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 OpenMP**

Выполнил: В.А. Петросян

Группа: 8О-408Б-17

Преподаватели: К.Г. Крашенинников,

А.Ю. Морозов

Москва, 2020

**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 2**: Распараллеливание в общем виде с разделением работы между нитями

вручную (“в стиле MPI”). Обмен граничными слоями через bsend, контроль сходимости allgather;

**Программное и аппаратное обеспечение**

Сведения о системе:

1. Процессор: Intel Core i7-Q720 1.60GHz
2. Количество ядер 4.
3. Количество потоков 8.

 4. Оперативная память: 8 ГБ

 5. HDD: 465 ГБ

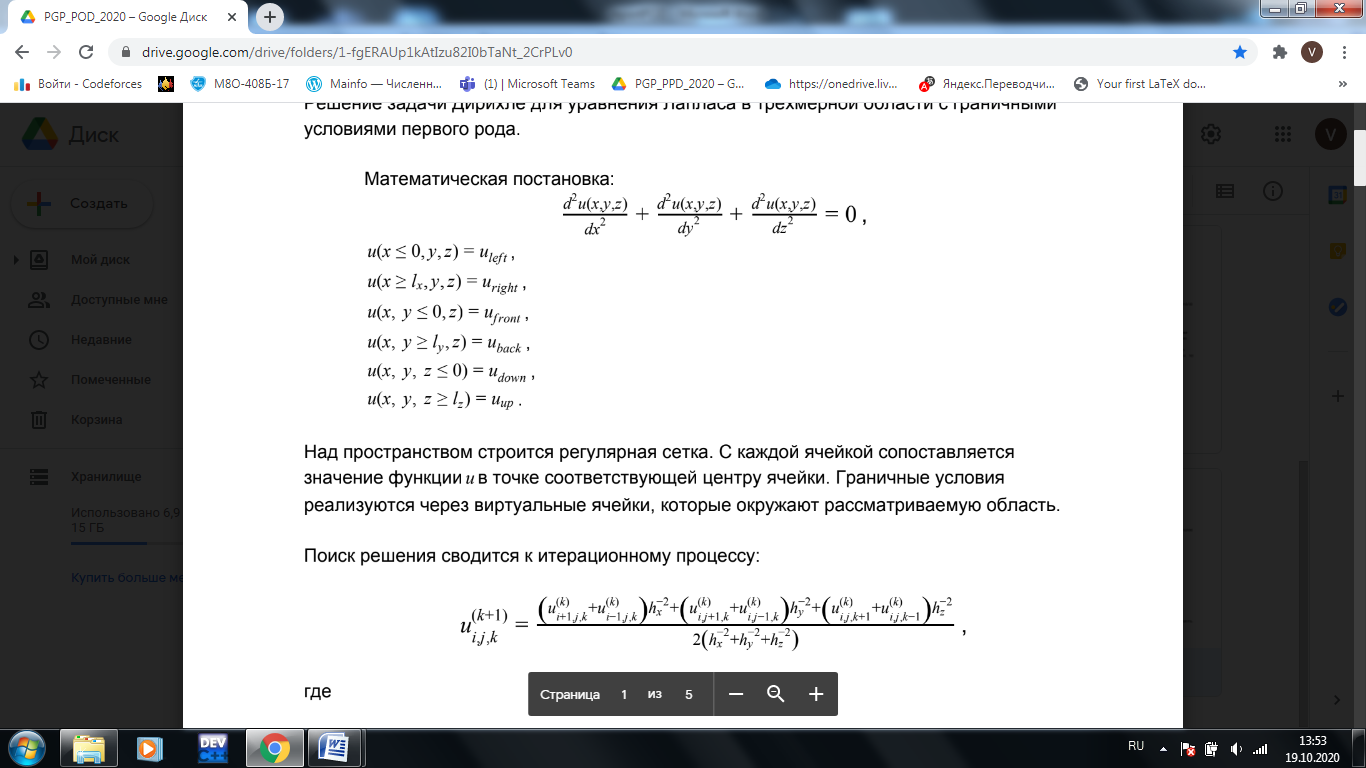
Программное обеспечение:

 1. OS: Windows 7

 2. IDE: Visual Studio 2019

 3. Компилятор: mpic++

**Метод решения**



Опишу немного логику работы с данными. Допустим размер блока, который обсчитывает один процесс это x \* y \* z. Мы выделим на каждое измерение два дополнительных элемента для хранения граничных условий. Теперь блок имеет размер (x + 2) \* (y + 2) \* (z + 2) . Храним данные блока в виде одномерного массива, но обращаемся к нему как к трёхмерному. Чтобы было проще взаимодействовать с массивом, напишем пару макросов для правильного доступа по индексу к данным.

// Индексация внутри блока

#define \_i(i, j, k) (((k) + 1)\*(blockY + 2)\*(blockX + 2) + ((j) + 1)\*(blockX + 2) + (i) + 1)

#define \_ix(id) (((id) % (blockX + 2)) - 1)

#define \_iy(id) ((((id) % ((blockY + 2) \* (blockX + 2))) / (blockX + 2)) - 1)

#define \_iz(id) ( ( (id) / ((blockY + 2) \* (blockX + 2)) ) - 1)

Пока не достигнем нужной точности *ε* будем делиться нашими данными с соседними по сетке процессами.

**Описание программы**

MPI\_Status status;

MPI\_Init(&argc, &argv);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &numproc); - общее количество процессов.

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &id); - номер нашего процесса 0 <= id < numproc.

MPI\_Bcast(&blockX, 1, MPI\_INT, 0, MPI\_COMM\_WORLD); - “Широковещательное сообщение” передает всем процессам значение переменной blockX.

ib = \_ibx(id); - индексация процесса в сетке блоков по x

jb = \_iby(id); - индексация процесса в сетке блоков по y

kb = \_ibz(id); - индексация процесса в сетке блоков по z

int buffer\_size = 12 \* sizeOfBuff \* sizeof(double) + 12 \* MPI\_BSEND\_OVERHEAD;

double \*buffer = (double \*)malloc(buffer\_size);

MPI\_Buffer\_attach(buffer, buffer\_size); - через этот буфер процесс будет общаться со всеми остальными. Я выделим место с двойным запасом.

for(i = -1; i <= blockX; i++){

   for(j = -1; j <= blockY; j++){

       for(k = -1; k <= blockZ; k++){

           data[\_i(i, j, k)] = startU; - инициализация начальным условием

       }

   }

}

MPI\_Barrier(MPI\_COMM\_WORLD); - синхронизирует процессы внутри коммутатора. В коммутаторе MPI\_COMM\_WORLD содержатся все процессы.

if (ib + 1 < gridX){

   for(j = 0; j < blockY; j++){

       for(k = 0; k < blockZ; k++){

           buff[j \* blockZ + k] = data[\_i(blockX - 1, j, k)];

       }

   }

Посылка данных процессу соседу. Так как пространство трехмерное, то мы посылаем двумерный массив. Посылка производится через Bsend.

   int tmpSize = blockY \* blockZ;

   MPI\_Bsend(buff, tmpSize, MPI\_DOUBLE, \_ib(ib + 1, jb, kb), id, MPI\_COMM\_WORLD);

}

if (ib > 0) {

    int tmpSize = blockY \* blockZ;

    MPI\_Recv(buff, tmpSize, MPI\_DOUBLE, \_ib(ib - 1, jb, kb),

\_ib(ib - 1, jb, kb), MPI\_COMM\_WORLD, &status);

    for(j = 0; j < blockY; j++){

        for(k = 0; k < blockZ; k++){

            data[\_i(-1, j, k)] = buff[j \* blockZ + k];

        }

    }

}

else {

    for(j = 0; j < blockY; j++){

        for(k = 0; k < blockZ; k++){

            data[\_i(-1, j, k)] = leftU;

        }

    }

}

Опишу работу с openMP на конкретном примере распараллеливания цикла

#pragma omp parallel reduction(max:localMax)

 {

  int numThreads = omp\_get\_num\_threads();

  int threadNum = omp\_get\_thread\_num();

  for(int index = threadNum; index < blockX \* blockY \* blockZ; index += numThreads) {

        int i = index % blockX;

         int j = (index % (blockX \* blockY)) / blockX;

         int k = index / (blockX \* blockY);

         next[\_i(i, j, k)] = 0.5 \*

( (data[\_i(i + 1, j, k)] + data[\_i(i 1, j, k)]) / (hx \* hx) + (data[\_i(i, j + 1, k)] + data[\_i(i, j - 1, k)]) / (hy \* hy) + (data[\_i(i, j, k + 1)] + data[\_i(i, j, k - 1)]) / (hz \* hz)) /

                                (1.0 / (hx \* hx) + 1.0 / (hy \* hy) + 1.0 / (hz \* hz));

         if( abs(next[\_i(i, j, k)] - data[\_i(i, j, k)]) > localMax ){

             localMax = abs(next[\_i(i, j, k)] - data[\_i(i, j, k)]);

         }

   }

}

Сначала объявляем начало параллельной области с помощью

#pragma omp parallel

У меня был тройной цикл по переменным i, j, k. Переписал его в один большой цикл, чтобы было проще распараллелить. Логика работы с данными точно такая же как в 1 ЛР по Cuda. Перебираю элементы массива со смещение для более эффективного результата.

Строчка

reduction(max:localMax)

означает, что переменную localMax нужно использовать для редукции через функцию max. После выхода из параллельной секции в переменной localMax действительно будет лежать правильное значение. Можно было обойтись без reduction, дописав критическую секцию в параллельный регион.

**Результаты**

**Общий размер задачи 30 х 30 х 30**

|  |  |  |  |
| --- | --- | --- | --- |
| **Time** | **gridX** | **gridY** | **gridZ** |
| 15.263869 | 1 | 1 | 1 |
| 27.128401 | 1 | 1 | 2 |
| 58.387280 | 1 | 2 | 2 |
| 88.006780 | 3 | 2 | 1 |
| 127.903893 | 2 | 2 | 2 |

**Общий размер задачи 40 х 40 х 40**

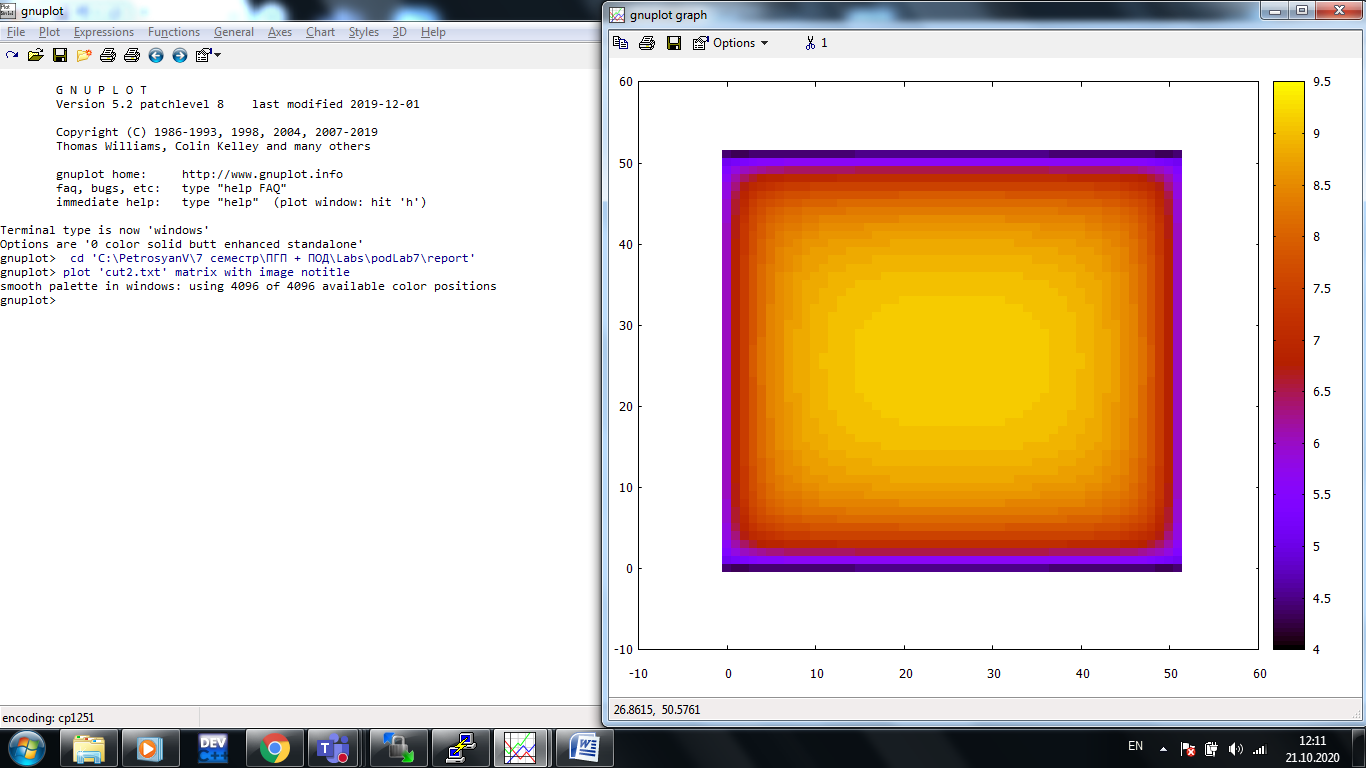
|  |  |  |  |
| --- | --- | --- | --- |
| **Time** | **gridX** | **gridY** | **gridZ** |
| 32.332243 | 1 | 1 | 1 |
| 52.342230 | 1 | 1 | 2 |
| 108.270650 | 1 | 2 | 2 |

**Общий размер задачи 52 х 52 х 52**

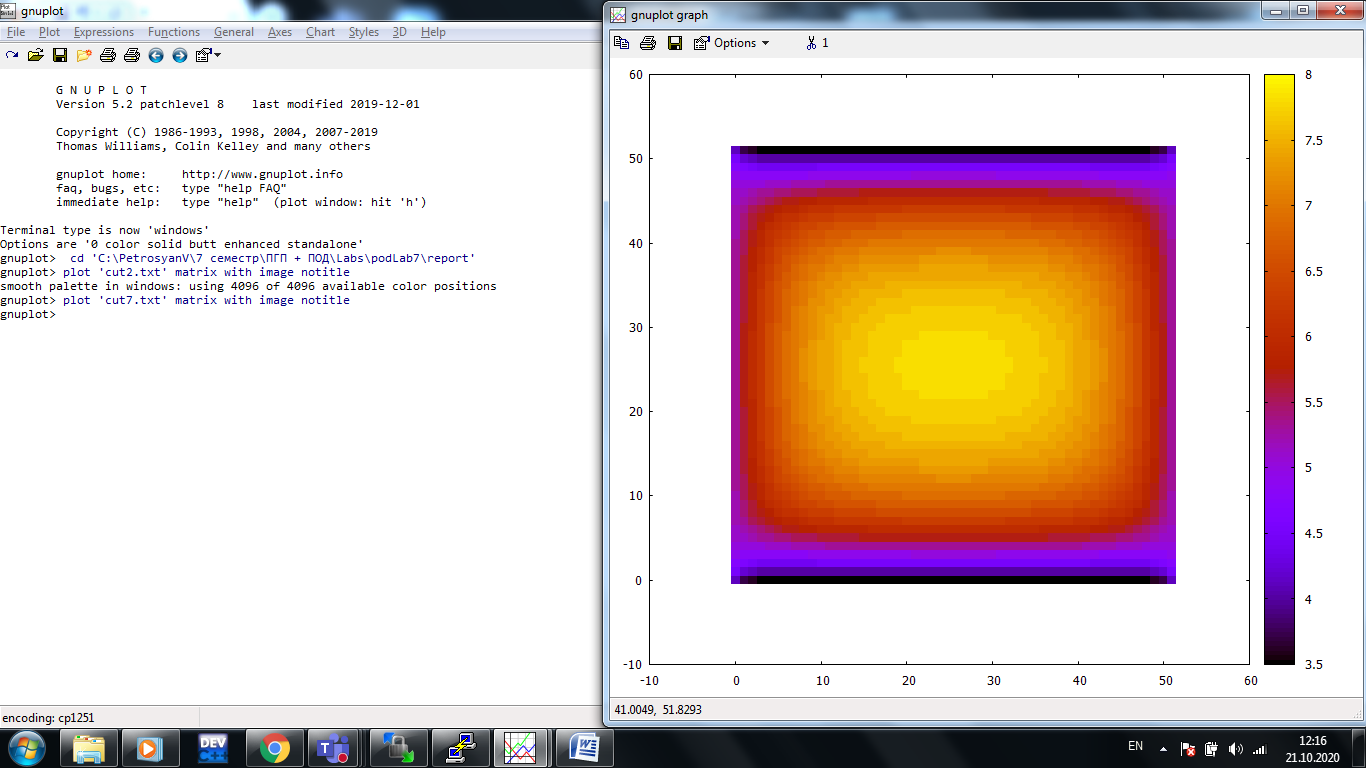
|  |  |  |  |
| --- | --- | --- | --- |
| **Time** | **gridX** | **gridY** | **gridZ** |
| 86.367333 | 1 | 1 | 2 |
| 166.529544 | 1 | 2 | 2 |

**Температурный срез для задачи 52 х 52 х 52**

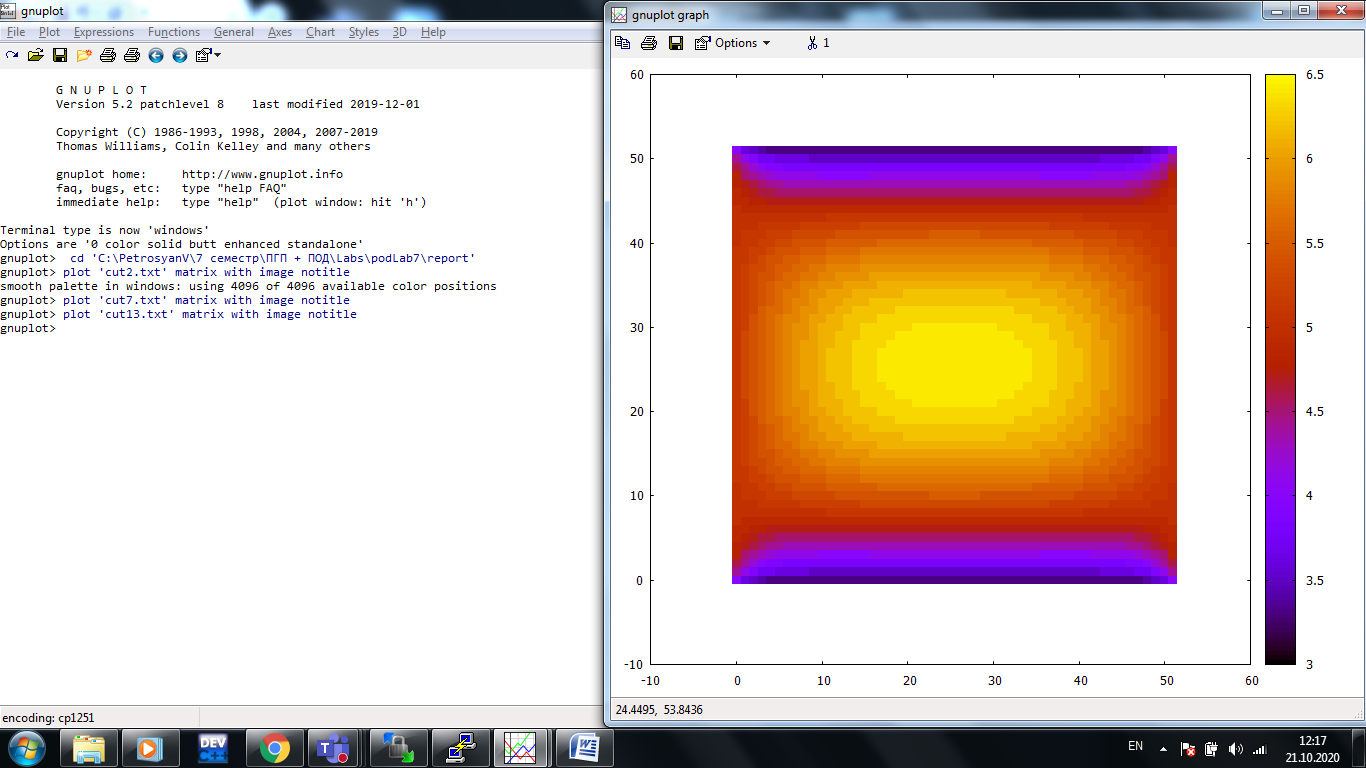
* Для z = 2

****

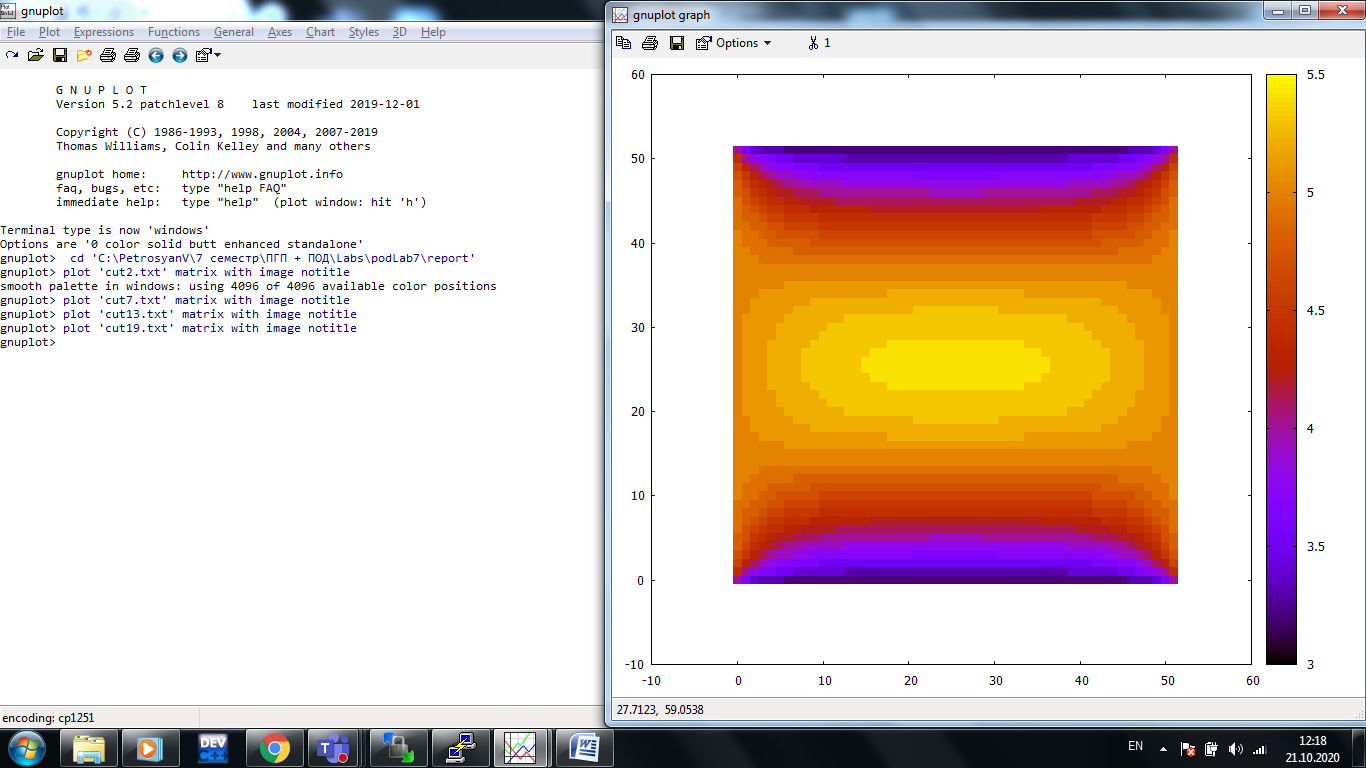
* Для z = 7

****

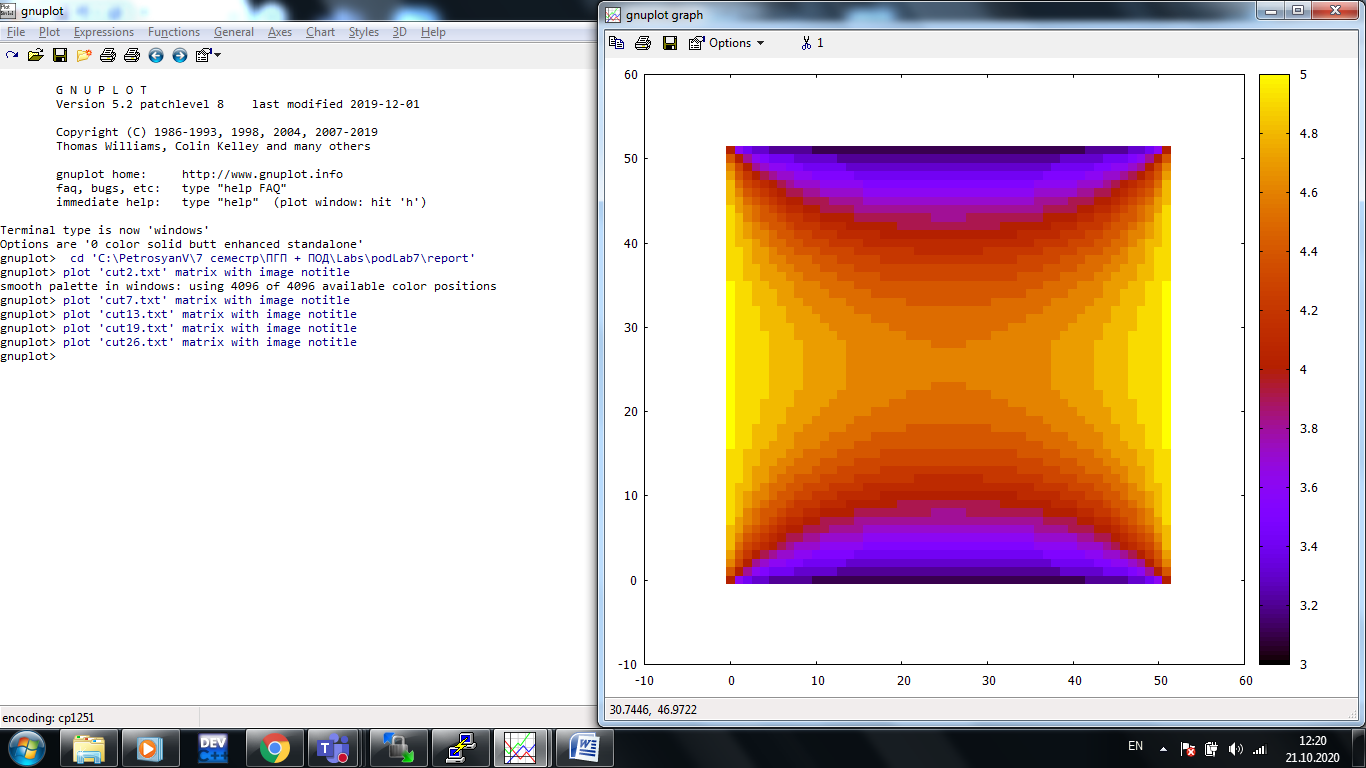
* Для z = 13

****

* Для z = 19



* Для z = 25



**Выводы**

Сложность в программировании данной лр не испытал. Весь код был написан за 1 час.

Если посмотреть на результаты замеров, то может показаться, что я написал плохую программу, которая работает слишком долго, неэффективно и при увеличении количества процессов становится только хуже. Дело в том, что я запускал на одном компьютере, а не на кластере. Каждый процесс запрашивал максимальное количество потоков через openMP, что замедляло программу. Можно заметить, что если поделить время работы программы с Х процессами на время работы программы с 1 процессом, то получим число близкое к Х, что доказывает мои доводы.

Технология openMP не требует глубоких знаний и легка в применении. Про MPI я бы такое не сказал. Считаю, что технологию openMP нужно знать всем программистам на С++ для быстрых оптимизаций прикладных решений в экстренных ситуациях.