

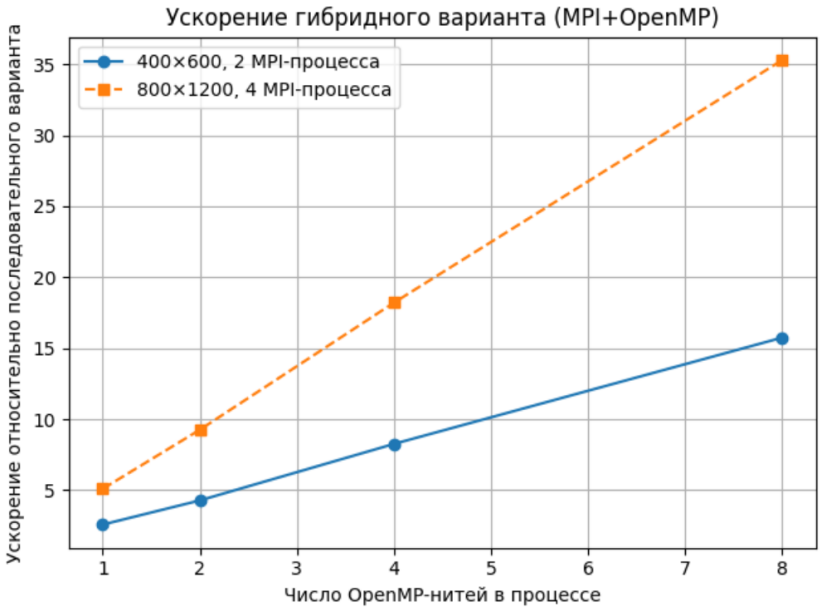

Рис. 5: Ускорение MPI и Openmp-программы для сеток 400×600 и 800×1200.

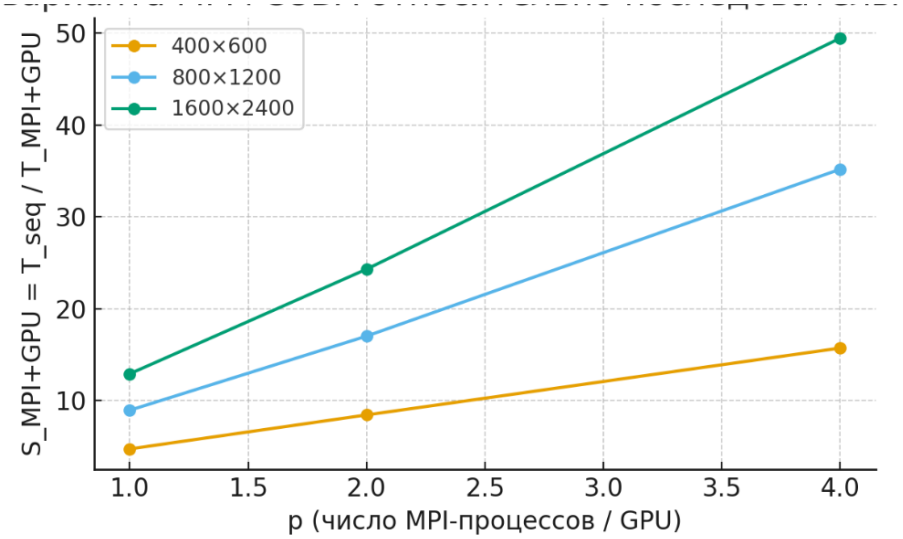

Рис. 6: Ускорение MPI+CUDA для сеток 400×600 , 800×1200 и 1600×2400.