Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Кафедра компьютерных систем и программных технологий

Параллельные вычисления

Создание многопоточной программы на языке C++ с использованием Message Passing Interface (MPI)

Работу выполнил:
Сафонов С.В.
Группа:
3540901/91502
Преподаватель:
Стручков И. В.

Санкт-Петербург
2020

Содержание

1. Цель работы

Получение навыков создания многопоточных программ с использованием интерфейса MPI.

2. Программа работы

1. Для алгоритма из полученного задания написать последовательную программу на языке C или C++, реализующую этот алгоритм;
2. Для созданной последовательной программы необходимо написать 3-5 тестов, которые покрывают основные варианты функционирования программы;
3. Проанализировать полученный алгоритм, выделить части, которые могут быть распараллелены, разработать структуру параллельной программы. Определить количество используемых потоков, а также правила и используемые объекты синхронизации;
4. Согласовать разработанную структуру и детали реализации параллельной программы с преподавателем;
5. Написать код параллельной программы и проверить ее корректность на созданном ранее наборе тестов. При необходимости найти и исправить ошибки;
6. Провести эксперименты для оценки времени выполнения последовательной и параллельной программ. Проанализировать полученные результаты;
7. Сделать общие выводы по результатам проделанной работы:
    - различия между способами проектирования последовательной и параллельной реализаций алгоритма;
    - Возможные способы выделения параллельно выполняющихся частей, Возможные правила синхронизации потоков;
    - Сравнение времени выполнения последовательной и параллельной программ;
    - Принципиальные ограничения повышения эффективности параллельной реализации по сравнению с последовательной.

3. Индивидуальное задание

Вариант №9
Расчёт определителя матрицы.
Средство распараллеливания - MPI.

4. Ход выполнения работы

4.1. Последовательная реализация

В качестве матрицы был выбран целочисленный массив. Реализованы методы для заполнения матрицы случайными числами от 1 до 100 и её отображения

Листинг 1: Функции инициализации и отображения матрицы

```
9  void generateMatrix(int* matrix, int n) {
10     std::random_device rd;
11     std::mt19937 mt(rd());
12     std::uniform_int_distribution<int> dist(-10, 10);
13     for (int i = 0; i < n; ++i) {
14         matrix[i] = dist(mt);
15     }
16 }
18 void displayMatrix(int* matrix, int n)
19 {
20     std::cout << "Matrix:" << std::endl;
21     for (int i = 0; i < n; ++i) {
22         std::cout << matrix[i] << " ";
24         if ((i + 1) % (int)sqrt(n) == 0) {
25             std::cout << "\n";
26         }
27     }
28     std::cout << "\n";
29 }
```

Реализация представлет из себя рекурсивный вызов функции вычисления определителя матрицы и его умножение на текущий элемент матрицы в диапазоне [(0,0);(0,n)], где n - размер матрицы. Основа реализации – формула вычисления по строке:

$$\Delta = \sum_{j=1}^{n} (-1)^{1+j} * a_{1j} * \bar{M}_j^1 \tag{1}$$

Листинг 2: Реализация вычисления определителя

```
31 void getCofactor(int* matrix, int* temp, int p, int q, int n)
32 {
33     int i = 0, j = 0;
34     for (int row = 0; row < n; ++row)
35     {
36         for (int col = 0; col < n; ++col)
37         {
38             if (row != p && col != q)
39             {
40                 temp[i * (n - 1) + j++] = matrix[row * n + col];
41                 if (j == n - 1)
42                 {
43                     j = 0;
44                     i++;
45                 }
46             }
47         }
48     }
49 }
51 long long determinantOfMatrix(int* matrix, int n, int init, int end, int sig)
52 {
53     long long D = 0;
54     if (n == 1) {
55         return matrix[0];
56     }
57     int* temp = new int[n * n];
```

```
58      int sign = sig;
59      for (int f = init; f < end; ++f)
60      {
61          getCofactor(matrix, temp, 0, f, n);
62          D += sign * matrix[f] * determinantOfMatrix(temp, n - 1, 0, n - 1, -1);
63          sign = -sign;
65
65      }
66      delete[] temp;
67      return D;
68  }
```

Для проверки корректности работы алгоритма пользователю предлагается ввести размер квадратной матрицы. Результаты вычисления определителя были сравнены с результатами, полученным с помощью онлайн-калькуляторов и программы WolframAlpha.

Листинг 3: Функция main

```
71  int main() {
72      int n;
73      std::cout << "Enter the size of the matrix:\n";
74      std::cin >> n;
75      int* matrix = new int[n * n];
77
77      generateMatrix(matrix, n * n);
78      displayMatrix(matrix, n * n);
80
80      int sig = n % 2 == 0 ? 1 : -1;
82
82      auto start = high_resolution_clock::now();
83      long long determinant = determinantOfMatrix(matrix, n, 0, n, sig);
84      std::cout << "Determinant: " << determinant << std::endl;
86
86      auto end_time = duration_cast<duration<double>>(high_resolution_clock::now()
        ↪  - start).count();
87      std::cout << "Execution time: " << end_time << " seconds" << std::endl;
88      return 0;
89  }
```

Листинг 4: Результаты тестирования последовательной реализации

```
1  Enter the size of the matrix:
2  2
3  Matrix:
4  -3 4
5  -8 7
7
7  Determinant: 11
8  Execution time: 0.0001893 seconds
10
10 Enter the size of the matrix:
11 3
12 Matrix:
13 -8 5 8
14 -7 5 -5
15 -3 5 -6
17
17 Determinant: -255
18 Execution time: 0.0003204 seconds
20
20 Enter the size of the matrix:
```

```
5
Matrix:
5 3 0 -1 1
3 7 -4 3 -5
8 -9 3 1 -5
5 5 -3 -2 3
-1 -7 7 -10 8

Determinant: -8266
Execution time: 0.0003336 seconds

Enter the size of the matrix:
10
Matrix:
-3 -2 1 3 -9 -3 -7 4 4 4
-7 4 -9 -3 8 -8 -6 10 -4 -7
2 10 1 -7 9 -9 -3 4 -4 9
3 -6 -6 -7 -6 -3 2 8 1 -2
7 -4 -4 -2 -1 6 7 -8 8 10
9 0 -5 1 -3 0 3 -6 2 5
-8 -1 2 -4 7 7 -5 2 -10 -4
6 -1 -1 3 -3 -5 10 -1 -5 -2
-8 -7 -5 8 -7 6 -10 5 -10 2
4 0 1 9 7 -8 -7 -3 -8 2

Determinant: -30293151852
Execution time: 2.16812 seconds
```

4.2. Параллельная реализация с использованием MPI

Для многопоточной реализации с помощью MPI необходимо осуществлять взаимодействие процессов путём передачи сообщений. Были введены два типа процессов: master и slave. master-процесс выполняет случайную генерацию элементов заданной матрицы и её отображение, отправку сгенерированной матрицы slave-процессам и получение результата вычислениий. slave-процессы осуществляют непосредственное вычисление определителя матрицы и отправку результата master-процессу. Количество рабочих процессов и размер целевой матрицы задаются при запуске программы из командной строки. Одним из изменений в параллельной реализации явилось задание целевой матрицы в качестве массива, так как сигнатура функции *MPI_Send()* не подразумевает передачу между процессами вектора значений.

Листинг 5: Параллельная реализация с использованием MPI

```
int main(int argc, char* argv[]) {
    int numtasks,
        taskid,
        numworkers,
        rc = 0;

    int n = atoi(argv[1]);
    int* matrix = new int[n * n];
    long long det;
    long long determinant = 0;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

```

```
92      if (numtasks < 2) {
93          printf("Need to have at least 2 processes. Aborting.\n");
94          MPI_Abort(MPI_COMM_WORLD, rc);
95          exit(1);
96      }
97      numworkers = numtasks - 1;
98      int* res = new int[numworkers];
100
100     if (taskid == MASTER) {
101         printf("mpi_mm initialization of %d processes.\n\n", numtasks);
103
103         generateMatrix(matrix, n * n);
104         displayMatrix(matrix, n * n);
106
106         auto start = high_resolution_clock::now();
108
108         for (int i = 1; i < numtasks; ++i) {
109             MPI_Send(matrix, n * n, MPI_INTEGER, i, 0, MPI_COMM_WORLD);
110         }
112
112         for (int i = 1; i < numtasks; ++i) {
113             MPI_Recv(&res[i - 1], 1, MPI_INTEGER, i, 0, MPI_COMM_WORLD,
    ↪ MPI_STATUS_IGNORE);
114         }
116
116         for (int i = 0; i < numworkers; ++i) {
117             determinant += res[i];
118         }
120
120         std::cout << "Determinant: " << determinant << std::endl;
121         auto end_time = duration_cast<duration<double>>(high_resolution_clock::
    ↪ now() - start).count();
122         std::cout << "Execution time: " << end_time << " seconds" << std::endl;
123     }
125
125     else if (taskid > MASTER) {
127
127         MPI_Recv(matrix, n * n, MPI_INTEGER, 0, 0, MPI_COMM_WORLD,
    ↪ MPI_STATUS_IGNORE);
129
129         int init = (n / numworkers * (taskid - 1));
130         int end = (n / numworkers * taskid);
131         int sig = init % 2 == 0 ? 1 : -1;
133
133         if (taskid == numworkers) {
134             end = n;
135         }
137
137         det = determinantOfMatrix(matrix, n, init, end, sig);
139
139         MPI_Send(&det, 1, MPI_INTEGER, 0, 0, MPI_COMM_WORLD);
140     }
142
142     delete[] res;
143     delete[] matrix;
144     MPI_Finalize();
145 }
```

Для проверки корректности работы алгоритма было проведено несколько итераций вычисления определителя с различными количеством рабочих процессов и размером целе-

вой матрицы. Так как персональный компьютер, на котором производились вычисления, имеет 2 физических ядра, то максимальное число рабочих потоков не стоит принимать больше 6.

Листинг 6: Результаты тестирования параллельной реализации

```
C:\Users\I535031\source\repos\matrix-determinant-mpi\x64\Debug>mpiexec -n 2
    ↪ matrix-determinant-mpi.exe 3
mpi_mm initialization of 2 processes.

Matrix:
-6 9 10
-2 -2 8
-4 2 -5

Determinant: -462
Execution time: 0.0017157 seconds

C:\Users\I535031\source\repos\matrix-determinant-mpi\x64\Debug>mpiexec -n 2
    ↪ matrix-determinant-mpi.exe 5
mpi_mm initialization of 2 processes.

Matrix:
1 -2 -7 10 -6
-7 -2 8 -9 8
10 4 -8 9 -7
8 -7 -7 3 7
-5 -6 6 -7 6

Determinant: -4572
Execution time: 0.0024056 seconds

C:\Users\I535031\source\repos\matrix-determinant-mpi\x64\Debug>mpiexec -n 2
    ↪ matrix-determinant-mpi.exe 10
mpi_mm initialization of 2 processes.

Matrix:
-2 -6 -1 -2 7 9 -8 -3 -4 -9
-1 -7 9 4 2 -2 -1 -8 -4 1
-1 -7 3 -4 -9 1 -3 -7 -5 -2
4 2 8 -5 7 8 -1 5 -2 -4
-9 5 2 -3 3 7 9 0 4 2
1 1 2 -10 0 3 -3 4 -1 -2
0 -3 3 8 9 3 10 -8 -7 -3
4 3 -7 2 -4 -7 -5 -9 -6 -3
0 8 1 -3 1 -4 -10 8 -1 -6
0 2 -4 10 7 -4 -3 -9 -9 -3

Determinant: -589776453
Execution time: 1.11964 seconds

C:\Users\I535031\source\repos\matrix-determinant-mpi\x64\Debug>mpiexec -n 4
    ↪ matrix-determinant-mpi.exe 3
mpi_mm initialization of 4 processes.

Matrix:
-3 -9 7
-7 -8 -6
7 7 7

Determinant: 28
```

```
Execution time: 0.0024047 seconds

C:\Users\I535031\source\repos\matrix-determinant-mpi\x64\Debug>mpiexec -n 4
    ↪ matrix-determinant-mpi.exe 5
mpi_mm initialization of 4 processes.

Matrix:
4 -7 1 -5 -8
0 -9 -7 5 -1
-7 -5 -3 6 8
4 -8 -8 1 -4
8 -1 10 2 1

Determinant: 28909
Execution time: 0.0086109 seconds

C:\Users\I535031\source\repos\matrix-determinant-mpi\x64\Debug>mpiexec -n 4
    ↪ matrix-determinant-mpi.exe 10
mpi_mm initialization of 4 processes.

Matrix:
8 -8 -3 5 8 -9 4 -6 4 8
-7 6 -6 5 -5 2 7 6 -7 1
-4 -10 1 -8 -2 -3 6 -3 -6 -3
-7 4 1 -4 6 6 -2 -7 -8 -9
10 -5 -4 9 -1 4 8 -2 5 1
-4 6 -2 -7 6 6 5 -2 -5 -5
1 -10 3 -8 7 0 7 -7 2 3
3 1 -10 3 -8 -10 1 -3 -5 -2
-2 3 7 8 0 7 -2 -6 -6 -3
-9 -10 -2 -2 -1 -10 9 -3 -8 -9

Determinant: -796698804
Execution time: 0.832552 seconds

C:\Users\I535031\source\repos\matrix-determinant-mpi\x64\Debug>mpiexec -n 6
    ↪ matrix-determinant-mpi.exe 3
mpi_mm initialization of 6 processes.

Matrix:
10 3 4
-4 6 5
-4 10 5

Determinant: -264
Execution time: 0.002634 seconds

C:\Users\I535031\source\repos\matrix-determinant-mpi\x64\Debug>mpiexec -n 6
    ↪ matrix-determinant-mpi.exe 5
mpi_mm initialization of 6 processes.

Matrix:
3 6 -4 8 3
3 0 8 -1 -3
4 -7 -6 -5 6
-9 -4 1 6 3
-7 -5 -10 10 -9

Determinant: -143802
Execution time: 0.006362 seconds
```

```
C:\Users\I535031\source\repos\matrix-determinant-mpi\x64\Debug>mpiexec -n 6 matrix-determinant-mpi.exe 10
mpi_mm initialization of 6 processes.

Matrix:
-4 -7 4 -9 8 -7 -2 -2 -8 10
8 1 7 4 -3 9 -7 -2 -4 0
4 9 6 0 -6 -8 -2 -9 -3 9
5 -1 2 -3 -3 4 6 -2 8 -8
-1 4 -10 -8 1 -3 10 -8 -7 7
-7 9 7 -6 1 -8 -6 -9 -5 0
-9 9 3 -8 -9 6 0 -9 -8 -2
-3 -10 -7 4 3 -5 -2 -3 -10 0
4 4 -10 -1 6 -9 -1 4 -4 -9
-1 -2 4 -1 5 -6 3 9 6 0

Determinant: -1744776075
Execution time: 0.517941 seconds

C:\Users\I535031\source\repos\matrix-determinant-mpi\x64\Debug>mpiexec -n 8 matrix-determinant-mpi.exe 3
mpi_mm initialization of 8 processes.

Matrix:
-5 3 2
5 -10 -6
7 -6 2

Determinant: 204
Execution time: 0.00254 seconds

C:\Users\I535031\source\repos\matrix-determinant-mpi\x64\Debug>mpiexec -n 8 matrix-determinant-mpi.exe 5
mpi_mm initialization of 8 processes.

Matrix:
5 5 -2 9 -5
5 -7 1 2 -6
1 -1 -10 6 -1
8 8 -7 3 -8
8 4 -7 0 10

Determinant: -148998
Execution time: 0.004858 seconds

C:\Users\I535031\source\repos\matrix-determinant-mpi\x64\Debug>mpiexec -n 8 matrix-determinant-mpi.exe 10
mpi_mm initialization of 8 processes.

Matrix:
6 -7 7 -2 -7 -2 0 -9 -4 -3
2 3 -6 2 -4 -3 -7 4 -6 1
-8 -2 -1 -1 2 -1 -5 0 -6 2
-7 4 -5 2 -10 3 4 -6 9 -10
-8 -2 8 9 -8 2 -1 2 3 -9
10 4 2 -10 -5 -4 -5 6 -3 -8
-10 2 0 -3 -5 8 1 4 -2 -4
-5 -8 -1 0 6 6 -9 7 -3 -10
8 -2 -9 -6 5 -8 6 1 6 8
```

```
164 -10  4  8 -1  1 -1 -6 -10  0 -1
166
166 Determinant: -6583018458
167 Execution time: 0.694848 seconds
```

4.3. Тестирование реализаций

Для всех реализаций осуществляется замер времени выполнения расчетов с помощью библиотеки std::chrono. Расчеты производятся по 50 раз. Затем по полученным данным рассчитываются математическое ожидание и дисперсия. Для многопоточной реализации производятся подсчеты в зависимости от количества потоков. В качестве тестового примера была взята матрица размера 10x10 со случайными элементами.

Величина математического ожидания рассчитывалась по формуле:

$$m = \frac{\sum_{i=1}^{n} t_i}{N} \tag{2}$$

где ti – время расчетов каждой итерации запуска, N – количество запусков. Величина дисперсии была рассчитана по формуле:

$$d = \frac{\sum_{i=1}^{n} (m - t_i)^2}{N - 1} \tag{3}$$

В качестве доверительного интервала был выбран интервал 99%. Для него коэффициент доверия для выборки равен 2.58. После этого необходимо рассчитать предел погрешности по формуле:

$$a * \frac{d^2}{\sqrt{n}} \tag{4}$$

где a – коэффициент доверия.

Таблица 4.1

Замеры времени вычисления определителя

Итерация	**default**	**MPI-2 proc**	**MPI-3 proc**	**MPI-5 proc**
1	2.07571	0.692681	0.515431	0.717301
2	1.34233	0.815205	0.527908	0.702959
3	1.14096	0.703165	0.531111	0.829458
4	1.1321	0.694572	0.534442	0.706654
5	1.24337	0.67976	0.523287	0.757939
6	1.12952	0.678707	0.529282	0.813273
7	1.12322	0.711364	0.521601	0.71341
8	1.15884	0.685567	0.521476	0.70486
9	1.15265	0.743634	0.507517	0.682383
10	1.17506	0.759868	0.498497	0.711701
11	1.12618	0.683686	0.520606	0.709359
12	1.13658	0.700199	0.499593	0.705552
13	1.25768	0.846655	0.508384	0.7045
14	1.29562	0.727013	0.510335	0.705649
15	1.17001	0.730039	0.517915	0.708236
16	1.20713	0.674883	0.545942	0.707888
17	1.51057	0.699807	0.534812	0.715098
18	1.42513	0.878924	0.528006	0.76157
19	1.1158	0.695928	0.518323	0.699263
20	1.1533	0.686539	0.550536	0.692137
21	1.14741	0.668399	0.509455	0.703839
22	1.15473	0.67323	0.501669	0.708944
23	1.16638	0.664091	0.504841	0.711881
24	1.11289	0.710328	0.621235	0.699103

Таблица 4.2

Замеры времени вычисления определителя (продолжение)

Итерация	**default**	**MPI-2 proc**	**MPI-3 proc**	**MPI-5 proc**
25	1.39787	0.687901	0.545871	0.732238
26	1.14376	0.716971	0.523495	0.711108
27	1.10926	0.694672	0.549682	0.744538
28	1.4855	0.716824	0.5172	0.725953
29	1.4844	0.681418	0.514268	0.690774
30	1.28231	0.680062	0.503939	0.862567
31	1.36022	0.706987	0.533539	0.723658
32	1.15144	0.704152	0.516764	0.720698
33	1.16186	0.66738	0.543154	0.713393
34	1.16795	0.672816	0.516293	0.694755
35	1.14853	0.822229	0.523525	0.702974
36	1.1264	0.676054	0.581894	0.706247
37	1.14326	0.710584	0.553407	0.691094
38	1.16188	0.657085	0.56372	0.850497
39	1.12214	0.683298	0.545045	0.708341
40	1.12067	0.670736	0.514469	0.734778
41	1.12464	0.725063	0.652333	0.704383
42	1.17666	0.70687	0.521986	0.698878
43	1.13841	0.674592	0.517781	0.692989
44	1.13113	0.679471	0.523758	0.703962
45	1.13541	0.705604	1.11033	0.705801
46	1.1491	0.817993	0.561258	0.695451
47	1.18261	0.744592	0.5211	0.71812
48	1.14144	0.700393	0.515253	0.723526
49	1.18891	0.709879	0.529824	0.794258
50	1.25449	0.685356	0.52968	0.699559

Результаты расчетов представлены ниже. В начале идет название метода (последовательная реализация – default, реализация с помощью MPI – MPI). Затем количество рабочих процессов (для многопоточной реализации). Параметр m соответствует величине математического ожидания, параметр d – дисперсии. Доверительный интервал для каждой из реализаций записан следующей строкой.

Листинг 7: Результаты тестирования реализаций

```
1  default : m = 1.216866 сек , d = 0.026424
2  99% interval : 1.216866 -+ 0.009642 сек
4
4  MPI 2 : m = 0,71 сек , d = 0.0023
5  99% interval : 0,71 -+ 0.00084 сек
7
7  MPI 3 : m = 0.542235 сек , d = 0.007509
8  99% interval : 0.542235 -+ 0.00274 сек
10
10 MPI 5 : m = 0.708488 сек , d = 0.001608
11 99% interval : 0.708488 -+ 0.000587 сек
```

По этим данным были построен график зависимости времени выполнения от количества рабочих процессов для обеих реализаций.

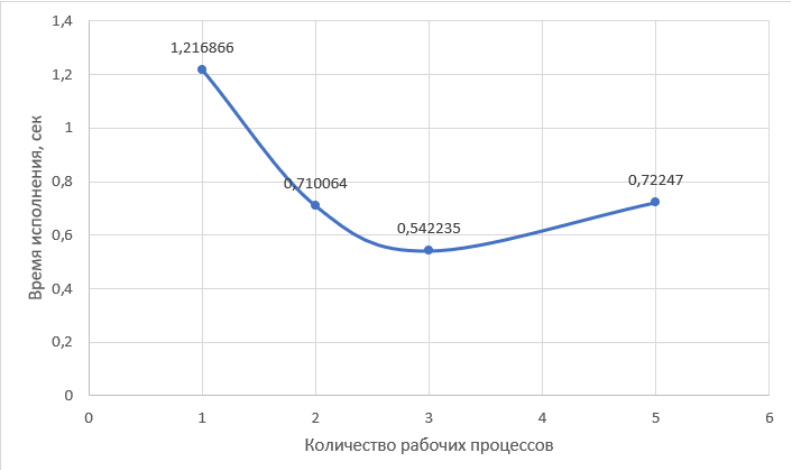

Рисунок 4.1. Зависимость времени исполнения от количества рабочих процессов

По полученным данным видно, что реализация на двух рабочих процессах MPI выигрывает у последовательной реализации примерно в 2 раза. При дальнейшем увеличении количества процессов (3 процесса) время выполнения уменьшается примерно в 1.5 раза. 3 рабочих процесса является крайней точкой, после которой происходит увеличение времени вычисления определителя. Связано это с тем, что персональный компьютер имеет 2 физических ядра и при большем количестве процессов они начинают вмешиваться в работу друг в друга, за счёт чего и увеличивается время исполнения.

5. Выводы

В рамках данной лабораторной работы были изучены основы многопоточных реализаций программ на C++ с использованием механизма MPI. Для написанных программ было проведено тестирование при разном количестве потоков, а также были оценены вероятностные характеристики времени работы. По результатам работы было установлено, что многопоточная реализация программы на MPI выиграет у последовательной реализации пропорционально количеству рабочих процессов. С другой стороны, при превышении количества процессов соответствующим строению процессора персонального компьютера происходит обратная ситуация. Рабочие процессы начинают вмешиваться в работу друг друга, за счёт чего увеличивается время исполнения программы.