Отчет по заданию №3 «Исследование численного решения задачи Дирихле для уравнения Пуассона в прямоугольной области» по курсу «Суперкомпьютерное моделирование и технологии».

Асирян Александр, 624 группа. Вариант 2\_23.

**Математическая постановка задачи**

В прямоугольной области требуется найти дважды гладкую функцию , удовлетворяющую дифференциальному уравнению:

и дополнительному условию:

во всех граничных точках прямоугольника. Оператор Лапласа определен равенством:

Вариант 2\_23:



**Численный метод решения задачи**

В расчетной области определяется прямоугольная сетка

где – разбиение отрезка оси (𝑜𝑥),

– разбиение отрезка оси (𝑜𝑦).

Через обозначим множество внутренних, а через – множество граничных узлов сетки . Пусть – переменный шаг сетки по оси абсцисс и ординат соответственно. Средние шаги сетки определяются равенствами:

Рассмотрим линейное пространство 𝐻 функций, заданных на сетке . Будем считать, что в пространстве 𝐻 задано скалярное произведение и евклидова норма

где – любые функции из пространства 𝐻.

Для аппроксимации уравнения Пуасона (1) воспользуемся пятиточечным разностным оператором Лапласа, который во внутренних узлах сетки определяется равенством:

Здесь предполагается, что функция определена во всех узлах сетки .

Приближенным решением задачи (1), (2) называется функция , удовлетворяющая уравнениям

Эти соотношения представляют собой систему линейных алгебраических уравнений с числом уравнений равным числу неизвестных и определяют единственным образом неизвестные значения . Совокупность уравнений (4) называется разностной схемой для задачи (1), (2).

Приближенное решение системы уравнений (4) может быть получено итерационным методом скорейшего спуска. В этом методе начальное приближение

во внутренних узлах сетки – любые числа. Метод является одношаговым. Итерация вычисляется по итерации согласно равенствам:

где невязка

Итерационный параметр

Известно, что с увеличением номера итерации 𝑘 последовательность сеточных функций сходится к точному решению 𝑝 задачи (4) по норме пространства 𝐻, то есть

Существенно большей скоростью сходимости обладает метод сопряженных градиентов. Начальное приближение и первая итерация вычисляются так же, как и в методе скорейшего спуска. Последующие итерации осуществляются по формулам:

Здесь

вектор

коэффициент

Вектор невязки вычисляется согласно равенствам (5). Итерационный процесс останавливается, как только

где 𝜀 – заранее выбранное положительное число. В последнем неравенстве, согласно варианту, используется евклидова сеточная норма.

Для аппроксимации дифференциальной задачи используется равномерная прямоугольная сетка:

Приближенное решение разностной схемы (4) следует вычислять методом сопряженных градиентов. Для остановки итерационного процесса предлагается использовать условие (6), положив .

**Постановка задачи**

Для функций :

* подобрать точное решение задачи Дирихле,
* методом сопряженных градиентов построить приближенное решение на сетке с числом узлов , определить погрешность решения
* методом сопряженных градиентов построить приближенное решение на сетке с числом узлов и вновь определить погрешность решения.

Расчеты необходимо проводить на многопроцессорных вычислительных комплексах IBM Blue Gene/P и «Ломоносов», используя различное количество вычислительных узлов, указанное в требованиях к отчету. Для каждого расчета определить его продолжительность и ускорение по сравнению с аналогичным расчетом на одном вычислительном узле. При распараллеливании программы необходимо использовать двумерное разбиение области на подобласти прямоугольной формы, в каждой из которых отношение 𝜃 количества узлов по ширине и длине должно удовлетворять неравенствам .

**Проделанная работа по созданию гибридных реализаций MPI/OpenMP и MPI/CUDA**

**MPI/OpenMP**

Процессоры разбиваются на сетку , где

|  |  |  |  |
| --- | --- | --- | --- |
| 0 (0, 0) | 1 (0, 1) | 2 (0, 2) | 3 (0, 3) |
| 4 (1, 0) | 5 (1, 1) | 6 (1, 2) | 7 (1, 3) |

То есть если количество процессоров является квадратом, то сетка квадратная, иначе она делится на две части по горизонтали, каждая из которых разбивается на квадратную. Далее, с помощью функции *MPI\_Cart\_create* создается виртуальная топология процессоров. Координаты процессора в топологии и номера соседних процессоров вычисляются с помощью функций *MPI\_Cart\_coor*ds и *MPI\_Cart\_shift* соответственно. Например, для топология будет иметь следующий вид:

Точки распределяются на процессоры, и каждый процессор обрабатывает только свои точки. Так как количество точек по координате сетки может не делиться нацело на количество процессоров по соответствующей координате в топологии, то лишние точки распределяются на последний процессор по этой координате. Таким образом, все процессоры, кроме нижней и правой границы, будут иметь одинаковое количество обрабатываемых точек.

Например, для топология будет иметь следующий вид (81 точка):

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |

После распределения каждый процессор вычисляет значения переменных из метода для своих точек. Итерации циклов в расчетах распределяются на потоки с помощью директивы OpenMP *parallel for*. Для подсчета частичного скалярного произведения используется директива *reduction* для суммы.

Для вычисления оператора Лапласа требуются данные с точек соседних процессоров. Для этого реализуется обмен между процессорами с помощью команд *MPI\_Isend* и *MPI\_Irecv* (так как все пересылки являются парными, то можно использовать асинхронные версии). Рассылаются только граничные точки:

|  |
| --- |
| N |

- внутренние точки процессора N

|  |
| --- |
|  |

- граничные точки

|  |
| --- |
|  |

- точки соседнего процессора

Процессор1 Процессор2

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  | 1 | 1 | 1 | 1 | 2 |
|  | 1 | 1 | 1 | 1 | 2 |
|  | 1 | 1 | 1 | 1 | 2 |
|  | 1 | 1 | 1 | 1 | 2 |
|  |  |  |  |  |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
| 1 | 2 | 2 | 2 | 2 |  |
| 1 | 2 | 2 | 2 | 2 |  |
| 1 | 2 | 2 | 2 | 2 |  |
| 1 | 2 | 2 | 2 | 2 |  |
|  |  |  |  |  |  |

Последовательность обменов следующая:

1. получить точки у соседа снизу
2. получить точки у соседа справа
3. отправить точки соседу сверху
4. отправить точки соседу слева
5. отправить точки соседу снизу
6. отправить точки соседу справа
7. получить точки у соседа сверху
8. получить точки у соседа слева

Для топологии, указанной выше, получаются следующая последовательность шагов, если рассматривать синхронные обмены:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **0 (0:0)** | **1 (0:1)** | **2 (0:2)** | **3 (0:3)** | **4 (1:0)** | **5 (1:1)** | **6 (1:2)** | **7 (1:3)** |
| . | . | . | RD7 | . | . | . | SU3 |
| . | . | . | . | . | . | RR7 | SL6 |
| . | . | RD6 | . | . | . | SU2 | . |
| . | . | RR3 | SL2 | . | RR6 | SL5 | . |
| . | RD5 | . | SD7 | . | SU1 | . | RU3 |
| . | RR2 | SL1 | RL2 | RR5 | SL4 | SR7 | RL6 |
| RD4 | . | SD6 |  | SU0 | . | RU2 |  |
| RR1 | SL0 | SR3 |  | . | SR6 | RL5 |  |
| . | SD5 | RL1 |  | . | RU1 |  |  |
| . | SR2 |  |  | SR5 | RL4 |  |  |
| SD4 | RL0 |  |  | RU0 |  |  |  |
| SR1 |  |  |  |  |  |  |  |

XYN – send/receive, направление, номер процессора.

RD7 – получить точки снизу от процессора 7.

Чтобы посчитать итерационный параметр , коэффициент и проверить условие остановки итерационного процесса необходимо вычислить полное скалярное произведение. Для этого, с помощью функции *MPI\_Allreduce*, локальные частичные скалярные произведение суммируются и отправляются по всем процессорам.

**MPI/CUDA**

Для решения задачи на графических ускорителях использовалась библиотека Thrust (<http://docs.nvidia.com/cuda/thrust/index.html>). Так как на одном узле вычислялось два процесса, то ГПУ привязывался к процессу с помощью вызова *cudaSetDevice*: процессы с четным номером устанавливали первую видеокарту, с нечетным – вторую. Все директивы OpenMP были заменены. Во всех случаях, кроме редукции, использовалась функция *transform*. Для некоторых ее вызовов были реализованы соответствующие функторы. Для сокращения количества обходимых точек, где это было возможно, применялся *permutation\_iterator*, в остальных случаях применялась комбинация *counting\_iterator* для идентификации индекса в функторе и условие выхода из функтора без расчетов. Редукция осуществлялась с помощью вызова *transform\_reduce*. Обмен граничными точками между GPU и CPU для передачи с помощью MPI и после их получения производился вызовом функции *copy*.

**Результаты расчетов**

**Погрешность и проверка результатов**

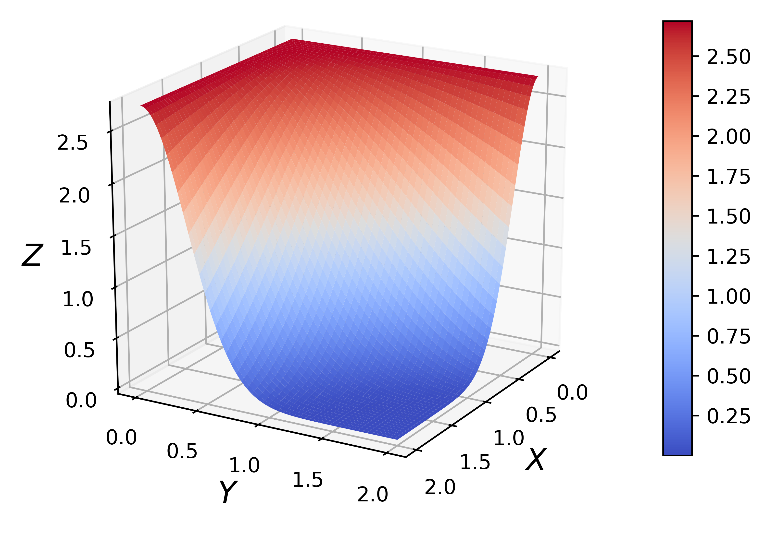
1500 x 1500: 0.011614

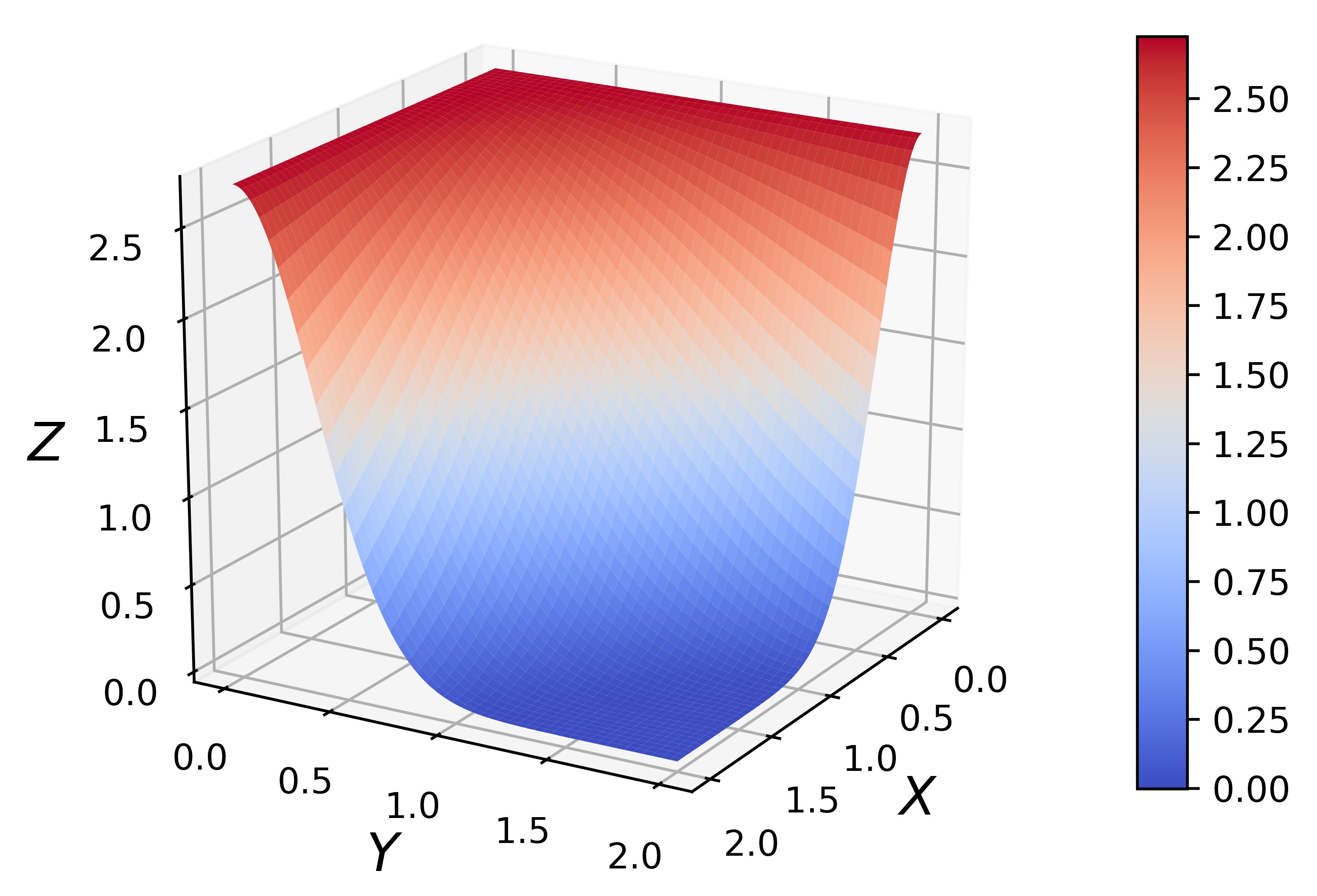
2000 x 2000: 0.011669

2500 x 2500: 0.011710

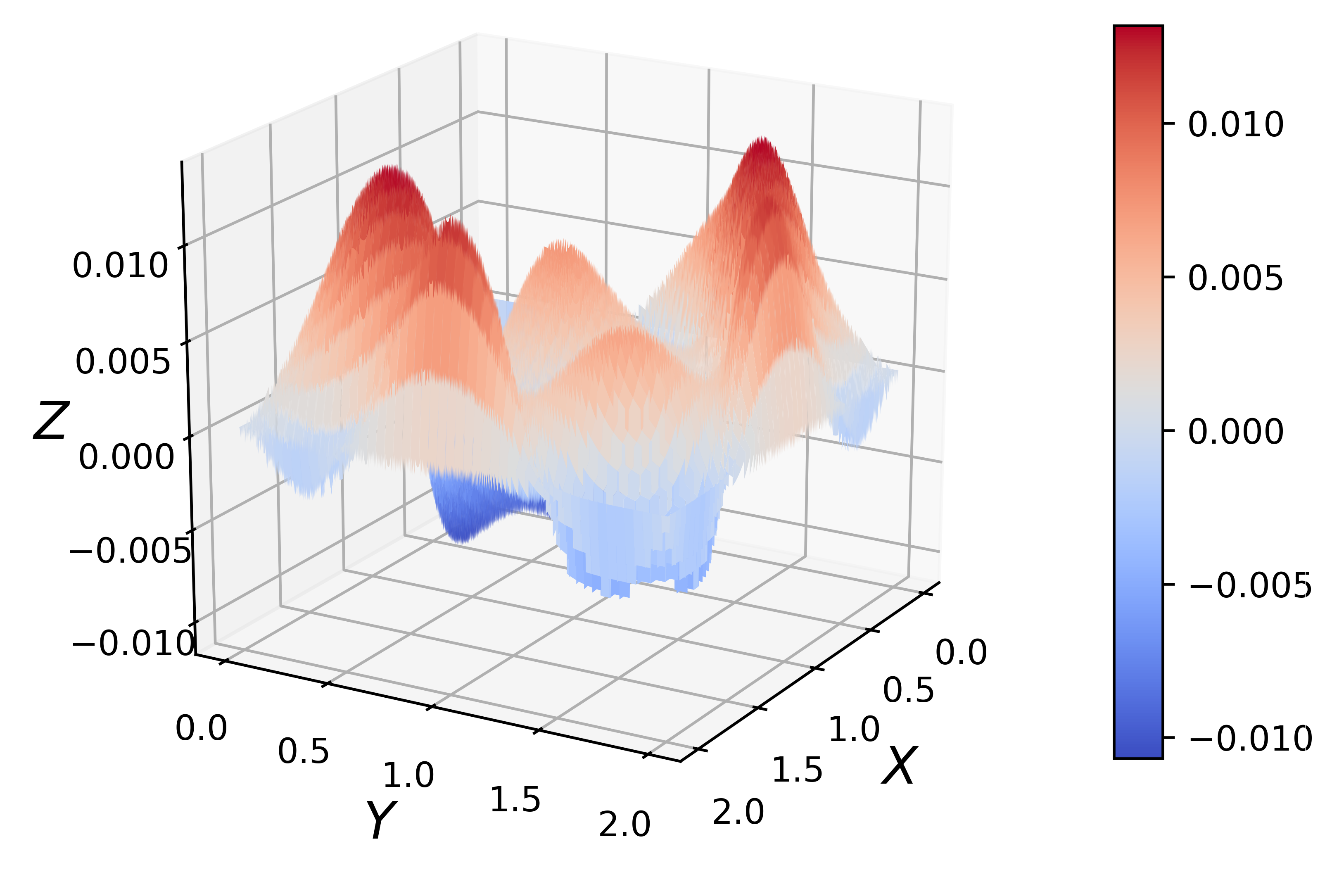
Пример проверки решения на сетке 2000 х 2000:

Приближенное решение Точное решение





Погрешность



**Время и ускорение**

Время рассчитывалось как усредненное значение трех запусков.

Ускорение

**Lomonosov MPI**

строка компиляции: mpicxx -O3 task2.cpp -o task2

* Очередь: test
* impi
* Количество процессов: 1, 2, 4, 8, 16
* Процессов на узел: 1

|  |  |  |  |
| --- | --- | --- | --- |
| Число процессов | Число точек сетки | Время решения 𝑇 | Ускорение 𝑆 |
| 1 | 1500 x 1500 | 145.708 | 1.000 |
| 2 | 1500 x 1500 | 73.982 | 1.970 |
| 4 | 1500 x 1500 | 35.891 | 4.060 |
| 8 | 1500 x 1500 | 15.943 | 9.139 |
| 16 | 1500 x 1500 | 7.631 | 19.093 |
| 1 | 2000 x 2000 | 337.375 | 1.000 |
| 2 | 2000 x 2000 | 169.346 | 1.992 |
| 4 | 2000 x 2000 | 89.261 | 3.780 |
| 8 | 2000 x 2000 | 48.671 | 6.932 |
| 16 | 2000 x 2000 | 19.084 | 17.679 |
| 1 | 2500 x 2500 | 680.888 | 1.000 |
| 2 | 2500 x 2500 | 362.957 | 1.876 |
| 4 | 2500 x 2500 | 170.626 | 3.991 |
| 8 | 2500 x 2500 | 145.160 | 4.691 |
| 16 | 2500 x 2500 | 48.325 | 14.090 |

**Lomonosov MPI/OpenMP**

строка компиляции: mpicxx -O3 –fopenmp task2.cpp -o task2\_omp

* Очередь: test
* impi
* Количество процессов: 1, 2, 4, 8, 16
* Процессов на узел: 1
* Количество ядер на процесс: 8
* Количество нитей на процесс: 16

|  |  |  |  |
| --- | --- | --- | --- |
| Число процессов | Число точек сетки | Время решения 𝑇 | Ускорение 𝑆 |
| 1 | 1500 x 1500 | 61.188 | 2.381 |
| 2 | 1500 x 1500 | 28.486 | 5.115 |
| 4 | 1500 x 1500 | 7.578 | 19.227 |
| 8 | 1500 x 1500 | 2.797 | 52.092 |
| 16 | 1500 x 1500 | 1.723 | 84.587 |
| 1 | 2000 x 2000 | 131.903 | 2.558 |
| 2 | 2000 x 2000 | 81.212 | 4.154 |
| 4 | 2000 x 2000 | 29.573 | 11.408 |
| 8 | 2000 x 2000 | 10.129 | 33.307 |
| 16 | 2000 x 2000 | 3.551 | 95.017 |
| 1 | 2500 x 2500 | 263.000 | 2.589 |
| 2 | 2500 x 2500 | 143.972 | 4.729 |
| 4 | 2500 x 2500 | 102.064 | 6.671 |
| 8 | 2500 x 2500 | 31.622 | 21.532 |
| 16 | 2500 x 2500 | 11.507 | 59.171 |

**Lomonosov MPI/CUDA**

строка компиляции: nvcc -arch=sm\_20 -ccbin mpicxx -o task3 -Xcompiler -O3 task3.cu

* Очередь: gputest
* impi
* Количество процессов: 2, 4, 8, 16
* Процессов на узел: 2
* Количество ГПУ на процесс: 1

|  |  |  |  |
| --- | --- | --- | --- |
| Число процессов | Число точек сетки | Время решения 𝑇 | Ускорение 𝑆 |
| 2 | 1500 x 1500 | 20.901 | 6.971 |
| 4 | 1500 x 1500 | 15.64 | 9.316 |
| 8 | 1500 x 1500 | 11.996 | 12.146 |
| 16 | 1500 x 1500 | 5.156 | 28.260 |
| 2 | 2000 x 2000 | 30.649 | 11.008 |
| 4 | 2000 x 2000 | 26.757 | 12.609 |
| 8 | 2000 x 2000 | 18.549 | 18.188 |
| 16 | 2000 x 2000 | 11.542 | 29.230 |
| 2 | 2500 x 2500 | 70.46 | 9.663 |
| 4 | 2500 x 2500 | 31.559 | 21.575 |
| 8 | 2500 x 2500 | 19.054 | 35.735 |
| 16 | 2500 x 2500 | 16.182 | 42.077 |

**График ускорений**

Видно, что для программ MPI и MPI/CUDA (1500 х 1500) ускорение меняется незначительно до 8 процессоров и уже на 16 заметно возрастает. Это может быть связано с размером матрицы, так как при большем количестве процессов, меньшее количество точек обрабатывается каждым процессом и графическим ускорителем. Также из-за небольшого порядка ( количества точек MPI пересылки и обмен данными между CPU и GPU несут издержки, сопоставимые с временем подсчета. Графики для MPI/CUDA на сетке размером 2000 х 2000 и 2500 х 2500 выглядят более линейно. Это может быть связано с тем, что на сетке такого размера время расчетов превосходит накладные расходы. Для MPI/OpenMP программы ускорение примерно растет так же, как и количество процессов. В отличие от MPI/CUDA обмен не происходит, и задача распараллеливается практически без накладных расходов, разве что при редукции.