МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

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

Лабораторная работа №7
по курсу «Параллельная обработка данных»

Message Passing Interface (MPI)

Студент: А. А. Садаков
Группа: 8О-406Б
Преподаватели: К.Г. Крашенинников,
А.Ю. Морозов

Москва, 2022

Условие

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

Вариант 5: обмен граничными слоями через send/receive, контроль сходимости allreduce.

Программное и аппаратное обеспечение

ГРАФИЧЕСКИЙ ПРОЦЕССОР
Имя устройства: NVIDIA GeForce GTX 1650
Compute capability: 7.5
Размер графической памяти: 4242604032
Размер разделяемой памяти: 49152
Размер константной памяти: 65536
Максимальное количество регистров на блок: 65536
Максимальное количество потоков на блок: 1024
Количество мультипроцессоров: 14

ПРОЦЕССОР
Имя устройства: Intel Core I5-10300H
Архитектура: Comet Lake-H
Количество ядер (потоков): 4(8)
Базовая (максимальная) частота: 2,5(4,5)ГГц
Кеш 1-го уровня: 64Кб (на ядро)
Кеш 2-го уровня: 256Кб (на ядро)
Кеш 3-го уровня: 6Мб (всего)

ОПЕРАТИВНАЯ ПАМЯТЬ
Объём: 8Гб
Тип: DDR4
Частота:2933МГц

ЖЁСТКИЙ ДИСК
Имя устройства: Intel SSDPEKNW512G8L
Тип диска: SSD
Объём памяти: 512Гб
Скорость чтения/записи: 1500/1000 Мб/с

ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ
OS: Windows 10,Подсистема Ubuntu 20.04
IDE: Visual Studio Code
Компилятор: nvcc

Метод решения

Каждый процесс отвечает за свою область, на которые разбита регулярная сетка. Каждая итерация состоит из 3-х этапов: обмен граничными слоями между процессами, обновление значений во всех ячейках и вычисление погрешности (локально и глобально).

Описание программы

ОСновные функции:

- $iteration1$,
- $iteration2$,
- $inputTaskInfo$

При помощи функции $iteration1$ происходит обмен граничными слоями между процессами.

При помощи функции $iteration2$ происходит обновление значений во всех ячейках.

Функции $iteration1$ и $iteration2$ используют MPI_Send и MPI_Recv для обмена сообщений между процессами.

Функция $inputTaskInfo$ нужна для ввода данных.

Решение задачи реализованно в виде класса. Вызов MPI_Init происходит в конструкторе, $MPI_Finalize$ — в деструкторе.

Итерации проходят в цикле пока не будет достигнута заданная точность.

Код функций:

```
1 void DirihleTask::inputTaskInfo () {
2     if (id == 0) {
```

```
3             std::cin >> block[X] >> block[Y] >> block[Z];
4             std::cin >> dimension[X] >> dimension[Y] >> dimension[Z];
5             std::cin >> output;
6             std::cin >> eps;
7             std::cin >> l[X] >> l[Y] >> l[Z];
8             std::cin >> u[DOWN] >> u[UP];
9             std::cin >> u[LEFT] >> u[RIGHT];
10            std::cin >> u[FRONT] >> u[BACK];
11            std::cin >> u0;
12        }
13
14        MPI_Bcast(dimension, 3, MPI_INT, 0, MPI_COMM_WORLD);
15        MPI_Bcast(block, 3, MPI_INT, 0, MPI_COMM_WORLD);
16        MPI_Bcast(&eps, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
17        MPI_Bcast(&l[0], l.size(), MPI_DOUBLE, 0, MPI_COMM_WORLD);
18        MPI_Bcast(&u[0], u.size(), MPI_DOUBLE, 0, MPI_COMM_WORLD);
19        MPI_Bcast(&u0, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
20    }
21
22    void DirihleTask::iteration1 (int ax, const std::vector<int> &b, std::vector<
          double> &buff, std::vector<double> &data) {
23        int *dim = dimension;
24        int d[] = {0, 0, 0};
25        d[ax] = 1;
26        int qq[2];
27        qq[ax % 2] = (ax + COUNT_OF_AXES - 1) % COUNT_OF_AXES;
28        qq[(ax + 1) % 2] = (ax + (COUNT_OF_AXES - 1) * 2) %
              COUNT_OF_AXES;
29        int dir = ax * 2;
30        if (b[ax] < block[ax] - 1) {
31            for (uint64_t k = 0; k < dim[qq[0]]; ++k) {
32                for (uint64_t j = 0; j < dim[qq[1]]; ++j) {
33                    int coeff[3];
34                    uint64_t tmp[] = {j, k};
35                    for (uint64_t l = 0; l < ax; ++l) {
36                        coeff[l] = tmp[l];
37                    }
38                    coeff[ax] = dim[ax] - 1;
```

```
39                  for (uint64_t l = ax + 1; l < COUNT_OF_AXES; ++l) {
40                      coeff[l] = tmp[l − 1];
41                  }
42                  buff[k ∗ dim[qq[1]] + j] = data[index(coeff[0], coeff[1],
                        coeff[2], dim)];
43              }
44          }
45          MPI_Send(&buff[0], dim[qq[0]] ∗ dim[qq[1]], MPI_DOUBLE,
                indexBlock(b[X] + d[X], b[Y] + d[Y], b[Z] + d[Z], block), id,
                MPI_COMM_WORLD);
46      }
47
48      if (b[ax] > 0) {
49          MPI_Recv(&buff[0], dim[qq[0]] ∗ dim[qq[1]], MPI_DOUBLE,
                indexBlock(b[X] − d[X], b[Y] − d[Y], b[Z] − d[Z], block),
                indexBlock(b[X] − d[X], b[Y] − d[Y], b[Z] − d[Z], block),
                MPI_COMM_WORLD, &status);
50          for (uint64_t k = 0; k < dim[qq[0]]; ++k) {
51              for (uint64_t j = 0; j < dim[qq[1]]; ++j) {
52                  int coeff[3];
53                  uint64_t tmp[] = {j, k};
54                  for (uint64_t l = 0; l < ax; ++l) {
55                      coeff[l] = tmp[l];
56                  }
57                  coeff[ax] = −1;
58                  for (uint64_t l = ax + 1; l < COUNT_OF_AXES; ++l) {
59                      coeff[l] = tmp[l − 1];
60                  }
61                  data[index(coeff[0], coeff[1], coeff[2], dim)] = buff[k ∗ dim[
                        qq[1]] + j];
62              }
63          }
64      } else {
65          for (uint64_t k = 0; k < dim[qq[0]]; ++k) {
66              for (uint64_t j = 0; j < dim[qq[1]]; ++j) {
67                  int coeff[3];
68                  uint64_t tmp[] = {j, k};
69                  for (uint64_t l = 0; l < ax; ++l) {
```

```
70                          coeff[l] = tmp[l];
71                      }
72                      coeff[ax] = -1;
73                      for (uint64_t l = ax + 1; l < COUNT_OF_AXES; ++l) {
74                          coeff[l] = tmp[l - 1];
75                      }
76                      data[index(coeff[0], coeff[1], coeff[2], dim)] = u[dir];
77                  }
78              }
79          }
80  }
81
82  void DirihleTask::iteration2 (int ax, const std::vector<int> &b, std::vector<
        double> &buff, std::vector<double> &data) {
83      int *dim = dimension;
84      int d[] = {0, 0, 0};
85      d[ax] = 1;
86      int qq[2];
87      qq[ax % 2] = (ax + COUNT_OF_AXES - 1) % COUNT_OF_AXES;
88      qq[(ax + 1) % 2] = (ax + (COUNT_OF_AXES - 1) * 2) %
            COUNT_OF_AXES;
89      int dir = 1 + ax * 2;
90      if (b[ax] > 0) {
91          for (uint64_t k = 0; k < dim[qq[0]]; ++k) {
92              for (uint64_t j = 0; j < dim[qq[1]]; ++j) {
93                  int coeff[3];
94                  uint64_t tmp[] = {j, k};
95                  for (uint64_t l = 0; l < ax; ++l) {
96                      coeff[l] = tmp[l];
97                  }
98                  coeff[ax] = 0;
99                  for (uint64_t l = ax + 1; l < COUNT_OF_AXES; ++l) {
100                     coeff[l] = tmp[l - 1];
101                 }
102                 buff[k * dim[qq[1]] + j] = data[index(coeff[0], coeff[1],
                        coeff[2], dim)];
103             }
104         }
```

```
105             MPI_Send(&buff[0], dim[qq[0]] * dim[qq[1]], MPI_DOUBLE,
                    indexBlock(b[X] - d[X], b[Y] - d[Y], b[Z] - d[Z], block), id,
                    MPI_COMM_WORLD);
106         }
107
108         if (b[ax] < block[ax] - 1) {
109             MPI_Recv(&buff[0], dim[qq[0]] * dim[qq[1]], MPI_DOUBLE,
                    indexBlock(b[X] + d[X], b[Y] + d[Y], b[Z] + d[Z], block),
                    indexBlock(b[X] + d[X], b[Y] + d[Y], b[Z] + d[Z], block),
                    MPI_COMM_WORLD, &status);
110             for (uint64_t k = 0; k < dim[qq[0]]; ++k) {
111                 for (uint64_t j = 0; j < dim[qq[1]]; ++j) {
112                     int coeff[3];
113                     uint64_t tmp[] = {j, k};
114                     for (uint64_t l = 0; l < ax; ++l) {
115                         coeff[l] = tmp[l];
116                     }
117                     coeff[ax] = dim[ax];
118                     for (uint64_t l = ax + 1; l < COUNT_OF_AXES; ++l) {
119                         coeff[l] = tmp[l - 1];
120                     }
121                     data[index(coeff[0], coeff[1], coeff[2], dim)] = buff[k * dim[
                            qq[1]] + j];
122                 }
123             }
124         } else {
125             for (uint64_t k = 0; k < dim[qq[0]]; ++k) {
126                 for (uint64_t j = 0; j < dim[qq[1]]; ++j) {
127                     int coeff[3];
128                     uint64_t tmp[] = {j, k};
129                     for (uint64_t l = 0; l < ax; ++l) {
130                         coeff[l] = tmp[l];
131                     }
132                     coeff[ax] = dim[ax];
133                     for (uint64_t l = ax + 1; l < COUNT_OF_AXES; ++l) {
134                         coeff[l] = tmp[l - 1];
135                     }
136                     data[index(coeff[0], coeff[1], coeff[2], dim)] = u[dir];
```

```
137                     }
138                 }
139             }
140 }
```

Результаты

Замеры времени работы программы с различными конфигурациями (время указано в микросекундах).

Размер сетки \ Число процессов	1	2	4	8
1000	6 327	22 351	7 021	7 401
10 000	157 941	102 521	97 145	121 873
30 000	1 123 839	741 983	452 901	669 230
60 000	3 522 883	1 775 134	1 052 952	1 525 567
100 000	8 453 963	4 464 091	2 963 146	3 153 957

По таблице видно, что лучшая производительность достигается при использовании 4 процессов.

Выводы

Во время выполнения данной работы я познакомился с технологией MPI, её базовыми механизмами обмена сообщениями между процессами, такими как блокирующая и неблокирующая передача данных между двумя процессами, а также функциями $MPI\_Allreduce$, MPI_Send, MPI_Recv.