МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Информационные технологии и прикладная математика»
Кафедра 806 «Вычислительная математика и программирование»

**Лабораторная работа №2**
**по курсу «Параллельная обработка данных»**

# Технология MPI и технология CUDA. MPI-IO

Выполнил: М.А. Бронников
Группа: М8О-407Б
Преподаватели: К.Г. Крашенинников,
А.Ю. Морозов

Москва, 2020

## Условие

**Цель работы:** Совместное использование технологии MPI и технологии CUDA. Применение библиотеки алгоритмов для параллельных расчетов Thrust. Реализация метода Якоби. Решение задачи Дирихле для уравнения Лапласа в трехмерной области с граничными условиями первого рода. Использование механизмов MPI-IO и производных типов данных.

**Вариант №1:** конструктор типа MPI_Type_create_subarray

## Программное и аппаратное обеспечение компьютера:

Device: GeForce GT 545
Размер глобальной памяти: 3150381056
Размер константной памяти : 65536
Размер разделяемой памяти: 49152
Регистров на блок: 32768
Максимум потоков на блок: 1024
Количество мультипроцессоров : 3
OS: Linux Mint 20 Cinnamon
Редактор: VSCode
Машины в кластере:
1. Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz, 16 Gb, GeForce GTX 1050, 2 Gb
2. Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz, 16 Gb, GeForce GT 545, 3 Gb
3. Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz, 16 Gb, GeForce GTX 650, 2 Gb
4. Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz, 12 Gb, GeForce GT 530, 2 Gb
5. Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz, 8 Gb, GeForce GT 530, 2 Gb
Все машины соединены гигабатным ethernet и находятся в подсети 10.10.1.1/24. В качестве операционной системы установлена Ubuntu 16.04.6 LTS. Версии софта:
mpirun 1.10.2, g++ 4.8.4, nvcc 7.0

## Метод решения

Для решения задачи на сетке заднного размера я использовал сетку процессов, каждый из которых имел свой участок памяти для обработки блока. Каждый процесс имел 2 равных по величине блока для того, чтобы на основе «старых» значений вычислять «новые» по формуле:

$$u_{i,j,k}^{(k+1)} = \frac{\left(u_{i+1,j,k}^{(k)}+u_{i-1,j,k}^{(k)}\right)h_x^{-2}+\left(u_{i,j+1,k}^{(k)}+u_{i,j-1,k}^{(k)}\right)h_y^{-2}+\left(u_{i,j,k+1}^{(k)}+u_{i,j,k-1}^{(k)}\right)h_z^{-2}}{2\left(h_x^{-2}+h_y^{-2}+h_z^{-2}\right)},$$

Проблема такого подхода заключается в том, что расчет граничных значений требует знаний о значениях, рассчитанных в другом блоке, что является не тривиальной задачей, требующей реализации межпроцессорного взаимодействия между соседями.

Общая схема решения:
1. На первом этапе происходит обмен граничными слоями между процессами.
2. На втором этапе выполняется обновление значений во всех ячейках.
3. Третий этап заключается в вычислении погрешности: сначала локально в рамках каждого процесса а потом через обмены и во всей области.

## Описание программы

Все расчеты новых значений на каждой иттерации алгоритма в данной лабораторной работе выполнялись на граффическом процессоре с помощью технологии Cuda. Для этого я выделил 2 блока данных на графическом процессоре, логика обновлений значений в которых аналогична тому, как мы вычисляли значения в лабораторной работе *MPI+OMP*. При этом для обмена данными между процессами я использую блоки, память для которых выделена на хосте и на каждой иттерации я делаю копирования данных между device и host памятью для переопределения отсылаемых/принимаемых значений.

Для обмена я в самом начале итерации начинаю ассинхронный обмен с помощью функций isend/irecv. После чего, чтобы не терять время на ожидании конца приема, было бы рационально начинать проход по середине блока, рассчитывая новые значения за исключением границ. Онако по опыту предыдущих лабораторных я отказался от этой идее, поэтому после вызова всех ассинхронных методов обмена данными я сразу вызываю функцию ожидания конца обмена и начинаю расчет значенй.

Во время расчета новых значений я постоянно изменяю значение нормы разности по блоку. Это необходимо для контроля границы. Для того, чтобы вычислить максимальное значение разности я создал дополнительный массив максимальных значений по каждому из потоков, максимальное значение уже из которой позволякт находить библилиотека thrust. Функция Allgather, которая возвращает все значения максимума по каждому из процессу, позволяет рассчитать значение для контроля простым проходом по выходному массиву.

Для записи результатов работы в файл я создал вспомогательный тип данных — субмассив с помощью которого я задал параметры общего многопроцессорного записи а файл.

## Результаты:

Сравним время выполнения двух разных программ: написанных для CPU, MPI и MPI+CUDA Будем делать честное сравнение, поэтому замерим полное время

выполнения программы, а не только основной цикл. Будем подбирать такие значения, чтобы размер общей сетки был одинаковым при разных запусках:

| Сетка \ Расчет | MPI | CPU | MPI+CUDA |
|---|---|---|---|
| 1 1 1 \| 40 40 40 | 19861ms | 9843.8ms | 5680 |
| 2 2 2 \| 20 20 20 | 4943.57ms | 9952.2ms | 38509.4 |
| 2 2 4 \| 20 20 10 | 6224.85ms | 9890.5ms | 97931.2 |

Как видим, использование нескольких процессов дает существенный прирост программе MPI, а также превосходит по времени программу, написанную на CPU, однако этот прирост едва ли можно назвать впечатляющим, поскольку плюсы MPI в лице распараллеливания кода нивелируются минусами, такими как необходимость постоянно делать обмен данными между процессами, что довольно сильно сказывается на производительности, особенно во время записи результата.

## Выводы

Данный метод Дирихле являетя одним из конечно-разностных методов, которые широко используются для решения дифференциальных уравнений на заданной сетке с высокой степенью точности. Без этих методов сложно представить современную физику, которая использует эти методы для расчета уравнений теплопроводности при разных условиях среды.

Эта лабораторная познакомила меня с тем, как использовать CUDA в связке с MPI, что позволяет разбивать программу на процессы, каждый из которых будет иметь несколько потоков исполнения. Также я научился делать мультипроцессорный вывод в файл.

Прирост полученный мной при однопроцессорном запуске меня приятно удивил, однако запуск программы на нескольких процессах меня огорчил своим безумно медленным выполнением, что объясняется тем, что кластеру не хватило ресурсов, чтобы справиться с сильно возросшей нагрузкой, а также повлияло время, которое тратиться на создание и инициализацию потоков на CPU, что более ресурсоёмко, нежели, чем на CPU. Скорее всего, если бы я изначально выбрал другую стратегию по организации своего кода, я бы смог добиться лучших результатов за счет уменьшения накладных расходов на распараллеливание участков моего кода.