МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Компьютерные науки и прикладная математика»
Кафедра 806 «Вычислительная математика и программирование»

Лабораторная работа №1
по курсу «Параллельная обработка данных»

Message Passing Interface (MPI).

Выполнил: А.С. Федоров
Группа: 8О-407Б
Преподаватели: К.Г. Крашенинников,
А.Ю. Морозов

Москва, 2022

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 задания. 1. Обмен граничными слоями через send/receive, контроль сходимости allgather.

Программное и аппаратное обеспечение

Графический процессор

Название: NVIDIA GeForce GTX 1050
Compute capability: 6.1
Объем графической памяти: 2147352576 байтов
Объем разделяемой памяти на блок: 49152 байтов
Объем регистров на блок: 65536
Размер варпа: 32
Максимальное количество потоков на блок: (1024, 1024, 64)
Максимальное число блоков: (2147483647, 65535, 65535)
Объем постоянной памяти: 65536 байтов
Число мультипроцессоров: 5

Процессор

Название: i5-8250U
Базовая тактовая частота процессора: 1,60 ГГц
Количество ядер: 4
Количество потоков: 8
Кеш L1: 256 Кб
Кеш L2: 1 Мб
Кеш L3: 6 Мб

Оперативная память

Тип: DDR4
Объем: 11.9 Гб
Частота: 2400 МГц

Програмное обеспечение

ОС: WSL2 (Windows 11)
IDE: Microsoft Visual Studio 2022 (аддон NVIDIA Nsight)
Компилятор: nvcc

Метод решения

Ращение задачи Дирихле методом Якоби возможно выполнять параллельно в несколько процессов. Дискретизировав задачу, можно разбить расчетную сетку на одинаковые блоки и делегировать их обсчет разным, параллельно выполняющимся, процессам. Граничными условиями для отдельного блока на текущей итерации будут являться значения на соприкасающихся границах блоков-соседей данного. Как только максимальное расхождение значений в узлах между текущей и следующей итерацией становится меньше определенного заданного значения, процесс останавливается.

Описание программы

Для вычисления следующего шага, программа сначала обновляет значения на границах блоков. Выполняется это в два этапа: отправка вправо, вперед, вверх и получения слева, сзади, снизу, синхронизация затем отправка влево, назад, вниз и получение справа, спереди, сверху, синхронизация. После обновления границ выполняется итерация (параллельно для каждого процесса), синхронизация и проверка условия остановки. Условие остановки проверяется параллельным подсчетом максимума расхождения для каждого блока и сборки всех значений в единый массив с помощью MPI_Allgather. На данном массиве ищется максимум и сверяется с условием остановки.

Основной цикл параллельного подсчета и обмена данными между процессами:

```
while(max_eps > eps){
        MPI_Barrier(MPI_COMM_WORLD);
        // отправка (right. back, up)
            // right
            if (xb < xnb-1){
                for (int z = 0; z < nz; ++z){
                    for (int y = 0; y < ny; ++y){
                        send_left_right_buff[z*ny+y] = data[_i(nx-1, y, z)];
                    }
                }
                MPI_Send(send_left_right_buff, nz*ny, MPI_DOUBLE, _ib(xb+1, yb, zb),
id, MPI_COMM_WORLD);
            }
            // back
            if (yb < ynb-1){
                for (int z = 0; z < nz; ++z){
                    for (int x = 0; x < nx; ++x){
                        send_front_back_buff[z*nx+x] = data[_i(x, ny-1, z)];
                    }
                }
```

```
                MPI_Send(send_front_back_buff, nz*nx, MPI_DOUBLE, _ib(xb, yb+1, zb),
id, MPI_COMM_WORLD);
            }
            // up
            if (zb < znb-1){
                for (int y = 0; y < ny; ++y){
                    for (int x = 0; x < nx; ++x){
                        send_up_down_buff[y*nx+x] = data[_i(x, y, nz-1)];
                    }
                }
                MPI_Send(send_up_down_buff, ny*nx, MPI_DOUBLE, _ib(xb, yb, zb+1), id,
MPI_COMM_WORLD);
            }
        // npuem (left, front, down)
            // left
            if (xb > 0){
                // std::cout << id << "left\n";
                MPI_Recv(recv_left_right_buff, nz*ny, MPI_DOUBLE, _ib(xb-1, yb, zb),
_ib(xb-1, yb, zb), MPI_COMM_WORLD, &status);
                for (int z = 0; z < nz; ++z){
                    for (int y = 0; y < ny; ++y){
                        data[_i(-1, y, z)] = recv_left_right_buff[z*ny+y];
                    }
                }
            } else {
                for (int z = 0; z < nz; ++z){
                    for (int y = 0; y < ny; ++y){
                        data[_i(-1, y, z)] = bc_left;
                    }
                }
            }
            // front
            if (yb > 0){
                // std::cout << id << "front\n";
                MPI_Recv(recv_front_back_buff, nz*nx, MPI_DOUBLE, _ib(xb, yb-1, zb),
_ib(xb, yb-1, zb), MPI_COMM_WORLD, &status);
                for (int z = 0; z < nz; ++z){
                    for (int x = 0; x < nx; ++x){
                        data[_i(x, -1, z)] = recv_front_back_buff[z*nx+x];
                    }
                }
            } else {
                for (int z = 0; z < nz; ++z){
                    for (int x = 0; x < nx; ++x){
                        data[_i(x, -1, z)] = bc_front;
```

```
                    }
                }
            }
            // down
            if (zb > 0){
                // std::cout << id << "down\n";
                MPI_Recv(recv_up_down_buff, ny*nx, MPI_DOUBLE, _ib(xb, yb, zb-1),
_ib(xb, yb, zb-1), MPI_COMM_WORLD, &status);
                for (int y = 0; y < ny; ++y){
                    for (int x = 0; x < nx; ++x){
                        data[_i(x, y, -1)] = recv_up_down_buff[y*nx+x];
                    }
                }
            } else {
                for (int y = 0; y < ny; ++y){
                    for (int x = 0; x < nx; ++x){
                        data[_i(x, y, -1)] = bc_down;
                    }
                }
            }
        MPI_Barrier(MPI_COMM_WORLD);
        // отправка (left, front, down)
            // left
            if (xb > 0){
                for (int z = 0; z < nz; ++z){
                    for (int y = 0; y < ny; ++y){
                        send_left_right_buff[z*ny+y] = data[_i(0, y, z)];
                    }
                }
                MPI_Send(send_left_right_buff, nz*ny, MPI_DOUBLE, _ib(xb-1, yb, zb),
id, MPI_COMM_WORLD);
            }
            // front
            if (yb > 0){
                for (int z = 0; z < nz; ++z){
                    for (int x = 0; x < nx; ++x){
                        send_front_back_buff[z*nx+x] = data[_i(x, 0, z)];
                    }
                }
                MPI_Send(send_front_back_buff, nz*nx, MPI_DOUBLE, _ib(xb, yb-1, zb),
id, MPI_COMM_WORLD);
            }
            // down
            if (zb > 0){
                for (int y = 0; y < ny; ++y){
```

```
                    for (int x = 0; x < nx; ++x){
                        send_up_down_buff[y*nx+x] = data[_i(x, y, 0)];
                    }
                }
                MPI_Send(send_up_down_buff, ny*nx, MPI_DOUBLE, _ib(xb, yb, zb-1), id,
MPI_COMM_WORLD);
            }
        // npuem (right, back, up)
            // right
            if (xb < xnb-1){
                MPI_Recv(recv_left_right_buff, nz*ny, MPI_DOUBLE, _ib(xb+1, yb, zb),
_ib(xb+1, yb, zb), MPI_COMM_WORLD, &status);
                for (int z = 0; z < nz; ++z){
                    for (int y = 0; y < ny; ++y){
                        data[_i(nx, y, z)] = recv_left_right_buff[z*ny+y];
                    }
                }
            } else {
                for (int z = 0; z < nz; ++z){
                    for (int y = 0; y < ny; ++y){
                        data[_i(nx, y, z)] = bc_right;
                    }
                }
            }
            // back
            if (yb < ynb-1){
                MPI_Recv(recv_front_back_buff, nz*nx, MPI_DOUBLE, _ib(xb, yb+1, zb),
_ib(xb, yb+1, zb), MPI_COMM_WORLD, &status);
                for (int z = 0; z < nz; ++z){
                    for (int x = 0; x < nx; ++x){
                        data[_i(x, ny, z)] = recv_front_back_buff[z*nx+x];
                    }
                }
            } else {
                for (int z = 0; z < nz; ++z){
                    for (int x = 0; x < nx; ++x){
                        data[_i(x, ny, z)] = bc_back;
                    }
                }
            }
            // up
            if (zb < znb-1){
                MPI_Recv(recv_up_down_buff, ny*nx, MPI_DOUBLE, _ib(xb, yb, zb+1),
_ib(xb, yb, zb+1), MPI_COMM_WORLD, &status);
                for (int y = 0; y < ny; ++y){
```

```
                    for (int x = 0; x < nx; ++x){
                        data[_i(x, y, nz)] = recv_up_down_buff[y*nx+x];
                    }
                }
            } else {
                for (int y = 0; y < ny; ++y){
                    for (int x = 0; x < nx; ++x){
                        data[_i(x, y, nz)] = bc_up;
                    }
                }
            }
        MPI_Barrier(MPI_COMM_WORLD);

        // итерация
        double eps = -10000000.0;
        for (int z = 0; z < nz; ++z){
            for (int y = 0; y < ny; ++y){
                for (int x = 0; x < nx; ++x){
                    next_data[_i(x, y, z)] = ((data[_i(x+1, y, z)]+data[_i(x-1, y,
z)])/(hx*hx) +
                                              (data[_i(x, y+1, z)]+data[_i(x, y-1,
z)])/(hy*hy) +
                                              (data[_i(x, y, z+1)]+data[_i(x, y, z-
1)])/(hz*hz)) /
                                              (2.0*((1/(hx*hx))+(1/(hy*hy))+(1/(hz*hz
))));
                    eps = std::max(eps, std::abs(next_data[_i(x, y, z)]-data[_i(x, y,
z)]));
                }
            }
        }

        MPI_Barrier(MPI_COMM_WORLD);
        MPI_Allgather(&eps, 1, MPI_DOUBLE, eps_buff, 1, MPI_DOUBLE, MPI_COMM_WORLD);

        max_eps = -10000000.0;
        for (int i = 0; i < znb*ynb*xnb; ++i){
            max_eps = std::max(max_eps, eps_buff[i]);
        }

        // обмен указателями
        double* tmp = data;
        data = next_data;
        next_data = tmp;
    }
```

Тест быстродействия (в миллисекундах, порог сходимости: 0.0001)

Размер сетки	Число процессов			
	1	**2**	**4**	**8**
1000	6.851	24.064	**6.667**	7.203
8000	165.033	104.732	**92.471**	113.995
27000	985.684	526.734	**332.417**	506.371
64000	3773.856	1992.764	**1159.812**	1523.108
125000	10161.430	5282.874	**3479.461**	3552.599

Выводы

Решение задачи Дирихле может возникать при анализе распределения температур, скоростных полей или магнитных и электрических полей в задачах проектирования. Распараллеливание решения данной задачи может существенно ускорить вычисление результата, так как ввиду специфики задачи, она хорошо распараллеливается. Реализация алгоритма решения в рамках одного блока не вызвала трудностей. Основную сложность вызвала организация обмена данными между блоками и синхронизация их работы. Исходя из замеров быстродействия видно, что лучший результат программа показывает на количестве процессов 4, что логично, так как у модели процессора, на котором производилось тестирование имеется 4 физических ядра. Распараллеливание на большее число потоков не имеет смысла, так как фактически избыточные процессы будут выполняться последовательно, прерывая друг друга на ядрах.