Отчет по практикуму №2 и 3

Численное решение задачи Дирихле для уравнения Пуассона в прямоугольной области

Шпилевой В.Д. 621 гр.

Оглавление

Математическая постановка задачи

Задача Дирихле — вид задач, появляющийся при решении дифференциальных уравнений в частных производных второго порядка.

Уравнение Пуассона — эллиптическое дифференциальное уравнение в частных производных, которое описывает некоторые физические явления (поле давления, электростатическое поле и др.).

Физической интерпретацией задачи Дирихле является поиск некоторой физической величины на заданной границе.

В данной работе задача Дирихле рассматривается исключительно с точки зрения математической системы уравнений, в абстракции от физической трактовки.

Формальный вид системы:

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

$$-\Delta u = F(x, y), \quad A_1 < x < A_2,\ B_1 < y < B_2$$

$$u(x, y) = \varphi(x, y)$$

$$\Delta u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$$

Функции $F(x_i, y_i)$ и $\varphi(x_i, y_i)$ для каждой конкретной задачи известны. Но даже при знании этих функций решить задачу, получив точный ответ, иногда нельзя, да и не нужно. Чаще достаточно получить численное решение задачи с некоторой заданной точностью.

Плюс численного решения не только в том, что оно является хоть каким-то решением, но и в том, что алгоритм получения численного решения относительно легко программируется через разностную схему.

Разностная схема решения задачи

Прямоугольник, на котором требуется решить задачу, разбивается на конечное число точек, образующих матрицу (или сетку) пар значений координат X и Y:

$$\bar{\omega}_h = \{(x_i, y_j),\ i = 0, 1, 2, \ldots, N_1,\ j = 0, 1, 2, \ldots, N_2\},$$
$$A_1 = x_0 < x_1 < x_2 < \ldots < x_{N_1} = A_2$$
$$B_1 = y_0 < y_1 < y_2 < \ldots < y_{N_2} = B_2$$

После чего решается система линейных уравнений:

$$-\Delta_h p_{ij} = F(x_i, y_j), \quad (x_i, y_j) \in \omega_h,$$
$$p_{ij} = \varphi(x_i, y_j), \quad (x_i, y_j) \in \gamma_h.$$

Где матрица P — искомое численное решение, ω_h - внутренние точки сетки, а γ_h - граничные точки сетки.

Левая часть первого уравнения — пятиточечный оператор Лапласа, определенный во внутренних точках:

$$-\Delta_h p_{ij} = \frac{1}{\hbar_i^{(1)}}\left(\frac{p_{ij} - p_{i-1\,j}}{h_{i-1}^{(1)}} - \frac{p_{i+1\,j} - p_{ij}}{h_i^{(1)}}\right) + \frac{1}{\hbar_j^{(2)}}\left(\frac{p_{ij} - p_{i\,j-1}}{h_{j-1}^{(2)}} - \frac{p_{i\,j+1} - p_{ij}}{h_j^{(2)}}\right)$$

$$\hbar_i^{(1)} = 0.5\left(h_i^{(1)} + h_{i-1}^{(1)}\right),\ \hbar_j^{(2)} = 0.5\left(h_j^{(2)} + h_{j-1}^{(2)}\right)$$

$$h_i^{(1)} = x_{i+1} - x_i,\ i = 0, 1, 2, \ldots, N_1 - 1,\ h_j^{(2)} = y_{j+1} - y_j,\ j = 0, 1, 2, \ldots, N_2 - 1$$

Система уравнений с неизвестным P решается итеративно до тех пор, пока не будет получена заданная заранее точность решения. Решение выполняется методом сопряженных градиентов. Первые два шага этого метода аналогичны методу скорейшего спуска.

Во всех формулах, следующих далее, скалярное произведение и норма полагаются такими:

$$(u,v) = \sum_{i=1}^{N_1-1}\sum_{j=1}^{N_2-1} \hbar_i^{(1)}\hbar_j^{(2)} u_{ij}v_{ij}, \quad \|u\| = \sqrt{(u,u)},$$

Метод скорейшего спуска

Первый шаг метода — инициализация P. Граничные точки P известны из постановки задачи и заполняются значениями функции $\phi(x_i, y_i)$. Внутренние точки могут быть любыми числами. В данной работе они заполнены нулями.

$$p_{ij}^{(0)} = \varphi(x_i, y_j), \quad (x_i, y_j) \in \gamma_h,$$

Каждая следующая итерация вычисляется по формулам:

$$p_{ij}^{(k+1)} = p_{ij}^{(k)} - \tau_{k+1} r_{ij}^{(k)},$$

$$\tau_{k+1} = \frac{\left(r^{(k)}, r^{(k)}\right)}{\left(-\Delta_h r^{(k)}, r^{(k)}\right)}.$$

$$r_{ij}^{(k)} = -\Delta_h p_{ij}^{(k)} - F_(x_i, y_j), \quad (x_i, y_j) \in \omega_h,$$
$$r_{ij}^{(k)} = 0, \quad (x_i, y_j) \in \gamma_h.$$

Метод сопряженных градиентов

Первые два шага выполняются по методу скорейшего спуска. Далее по формулам:

$$p_{ij}^{(k+1)} = p_{ij}^{(k)} - \tau_{k+1} g_{ij}^{(k)}, \quad k = 1, 2, \ldots$$

$$\tau_{k+1} = \frac{\left(r^{(k)}, g^{(k)}\right)}{\left(-\Delta_h g^{(k)}, g^{(k)}\right)}, \qquad \alpha_k = \frac{\left(-\Delta_h r^{(k)}, g^{(k-1)}\right)}{\left(-\Delta_h g^{(k-1)}, g^{(k-1)}\right)}.$$

$$g_{ij}^{(k)} = r_{ij}^{(k)} - \alpha_k g_{ij}^{(k-1)}, \quad k = 1, 2, \ldots,$$
$$g_{ij}^{(0)} = r_{ij}^{(0)},$$

.

Итерирование происходит до тех пор, пока не выполнится неравенство:

$$\|p^{(n)} - p^{(n-1)}\| < \varepsilon,$$

Здесь ε - заранее зафиксированное число.

Вариант задания

В данной работе запрограммировано решение задачи со следующими параметрами:

$$F(x,y) = 4(2 - 3x^2 - 3y^2), \quad \varphi(x,y) = (1-x^2)^2 + (1-y^2)^2,$$

$$\Pi = [0,1] \times [0,1].$$

Сетка равномерная — то есть разбиение равномерно по оси X и равномерно по оси Y. При этом реализация выполнена без привязки к «квадратности» сетки. То есть область может быть по желанию разбита по оси X не так же, как по оси Y.

Число ε для остановки итерационного процесса равно 0.0001.

Погрешность определяется по формуле:

$$\psi = \|u(x_i, y_j) - p_{ij}\|,$$

Описание реализации

Реализация выполнена в двух вариантах: MPI + опциональный OMP и MPI + CUDA. Код: https://github.com/Gerold103/supercomputers2/ Ветки master и hw3-cuda. Рассмотрим детали каждой реализации отдельно, но сначала немного об общем виде алгоритма.

Общий алгоритм обеих реализаций устроен таким образом. После старта программы определяется размер MPI коммуникатора и индекс текущего процесса в нем. Из аргументов командной строки берется количество точек сетки по X и количество точек сетки по Y.

Из этой информации определяется, каким прямоугольником точек владеет текущий процесс. Внутри этого прямоугольника процесс выполняет все свои вычисления.

Но есть проблема, что не всегда область можно поровну разделить между всеми процессами. На сетке одного и того же размера можно так подбирать количество процессов, что некоторые точки останутся бесхозными — это из-за остатков от делений при вычислении того, какие прямоугольники получит каждый процесс. Реализованный код решает эту проблему следующим образом — остатки точек распределяются по одной точке между некоторыми процессами (процессы верхнего левого угла, если представить себе процессы, как прямоугольники, покрывающие область решения задачи).

Разбиение выполняется так, чтобы удовлетворить правило, что стороны прямоугольников не должны отличаться более, чем в два раза. Если не удается разбить область с заданным количеством процессов и точек с соблюдением этого правила, то программа выдаст ошибку.

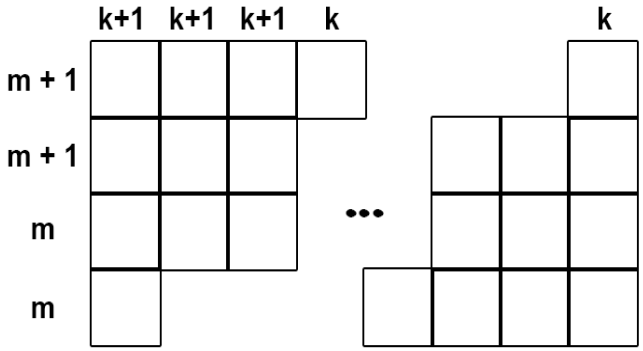

Рисунок 1. Разбиение области на прямоугольники.

Однако оператор Лапласа не может быть вычислен процессом самостоятельно без участия соседей. Поэтому на каждой итерации каждый процесс выполняет один обмен границами своей части матрицы R с соседями (обмен границами P и G не нужен, так как они вычисляются по границам R). При этом процесс обмена является асинхронным. То есть, послав свои границы

соседям, процесс вместо блокировки выполняет вычисления, нужные на следующем шаге, после чего получает границы от соседей.

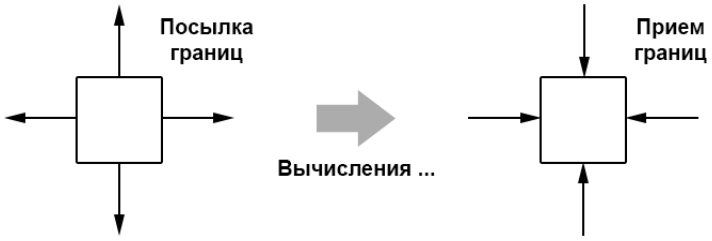

Рисунок 2. Асинхронность обмена границами.

Скалярное произведение, как и оператор Лапласа, не может быть вычислено процессом полностью автономно, поэтому на каждом шаге алгоритма выполняется два глобальных вычисления скалярных произведений.

Здесь можно заметить кажущуюся неточность — ведь на каждом шаге алгоритма вычисляется 4 скалярных произведения, а не 2. Но эти 4 скалярных произведения считаются в двух парах, и каждое из скалярных произведений в паре можно посчитать независимо от второго. Поэтому каждый процесс считает у себя сразу два локальных скалярных произведения, и затем за одну рассылку выполняется вычисление сразу двух полноценных скалярных произведений. Это проделывается дважды на каждой итерации — один раз для вычисления τ и второй раз для α .

Реализация MPI + OMP

Программа следует описанному выше алгоритму, написана полностью на языке C и состоит из двух модулей: модуль расчетов, локальных для каждого процесса, и модуль взаимодействия с другими процессами.

Модуль взаимодействия между процессами первым начинает работу после старта программы. Он определяет размер MPI коммуникатора, индекс текущего процесса, наличие соседних процессов, их индексы.

Модуль вычислений создает матрицы в виде массивов чисел с плавающей точкой. То есть двумерные матрицы «расплющиваются» в массивы. Это ускоряет обращения к памяти, так как меньше обращений по указателям (вся матрица по одному адресу, а не каждая строка по своему собственному адресу), и также уменьшает фрагментацию памяти и упрощает ее удаление в конце работы программы.

OMP используется для распараллеливания вычислений. Была произведена попытка параллелить отправку сообщений, но оказалось, что MPICH на BlueGene не работает, если слать сообщения из нескольких потоков.

Реализация MPI + CUDA

Реализация следует алгоритму, описанному в начале главы. Но при этом кодовая база модуля вычислений почти полностью переделана в третий модуль — модуль работы с GPU. Этот модуль написан на C++ и предоставляет C API для модуля вычислений.

Модуль GPU реализует API для работы с матрицами в памяти устройства:

- функция создания матриц в памяти устройства;
- функция применения к матрице оператора Лапласа;
- функция линейной комбинации двух матриц;
- функция обновления матриц P, G, R;
- другие вспомогательные функции и методы.

Все детали алгоритма реализованы так же, как на процессоре в MPI + OMP реализации, но отдельного внимания заслуживает то, как хранятся, обновляются и пересылаются между устройством и процессором границы матриц.

Дело в том, что арифметические операции на GPU так быстры, что новым «бутылочным горлышком» реализации становится не эффективность вычислений, а то, как их результаты эффективно передать процессору.

Кроме того, обращения к GPU даже для вызова функции обработки данных прямо на GPU без пересылки на процессор, тоже занимают время, и потому нужно как можно сильнее сократить количество обращений.

Рассмотрим по отдельности, как решены эти проблемы.

Хранение матрицы и ее границ

В процессе тестирования первоначальной версии реализации было выявлено, что одно из самых медленных мест — обработка границ матриц.

Дело в том, что у процессов, владеющих только внутренними точками сетки, по 4 соседа. Из-за этого хранение границ каждого соседа для матриц R, G и P в отдельных массивах и их отдельная обработка стали невозможны — с ростом числа процессов скорость лишь падала. Поэтому реализация на GPU использует особый метод хранения матриц — в едином массиве хранится не только матрица точек процесса, но и границы от всех ее соседей, как на рисунке 3.

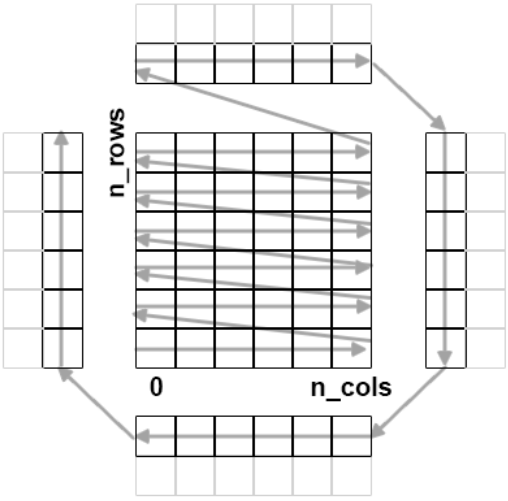

Рисунок 3. Хранение матрицы и границ соседних матриц.

Благодаря такой реализации хранения матрицы и соседних границ стало возможным за одно обращение к устройству обновить всю матрицу вместе с ее границами.

Обмен границами между процессором и устройством

Даже после избавления от 4-х кратного обращения к GPU на обновление R, P и G оказалось, что скорость недостаточно быстро растет с увеличением числа процессов.

Это происходило из-за того, что для посылки границ соседям их нужно было скопировать с устройства на процессор. А при получении границ от соседей нужно было скопировать их обратно.

Ситуация осложняется еще и тем, что ячейки столбцов матриц не лежат в соседних участках памяти — из-за этого копирование столбца с устройства на процессор становится операцией map-reduce — собрать ячейки столбца из всего массива «из-под капота» матрицы и скопировать результат в специальный буфер на процессоре.

Рассмотрим по отдельности, как реализовано копирование границ матрицы на процессор, и как реализована обратная операция:

- Копирование с устройства на процессор: во время вычисления матрицы R ее граничные столбцы копируются в два буфера, аллоцированных на устройстве — так операция map-reduce совмещается с вычислением R. После окончания вычисления есть два буфера с уже собранными значениями граничных столбцов, и есть две граничных строки, которые уже итак лежат в непрерывной памяти. Все 4 границы объединяются в один поток переноса данных (cudaStream) и запускается их асинхронное копирование на процессор (cudaMemcpyAsync). После чего процесс ожидает завершения всех 4-х копирований;

- Копирование с процессора на устройство: эта операция гораздо проще, так как границы соседей хранятся, согласно рисунку 3, в 4-х непрерывных областях памяти устройства. Для их копирования в тот же поток (cudaStream) процессор запускает 4 асинхронных копирования и ожидает их завершения.

Результаты запусков

Ломоносов, MPI

Число процессоров	Число точек сетки	Время решения	Ускорение
1	1000 x 1000	45.62 s	-
4	1000 x 1000	12.16 s	3.75
8	1000 x 1000	7.53 s	6.06
16	1000 x 1000	2.84 s	16.06
32	1000 x 1000	1.44 s	31.68
64	1000 x 1000	0.73 s	62.49
128	1000 x 1000	0.41 s	111.27
1	2000 x 2000	358.26 s	-
4	2000 x 2000	95.54 s	3.75
8	2000 x 2000	71.19 s	5.03
16	2000 x 2000	35.86 s	9.99
32	2000 x 2000	15.05 s	23.8
64	2000 x 2000	5.31 s	67.47
128	2000 x 2000	2.75 s	130.28

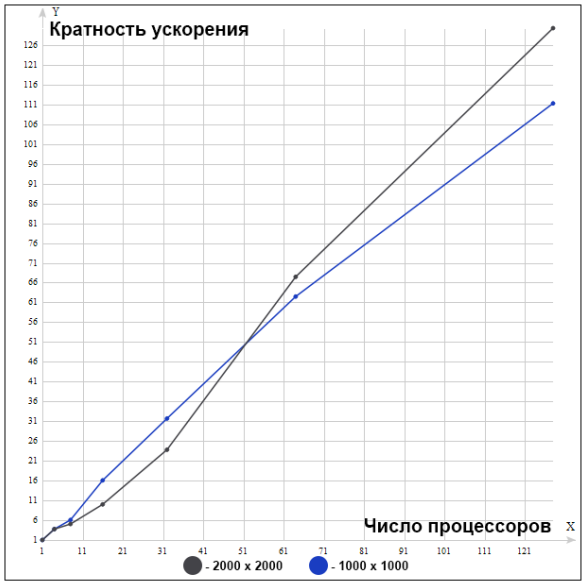

График 1. Зависимость ускорения от числа процессоров на компьютере Ломоносов.

Ломоносов, MPI + CUDA

Число процессоров	Число точек сетки	Время решения	Ускорение
1	1000 x 1000	4.83 s	-
4	1000 x 1000	4.31 s	1.12
8	1000 x 1000	4.36 s	1.11
16	1000 x 1000	4.47 s	1.08
1	2000 x 2000	31.44 s	-
4	2000 x 2000	16.74 s	1.88
8	2000 x 2000	10.33 s	3.04
16	2000 x 2000	8.48 s	3.71
1	4000 x 4000	224.76 s	-
4	4000 x 4000	105.28 s	2.13
8	4000 x 4000	57.24 s	3.93
16	4000 x 4000	28.45 s	7.9

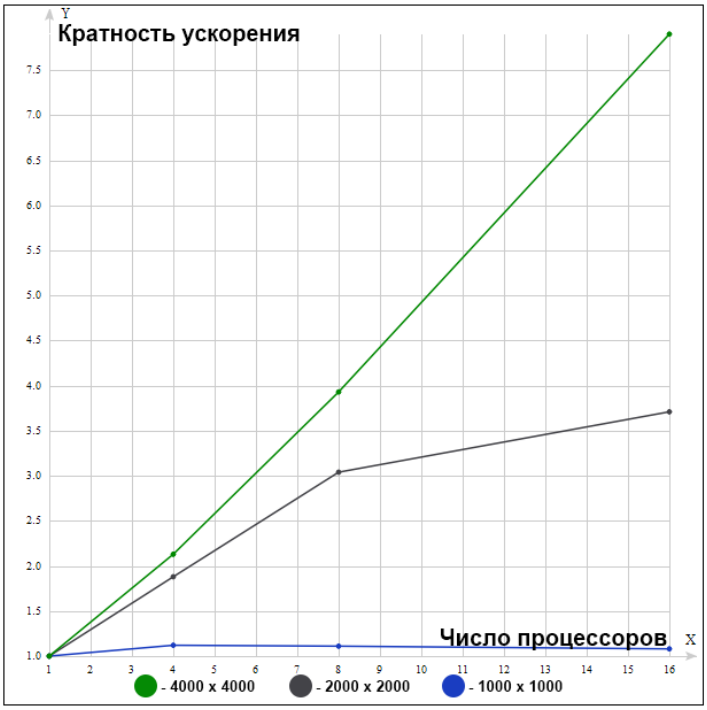

График 1. Зависимость ускорения от числа процессоров на компьютере Ломоносов с реализацией вычислений на CUDA.

BlueGene, MPI

Число процессоров	Число точек сетки	Время решения	Ускорение
1	1000 x 1000	398.21 s	-
4	1000 x 1000	100.85 s	3.95
8	1000 x 1000	50.81 s	7.84
16	1000 x 1000	25.24 s	15.78
128	1000 x 1000	3.51 s	113.45
256	1000 x 1000	1.86 s	214.09
512	1000 x 1000	1.03 s	386.61
1	2000 x 2000	3183.57 s	-
4	2000 x 2000	736.61 s	4.32
8	2000 x 2000	369.59 s	8.61
16	2000 x 2000	185.11 s	17.2
128	2000 x 2000	23.84 s	133.54
256	2000 x 2000	12.24 s	260.1
512	2000 x 2000	6.43 s	495.11

График 1. Зависимость ускорения от числа процессоров на компьютере BlueGene.

BlueGene, MPI + OMP

Число процессоров	Число точек сетки	Время решения	Ускорение
1	1000 x 1000	134.83 s	-
4	1000 x 1000	34.28 s	3.93
8	1000 x 1000	17.31 s	7.79
16	1000 x 1000	8.90 s	15.15
128	1000 x 1000	1.53 s	88.12
256	1000 x 1000	0.96 s	140.45
512	1000 x 1000	0.69 s	195.4
1	2000 x 2000	1132.67 s	-
4	2000 x 2000	247.93 s	4.57
8	2000 x 2000	124.22 s	9.12
16	2000 x 2000	62.61 s	18.09
128	2000 x 2000	8.72 s	129.89
256	2000 x 2000	4.79 s	236.47
512	2000 x 2000	2.81 s	403.08

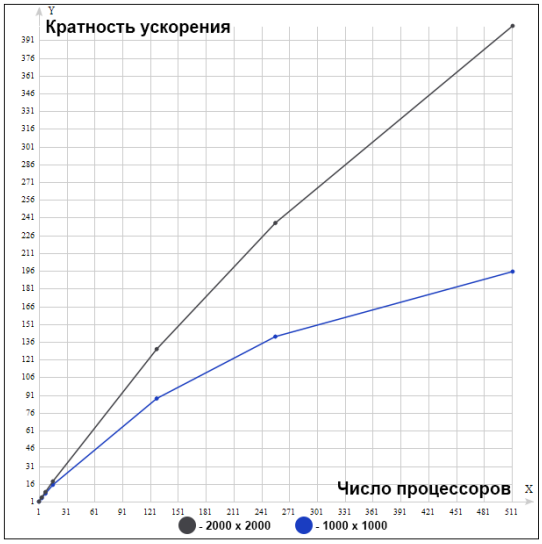

График 1. Зависимость ускорения от числа процессоров на компьютере BlueGene с OMP.

Результат и выводы

После работы программы получен следующий вид приближенного решения:

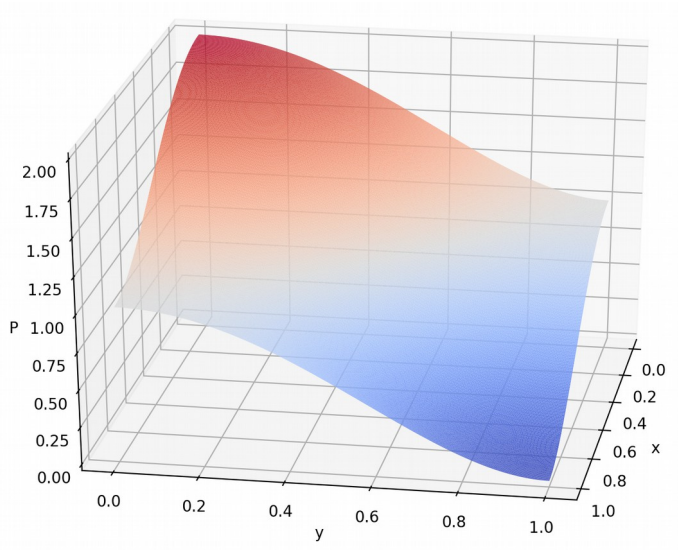

Рисунок 4. Решение.

Вид точного решения на графике неотличим от приближенного, и потому не продублирован здесь. Погрешность решения на сетках разного размера составляла порядка сотых или тысячных, в зависимости от размеров сетки.

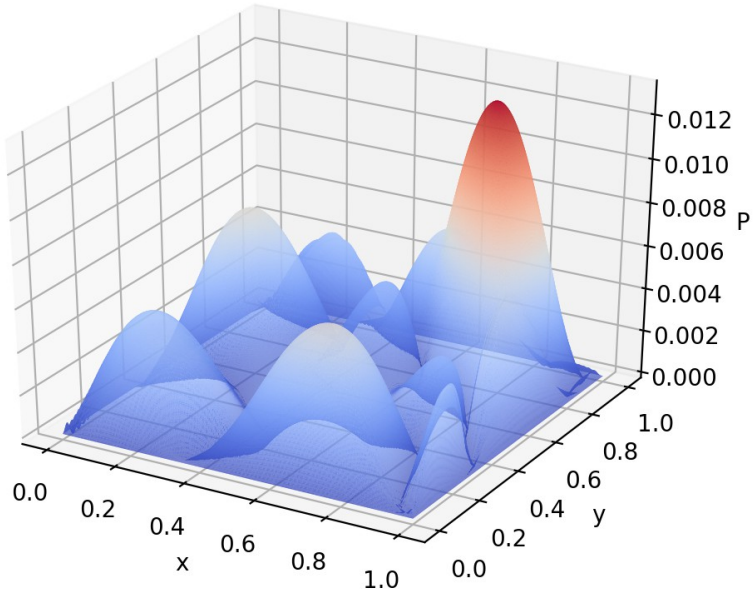

Рисунок 5. Абсолютная погрешность (относительно аналитического решения).

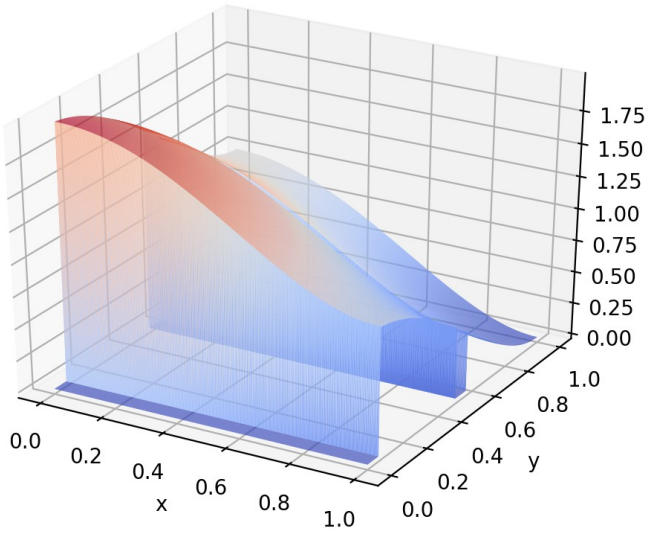

Рисунок 6. Относительная погрешность (относительно предпоследней итерации алгоритма).

Из анализа времени работы программы на различных компьютерах получены выводы:

- На Ломоносове без CUDA производительность растет почти линейно с увеличением числа процессоров, что вполне ожидаемо до некоторого предела. Когда все вычисления делаются на процессоре, а матрицы каждого процесса достаточно велики, то время обмена сообщениями не особенно влияет на скорость, тем более что обмен асинхронный;
- Реализация с CUDA на Ломоносове дает непростой для толкования результат. Как видно, рост скорости не линеен в общем случае, а на небольшой сетке даже замедляется. Все дело в том, что с ростом числа процессоров появляется больше накладных расходов на копирование границ матрицы R между CPU и GPU. На небольшой сетке эти расходы могут даже превосходить время расчетов. С ростом размеров сетки все больше времени уходит на вычисления, и относительно них время на копирование уже влияет не так сильно. Однако даже при этом скорость на всех запусках на одинаковых сетках и одинаковом числе процессоров

оказалась выше, чем без CUDA, чем BlueGene без OMP и чем BlueGene с OMP;

- Запуски на BlueGene приводят к таким же выводам, что и запуски на Ломоносове без CUDA — рост линейный. Можно разве что отметить, что с тремя потоками OMP скорость на том же числе процессоров, что и без OMP, возросла почти ровно в три раза.