

МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ  
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

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

**Лабораторная работа №3**  
**по курсу «Параллельная обработка данных»**  
**Технологии MPI и OpenMP.**

Выполнил: М.А. Жерлыгин

Группа: 8О-408Б

Преподаватели: К.Г. Крашенинников,  
А.Ю. Морозов

Москва, 2021

## Условие

**Цель работы:** Совместное использование технологии MPI и технологии OpenMP. Реализация метода Якоби. Решение задачи Дирихле для уравнения Лапласа в трехмерной области с граничными условиями первого рода.

**Вариант 3.** Распараллеливание основных циклов через parallel for

## Программное и аппаратное обеспечение

GPU:

1. Compute capability: 7.5;
2. Графическая память: 4294967296;
3. Разделяемая память: 49152;
4. Константная память: 65536;
5. Количество регистров на блок: 65536;
6. Максимальное количество блоков: (2147483647, 65535, 65535);
7. Максимальное количество нитей: (1024, 1024, 64);
8. Количество мультипроцессоров: 6.

Сведения о системе:

1. Процессор: Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.59 GHz
2. Память: 16,0 ГБ;
3. HDD: 237 ГБ.

Программное обеспечение:

1. OS: Windows 10;
2. IDE: Visual Studio 2019;
3. Компилятор: mpirun.

## Метод решения

Для решения задачи на сетке заданного 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:

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

Проблема заключается в том, что расчет граничных значений требует значения, рассчитанные в другом блоке, что является нетривиальной задачей, требующей реализации межпроцессорного взаимодействия между соседями.

Схема решения:

1. Передать данные другим процессам на границах.
2. Обновить данные во всех ячейках.
3. Вычисление локальной (в рамках процесса) погрешности и во всей области.

## Описание программы

В отличии от лабораторной № 1 добавились директивы препроцессора `#pragma parallel`, которые позволяют задавать участки, которые будут выполняться в многопроцессорном режиме для каждого из потоков. Для того, чтобы каждый поток отвечал за отдельный участок, нужно было добавить эту директиву перед каждым циклом, за исключением вычисления погрешности.

```
#pragma omp parallel for private(i, j, k) shared(values, next_values)
```

Где `values` - буффер, а `next_values` - след. массив с данными.

```
#pragma omp parallel for private(i, j, k) shared(values, next_values) reduction(max: diff)
```

Директива `reduction` необходима для вычисления погрешности.

Для передачи и получения данных я использую `sendrecv`. Она комбинирует в одном обращении посылку сообщения одному получателю и прием сообщения от другого отправителя.

Затем вычисляю локальную (в рамках процесса) погрешность.

И вызываю `Allgather`.

Как работает `allgather`.

Он получает значения от одного процесса и отдаёт их всем остальным. В таком случае получается, что накапливаются значения во всех остальных процессах, включая тот самый. И в итоге мы получаем массив отсортированных по процессам значений, то есть первый процесс - одно значение, второй процесс - второе и т.д. Таким образом мы можем накопить побочный максимум во всех остальных процессах и найти наибольший из них.

```
MPI_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)
```

`sendbuf` - указатель на буфер, `sendcount` - сколько слов отправляется, `sendtype` - тип отправленного (`int`, `double`), `recvbuf` - куда будем принимать, `recvcount` - сколько слов

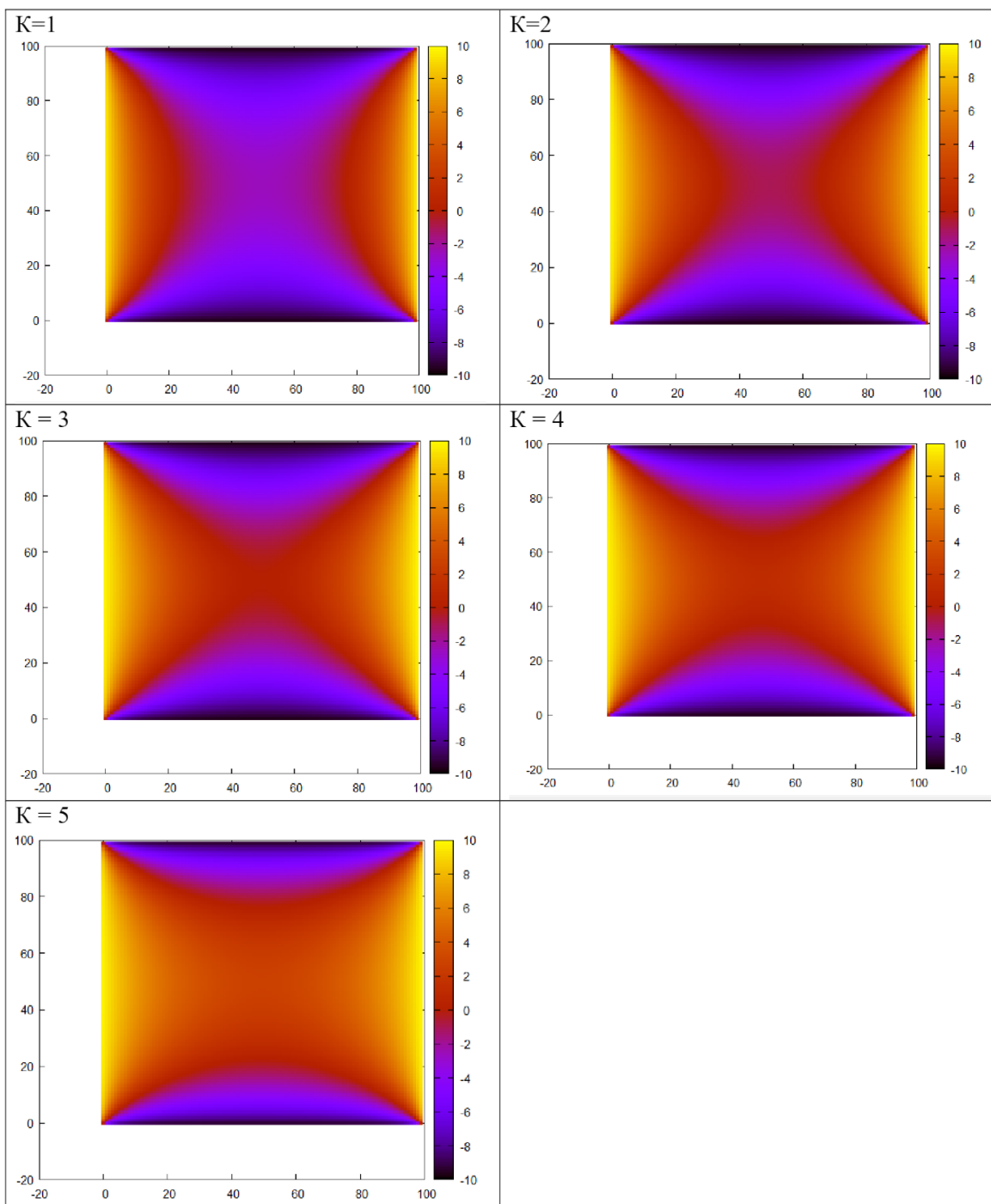
мы принимаем от одного другого процесса, включая этот, recvtype - тип полученного (int, double), comm - коммуникатор.

Затем сравниваю полученный максимум с eps, и если он меньше, прекращаю основной цикл.

## Результаты

Размер блоков / расчета	MPI	MPI + Cuda	MPI + OMP
1 1 1   30 30 30	17861ms	4892ms	7121ms

Можно заметить, что совместное использование MPI и OMP работает быстрее MPI, но медленнее MPI + Cuda, несмотря даже на то, что эта программа внутри себя сталкивается со всеми издержками на пересылки и прием данных. Также можно добиться лучшего распараллеливания, используя директиву ещё в некоторых местах программы.



## **Выводы**

Данный метод Дирихле является 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.

Данная лабораторная работа познакомила меня с технологией OMP, позволяющей легко и быстро распараллеливать свой код с помощью легковесных потоков. Также я научился использовать ее вместе с MPI, что позволяет разбивать программу на процессы, каждый из которых будет иметь несколько потоков исполнения и получать существенный прирост в скорости вычисления.