МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

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

Лабораторная работа №1
по курсу «Параллельная обработка данных»
Message passing interface (MPI)

Выполнил: М.А. Жерлыгин
Группа: 8О-408Б
Преподаватели: К.Г. Крашенинников,
А.Ю. Морозов

Москва, 2021

Условие

Цель работы: Знакомство с технологией MPI. Реализация метода Якоби. Решение задачи Дирихле для уравнения Лапласа в трехмерной области с граничными условиями первого рода.

Вариант 3. обмен граничными слоями через sendrecv, контроль сходимости allgather;

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:

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

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

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

```
if (block[x_on] > 1) {
   if (i_b == 0) {
       for (int k = 0; k < dim[z_on]; k++) {
           for (int j = 0; j < dim[y_on]; j++) {
               send[_ib(j, k)] = values[_i(dim[x_on] - 1, j, k)];
           }
       }
       MPI_Sendrecv(send, buffer_size, MPI_DOUBLE, _ip(i_b + 1, j_b, k_b),
                    id, recv, buffer_size, MPI_DOUBLE, _ip(i_b + 1, j_b, k_b),
                    _ip(i_b + 1, j_b, k_b), MPI_COMM_WORLD, &status);
       for (int k = 0; k < dim[z_on]; k++) {
           for (int j = 0; j < dim[y_on]; j++) {
               values[_i(dim[x_on], j, k)] = recv[_ib(j, k)];
           }
       }
   } else if (i_b + 1 == block[x_on]) {
       for (int k = 0; k < dim[z_on]; k++) {
           for (int j = 0; j < dim[y_on]; j++) {
               send[_ib(j, k)] = values[_i(0, j, k)];
           }
       }
       MPI_Sendrecv(send, buffer_size, MPI_DOUBLE, _ip(i_b - 1, j_b, k_b),
                    id, recv, buffer_size, MPI_DOUBLE, _ip(i_b - 1, j_b, k_b),
                    _ip(i_b - 1, j_b, k_b), MPI_COMM_WORLD, &status);
       for (int k = 0; k < dim[z_on]; k++) {
           for (int j = 0; j < dim[y_on]; j++) {
               values[_i(-1, j, k)] = recv[_ib(j, k)];
           }
       }
   } else {
       for (int k = 0; k < dim[z_on]; k++) {
           for (int j = 0; j < dim[y_on]; j++) {
               send[_ib(j, k)] = values[_i(dim[x_on] - 1, j, k)];
           }
       }
       MPI_Sendrecv(send, buffer_size, MPI_DOUBLE, _ip(i_b + 1, j_b, k_b),
                    id, recv, buffer_size, MPI_DOUBLE, _ip(i_b - 1, j_b, k_b),
                    _ip(i_b - 1, j_b, k_b), MPI_COMM_WORLD, &status);
       for (int k = 0; k < dim[z_on]; k++) {
           for (int j = 0; j < dim[y_on]; j++) {
               values[_i(-1, j, k)] = recv[_ib(j, k)];
               send[_ib(j, k)] = values[_i(0, j, k)];
           }
       }
       MPI_Sendrecv(send, buffer_size, MPI_DOUBLE, _ip(i_b - 1, j_b, k_b),
```

```
                    id, recv, buffer_size, MPI_DOUBLE, _ip(i_b + 1, j_b, k_b),
                    _ip(i_b + 1, j_b, k_b), MPI_COMM_WORLD, &status);
        for (int k = 0; k < dim[z_on]; k++) {
            for (int j = 0; j < dim[y_on]; j++) {
                values[_i(dim[x_on], j, k)] = recv[_ib(j, k)];
            }
        }
    }
}

if (block[y_on] > 1) {
    if (j_b == 0) {
        for (int k = 0; k < dim[z_on]; k++) {
            for (int i = 0; i < dim[x_on]; i++) {
                send[_ib(i, k)] = values[_i(i, dim[y_on] - 1, k)];
            }
        }
        MPI_Sendrecv(send, buffer_size, MPI_DOUBLE, _ip(i_b, j_b + 1, k_b),
                    id, recv, buffer_size, MPI_DOUBLE, _ip(i_b, j_b + 1, k_b),
                    _ip(i_b, j_b + 1, k_b), MPI_COMM_WORLD, &status);
        for (int k = 0; k < dim[z_on]; k++) {
            for (int i = 0; i < dim[x_on]; i++) {
                values[_i(i, dim[y_on], k)] = recv[_ib(i, k)];
            }
        }
    }
    else if (j_b + 1 == block[y_on]) {
        for (int k = 0; k < dim[z_on]; k++) {
            for (int i = 0; i < dim[x_on]; i++) {
                send[_ib(i, k)] = values[_i(i, 0, k)];
            }
        }
        MPI_Sendrecv(send, buffer_size, MPI_DOUBLE, _ip(i_b, j_b - 1, k_b),
                    id, recv, buffer_size, MPI_DOUBLE, _ip(i_b, j_b - 1, k_b),
                    _ip(i_b, j_b - 1, k_b), MPI_COMM_WORLD, &status);
        for (int k = 0; k < dim[z_on]; k++) {
            for (int i = 0; i < dim[x_on]; i++) {
                values[_i(i, -1, k)] = recv[_ib(i, k)];
            }
        }
    }
    else {
        for (int k = 0; k < dim[z_on]; k++) {
            for (int i = 0; i < dim[x_on]; i++) {
                send[_ib(i, k)] = values[_i(i, dim[y_on] - 1, k)];
            }
        }
        MPI_Sendrecv(send, buffer_size, MPI_DOUBLE, _ip(i_b, j_b + 1, k_b),
                    id, recv, buffer_size, MPI_DOUBLE, _ip(i_b, j_b - 1, k_b),
                    _ip(i_b, j_b - 1, k_b), MPI_COMM_WORLD, &status);
        for (int k = 0; k < dim[z_on]; k++) {
            for (int i = 0; i < dim[x_on]; i++) {
                values[_i(i, -1, k)] = recv[_ib(i, k)];
                send[_ib(i, k)] = values[_i(i, 0, k)];
            }
        }
        MPI_Sendrecv(send, buffer_size, MPI_DOUBLE, _ip(i_b, j_b - 1, k_b),
                    id, recv, buffer_size, MPI_DOUBLE, _ip(i_b, j_b + 1, k_b),
                    _ip(i_b, j_b + 1, k_b), MPI_COMM_WORLD, &status);
        for (int k = 0; k < dim[z_on]; k++) {
            for (int i = 0; i < dim[x_on]; i++) {
                values[_i(i, dim[y_on], k)] = recv[_ib(i, k)];
            }
        }
    }
}
```

```
if (block[z_on] > 1) {
   if (k_b == 0) {
       for (int j = 0; j < dim[y_on]; j++) {
           for (int i = 0; i < dim[x_on]; i++) {
               send[_ib(i, j)] = values[_i(i, j, dim[z_on] - 1)];
           }
       }
       MPI_Sendrecv(send, buffer_size, MPI_DOUBLE, _ip(i_b, j_b, k_b + 1),
                    id, recv, buffer_size, MPI_DOUBLE, _ip(i_b, j_b, k_b + 1),
                    _ip(i_b, j_b, k_b + 1), MPI_COMM_WORLD, &status);
       for (int j = 0; j < dim[y_on]; j++) {
           for (int i = 0; i < dim[x_on]; i++) {
               values[_i(i, j, dim[z_on])] = recv[_ib(i, j)];
           }
       }
   }
   else if (k_b + 1 == block[z_on]) {
       for (int j = 0; j < dim[y_on]; j++) {
           for (int i = 0; i < dim[x_on]; i++) {
               send[_ib(i, j)] = values[_i(i, j, 0)];
           }
       }
       MPI_Sendrecv(send, buffer_size, MPI_DOUBLE, _ip(i_b, j_b, k_b - 1),
                    id, recv, buffer_size, MPI_DOUBLE, _ip(i_b, j_b, k_b - 1),
                    _ip(i_b, j_b, k_b - 1), MPI_COMM_WORLD, &status);
       for (int j = 0; j < dim[y_on]; j++) {
           for (int i = 0; i < dim[x_on]; i++) {
               values[_i(i, j, -1)] = recv[_ib(i, j)];
           }
       }
   }
   else {
       for (int j = 0; j < dim[y_on]; j++) {
           for (int i = 0; i < dim[x_on]; i++) {
               send[_ib(i, j)] = values[_i(i, j, dim[z_on] - 1)];
           }
       }
       MPI_Sendrecv(send, buffer_size, MPI_DOUBLE, _ip(i_b, j_b, k_b + 1),
                    id, recv, buffer_size, MPI_DOUBLE, _ip(i_b, j_b, k_b - 1),
                    _ip(i_b, j_b, k_b - 1), MPI_COMM_WORLD, &status);
       for (int j = 0; j < dim[y_on]; j++) {
           for (int i = 0; i < dim[x_on]; i++) {
               values[_i(i, j, -1)] = recv[_ib(i, j)];
               send[_ib(i, j)] = values[_i(i, j, 0)];
           }
       }
       MPI_Sendrecv(send, buffer_size, MPI_DOUBLE, _ip(i_b, j_b, k_b - 1),
                    id, recv, buffer_size, MPI_DOUBLE, _ip(i_b, j_b, k_b + 1),
                    _ip(i_b, j_b, k_b + 1), MPI_COMM_WORLD, &status);
       for (int j = 0; j < dim[y_on]; j++) {
           for (int i = 0; i < dim[x_on]; i++) {
               values[_i(i, j, dim[z_on])] = recv[_ib(i, j)];
           }
       }
   }
}
```

Затем вычисляю локальную (в рамках процесса) погрешность.

```
double diff = 0.0;
double h2[3];
h2[x_on] = h[x_on] * h[x_on];
h2[y_on] = h[y_on] * h[y_on];
h2[z_on] = h[z_on] * h[z_on];
```

```
for (int i = 0; i < dim[x_on]; i++) {
   for (int j = 0; j < dim[y_on]; j++) {
       for (int k = 0; k < dim[z_on]; k++) {
              next_values[_i(i, j, k)] = 0.5 * ((values[_i(i + 1, j, k)] +
values[_i(i - 1, j, k)]) / h2[x_on] +
                                                (values[_i(i, j + 1, k)] +
values[_i(i, j - 1, k)]) / h2[y_on] +
                                                (values[_i(i, j, k + 1)] +
values[_i(i, j, k - 1)]) / h2[z_on]) /
                                            (1 / h2[x_on] + 1 / h2[y_on] + 1
/ h2[z_on]);
             diff = max(abs(next_values[_i(i, j, k)] - values[_i(i, j, k)]),
diff);
       }
   }
}
```

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 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.

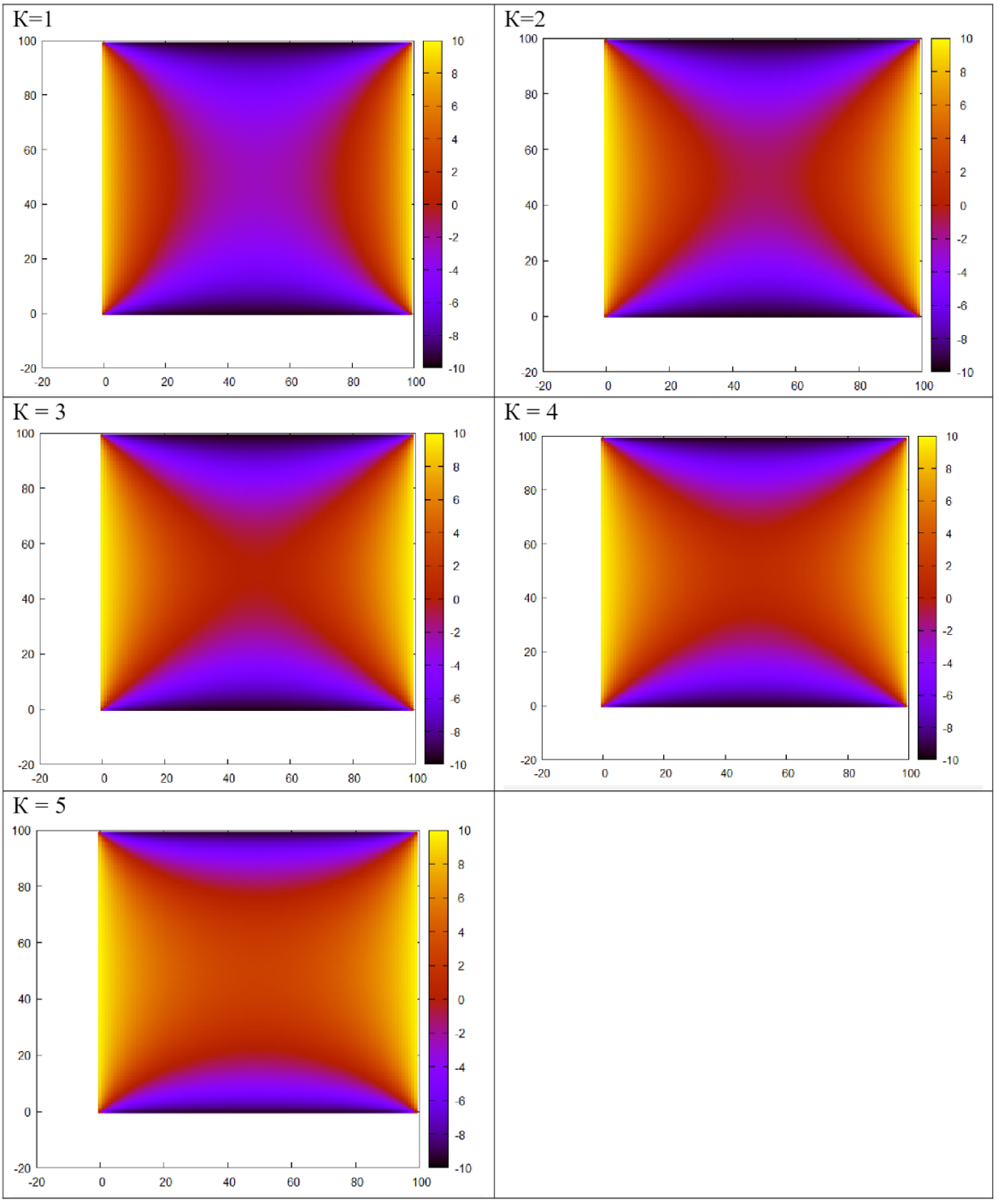
MPI_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)
sendbuf - указатель на буфер, sendcount - сколько слов отправляется, sendtype - тип отправленного (int, double), recvbuf - куда будем принимать, recvcount - сколько слов мы принимаем от одного другого процесса, включая этот, recvtype - тип полученного (int, double), comm - коммуникатор.

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

Результаты

Размер блоков / расчета	**time**
1 1 1 | 30 30 30	17861ms
2 2 2 | 15 15 15	4621ms
2 2 5 | 15 15 6	5938ms

Можно заметить, что использование нескольких процессов заметно ускоряет процесс вычисления.



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 научила меня работе с технологией межпроцессорного взаимодействия на больших вычислительных кластерах — MPI.

После выполнения данной лабораторной работы я убедился, что параллельная обработка данных с несколькими процессами происходит гораздо быстрее, даже несмотря на пересылки данных.