Правительство Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования «Национальный исследовательский университет «Высшая школа экономики»

Факультет компьютерных наук

Департамент программной инженерии

Пояснительная записка к домашнему 4 заданию
По дисциплине
«Архитектура вычислительных систем»
2 вариант

Работу выполнил:

Студент группы БПИ-199, Асатиани Тимур Ренадиевич

Преподаватель:

Легалов Александр Иванович

Москва 2020

ЗАДАНИЕ

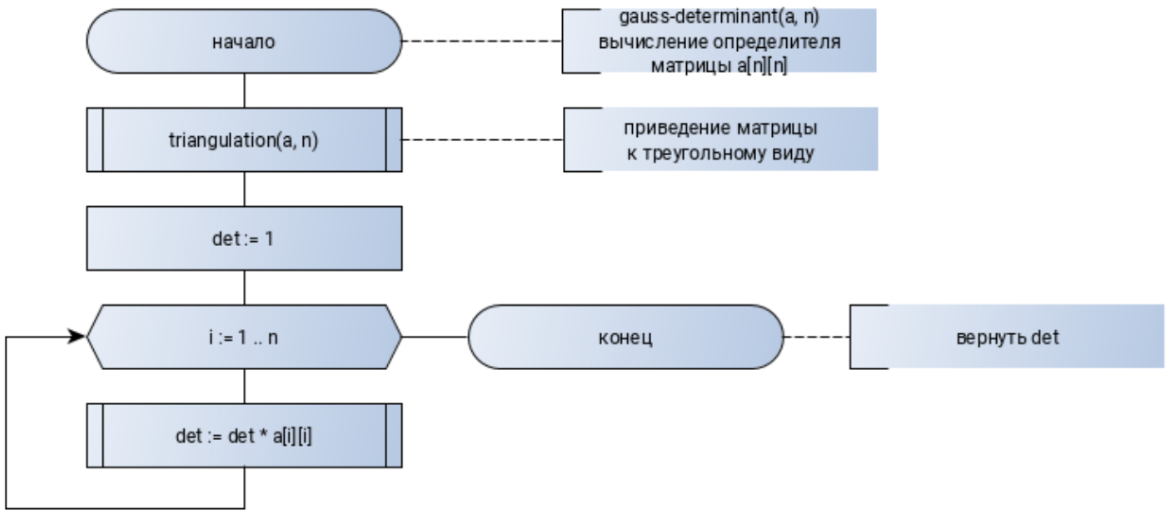
Разработать алгоритм вычисления определителя матрицы с учетом особенностей OpenMP.

Описание задания: “Найти определитель матрицы A. Входные данные: целое положительное число *n*, произвольная матрица A размерности *n* x *n*. Количество потоков является входным параметром, при этом размерность матриц может быть не кратна количеству потоков” (Вариант 2).

РЕШЕНИЕ

Метод Гаусса

Для нахождения определителя применим метод Гаусса. Алгоритм метода схематически изображен ниже:



Важной частью алгоритма является приведение матрицы к треугольному виду. Этот процесс занимает больше всего времени и ресурсов компьютера, поэтому займемся его оптимизацией с помощью OpenMP.

Триангуляция матрицы

Алгоритм приведения матрицы к треугольному виду содержит 3 главных части:

1) Проход по всей матрице от 0 … n ;

2) Поиск максимального элемента в i-ом столбце;

3) Элементарные преобразования со всеми j строками (j = i + 1).

Применим “Итеративный параллелизм” для оптимизации 2 и 3 пунктов.

Поиск максимального элемента

Для того чтобы найти максимум в столбце используем реализованный средствами OpenMP:

Каждый поток будет просматривать свои элементы и сравнивать их с локальным максимум среди них {local_max}, а после этого в критической секции все потоки соединится и по очереди сравнят свои локальнее максимумы с глобальным.

Т.к Матрица {matrix} передается как константа, используется параметр shared(matrix) чтобы не занимать лишнюю память локальными переменными в потоках.

```
// Функция ищет максимальный элемент в {col} столбце матрицы {matrix}.
int col_max(const vector<vector<double>>& matrix, int col) {
    int n = matrix.size();
    double max = abs(matrix[col][col]);
    int maxPos = col;

    #pragma omp parallel shared(matrix)
    {
        // локальный максимум в потоке.
        double loc_max = max;
        int loc_max_pos = maxPos;
        #pragma omp for
        for (int i = col + 1; i < n; ++i) {
            double element = abs(matrix[i][col]);
            if (element > loc_max) {
                loc_max = element;
                loc_max_pos = i;
            }
        }
        #pragma omp critical    // ищем максимум среди всех потоков.
        {
            if (max < loc_max) {
                max = loc_max;
                maxPos = loc_max_pos;
            }
        }
    }
    return maxPos;
}
```

Элементарные преобразования

Между потоками можно разделить каждую из {n - 1} строк, для которых выполняются преобразования:

```
// Функция, вычитающая i-ю строку из остальных строк, находящихся ниже неё так, чтобы i-й элемент каждой строки был равен 0.
void elementaryConversion(vector<vector<double>>& matrix, int mainRow) {
    int n = matrix.size();

    // параллельно вычитаем {mainRow} строку из всех строк, ниже неё.
    #pragma omp parallel for
    for (int j = mainRow + 1; j < n; ++j) {
        // множитель для {mainRow} строки.
        double mul = -matrix[j][mainRow] / matrix[mainRow][mainRow];
        for (int k = mainRow; k < n; ++k)
            matrix[j][k] += matrix[mainRow][k] * mul;
    }

}
```

Здесь каждый поток работает со своей строкой поэлементно прибавляя к ней {mainRow} строку.

ФОРМАТ ВХОДНЫХ ДАННЫХ

Названия файлов вводятся в аргументах командной строки (1 и 2 аргументы).

Например: test_1000.txt out_1000.txt

В первой строке файла вводится N - размер матрицы(матрица квадратная). В последующих N строках содержатся элементы матрицы через пробел.

Пример входных данных см. в папке input.

ФОРМАТ ВЫХОДНЫХ ДАННЫХ

В первой строке файла будет выведен размер исходной матрицы.

Во второй - определитель.

В третьей - время работы последовательной программы.

В четвертой – время работы многопоточной программы.

Пример входных данных см. в папке output.

ТЕКСТ ПРОГРАММЫ

```
1.  #include <iostream>
2.  #include <vector>
3.  #include <cmath>
4.  #include <fstream>
5.  #include <ctime>
6.  #include "omp.h"
7.
8.  using namespace std;
9.
10. // Функция ищет максимальный элемент в {col} столбце матрицы {matrix}.
11. int col_max(const vector<vector<double>>& matrix, int col) {
12.     int n = matrix.size();
13.     double max = abs(matrix[col][col]);
14.     int maxPos = col;
15.
16. #pragma omp parallel shared(matrix)
17.     {
18.         // локальный максимум в потоке.
19.         double loc_max = max;
20.         int loc_max_pos = maxPos;
21. #pragma omp for
22.         for (int i = col + 1; i < n; ++i) {
23.             double element = abs(matrix[i][col]);
24.             if (element > loc_max) {
25.                 loc_max = element;
26.                 loc_max_pos = i;
27.             }
28.         }
29. #pragma omp critical    // ищем максимум среди всех потоков.
30.         {
31.             if (max < loc_max) {
32.                 max = loc_max;
33.                 maxPos = loc_max_pos;
34.             }
35.         }
36.     }
37.     return maxPos;
38. }
39.
40. int col_maxSimple(const vector<vector<double>>& matrix, int col) {
41.     int n = matrix.size();
42.     double max = abs(matrix[col][col]);
43.     int maxPos = col;
44.
45.     //#pragma omp parallel shared(matrix)
46.     {
47.         // локальный максимум в потоке.
48.         double loc_max = max;
49.         int loc_max_pos = maxPos;
50.         //#pragma omp for
51.         for (int i = col + 1; i < n; ++i) {
52.             double element = abs(matrix[i][col]);
53.             if (element > loc_max) {
54.                 loc_max = element;
55.                 loc_max_pos = i;
56.             }
57.         }
58.         //#pragma omp critical  // ищем максимум среди всех потоков.
59.         {
60.             if (max < loc_max) {
61.                 max = loc_max;
62.                 maxPos = loc_max_pos;
63.             }
```

```
64.         }
65.     }
66.     return maxPos;
67. }
68.
69. // Функция меняет местами {origin} и {sub} строки матрицы.
70. void swap(vector<vector<double>>& matrix, int origin, int sub) {
71.     vector<double> temp = matrix[origin];
72.     matrix[origin] = matrix[sub];
73.     matrix[sub] = temp;
74. }
75.
76. // Функция, вычитающая i-ю строку из остальных строк, находящихся ниже неё так, чтобы
    i-й элемент каждой строки был равен 0.
77. void elementaryConversion(vector<vector<double>>& matrix, int mainRow) {
78.     int n = matrix.size();
79.
80.     // параллельно вычитаем {mainRow} строку из всех строк, ниже неё.
81. #pragma omp parallel for
82.     for (int j = mainRow + 1; j < n; ++j) {
83.         // множитель для {mainRow} строки.
84.         double mul = -matrix[j][mainRow] / matrix[mainRow][mainRow];
85.         for (int k = mainRow; k < n; ++k)
86.             matrix[j][k] += matrix[mainRow][k] * mul;
87.     }
88.
89. }
90.
91. void elementaryConversionSimple(vector<vector<double>>& matrix, int mainRow) {
92.     int n = matrix.size();
93.
94.     // параллельно вычитаем {mainRow} строку из всех строк, ниже неё.
95.     //#pragma omp parallel for
96.     for (int j = mainRow + 1; j < n; ++j) {
97.         // множитель для {mainRow} строки.
98.         double mul = -matrix[j][mainRow] / matrix[mainRow][mainRow];
99.         for (int k = mainRow; k < n; ++k)
100.             matrix[j][k] += matrix[mainRow][k] * mul;
101.     }
102.
103. }
104.
105. // Функция, приводящая матрицу, переданную в неё к треугольному виду.
106. int triangulation(vector<vector<double>>& matrix) {
107.     int n = matrix.size();
108.     int swapCount = 0;
109.
110.     // проходимся по всей матрице.
111.     for (int i = 0; i < n; ++i) {
112.
113.         // находим строку, в которой i-й элемент самый большой в своем столбце.
114.         // ПАРАЛЛЕЛЬНО.
115.         int sub = col_max(matrix, i);
116.
117.
118.         // меняем найденную строку с i-й местами.
119.         if (sub != i) {
120.             swap(matrix, i, sub);
121.             swapCount++;
122.         }
123.
124.
125.
126.         // вычитаем i-ю строку из остальных строк, находящихся ниже неё так чтобы
    i-й элемент каждой строки был равен 0.
127.         // ПАРАЛЛЕЛЬНО.
```

```
128.          elementaryConversion(matrix, i);
129.      }
130.      return swapCount;
131.  }
132.
133.  int triangulationSimple(vector<vector<double>>& matrix) {
134.      int n = matrix.size();
135.      int swapCount = 0;
136.
137.      // проходимся по всей матрице.
138.      for (int i = 0; i < n; ++i) {
139.
140.          // находим строку, в которой i-й элемент самый большой в своем столбце.
141.          // ПАРАЛЛЕЛЬНО.
142.          int sub = col_maxSimple(matrix, i);
143.
144.
145.          // меняем найденную строку с i-й местами.
146.          if (sub != i) {
147.              swap(matrix, i, sub);
148.              swapCount++;
149.          }
150.
151.
152.          // вычитаем i-ю строку из остальных строк, находящихся ниже неё так чтобы
    i-й элемент каждой строки был равен 0.
153.          // ПАРАЛЛЕЛЬНО.
154.          elementaryConversionSimple(matrix, i);
155.      }
156.      return swapCount;
157.  }
158.
159.  // Функция, вычисляющая определитель переданной в нее матрицы.
160.  double calculateGaussDeterminant(vector<vector<double>> matrix) {
161.      int n = matrix.size();
162.
163.      // приводим матрицу к треугольному виду.
164.      int swapCount = triangulation(matrix);
165.
166.      // подсчитываем определитель.
167.      double det = 1;
168.      if (swapCount % 2 == 1)
169.          det = -1;
170.      #pragma omp parallel reduction (*: det)
171.      {
172.      # pragma omp for
173.          for (int i = 0; i < n; ++i) {
174.              det *= matrix[i][i];
175.          }
176.
177.
178.      }
179.      return round(det);
180.  }
181.
182.  double calculateGaussDeterminantSimple(vector<vector<double>> matrix) {
183.      int n = matrix.size();
184.
185.      // приводим матрицу к треугольному виду.
186.      int swapCount = triangulationSimple(matrix);
187.
188.      // подсчитываем определитель.
189.      double det = 1;
190.      if (swapCount % 2 == 1)
191.          det = -1;
192.      //#pragma omp parallel reduction (*: det)
```

```
193.     {
194.         //# pragma omp for
195.         for (int i = 0; i < n; ++i) {
196.             det *= matrix[i][i];
197.         }
198.
199.
200.     }
201.     return round(det);
202. }
203.
204.
205. // Функция, читающая матрицу {matrix} из файла.
206. void readMatrixFromFile(vector<vector<double>>& matrix, ifstream& input) {
207.     // первая строка в файле - размерность матрицы.
208.     int n = matrix[0].size();
209.
210.     for (int i = 0; i < n; ++i) {
211.         for (int j = 0; j < n; ++j) {
212.             input >> matrix[i][j];
213.         }
214.     }
215. }
216.
217. double checkTime(bool isMP, vector<vector<double>> matrix, double& determinant) {
218.     double start = clock();
219.
220.     if (isMP) {
221.         // запускаем многопоточную версию функции вычисления определителя методом
    Гаусса.
222.         determinant = calculateGaussDeterminant(matrix);
223.     }
224.     else {
225.         determinant = calculateGaussDeterminantSimple(matrix);
226.     }
227.     double end = clock();
228.
229.     return (double)(end - start) / CLOCKS_PER_SEC;
230. }
231.
232. int main(int argc, char* argv[]) {
233.
234.     ifstream input;
235.     ofstream output;
236.     input.open(argv[1]);
237.     output.open(argv[2]);
238.
239.     int n;
240.     input >> n;
241.
242.     // инициализируем матрицу с которой будем работать.
243.     vector<vector<double>> matrix(n, vector<double>(n));
244.
245.     // считываем матрицу из файла с именем {argv[1]}.
246.     readMatrixFromFile(matrix, input);
247.
248.     double determinant = 0;
249.
250.     // засекаем время работы многопточного алгоритма.
251.     double time1 = checkTime(true, matrix, determinant);
252.     // засекаем время работы последовательного агоритма.
253.     double time2 = checkTime(false, matrix, determinant);
254.
255.     output << "При размере матрицы == " << n << endl;
256.     output << "Определитель == " << determinant << endl;
257.     output << "Время работы обычной программы : " << time2 << " сек." << endl;
```

```
258.      output << "Время работы многопоточной программы : " << time1 << " сек.";
```

ТЕСТЫ И ВЫВОД

*тесты запускались в режиме Release.

Тест 1

```
При размере матрицы == 3
Определитель == -1818
Время работы обычной программы : 0 сек.
Время работы многопоточной программы : 0.006 сек.
```

Тест 2

```
При размере матрицы == 100
Определитель == 2.20143e+93
Время работы обычной программы : 0.001 сек.
Время работы многопоточной программы : 0.008 сек.
```

Тест 3

```
При размере матрицы == 500
Определитель == inf
Время работы обычной программы : 0.104 сек.
Время работы многопоточной программы : 0.037 сек.
```

Тест 4

```
При размере матрицы == 1000
Определитель == inf
Время работы обычной программы : 0.828 сек.
Время работы многопоточной программы : 0.244 сек.
```

Тест 5

```
При размере матрицы == 2000
Определитель == inf
Время работы обычной программы : 6.058 сек.
Время работы многопоточной программы : 1.711 сек.
```

Вывод: как и ожидалось: распараллеливание процессов приносит ощутимую экономию ресурсов и более быстрою работу программы, как следствие. Но параллельность имеет смысл при обработке больших данных. На данных тестах видно, что применение многопоточности становится выгодным при размере матрицы от 500 и больше.

Примечание: на больших данных происходит переполнение double и определитель посчитать не получается. Можно считать по модулю, но я решил оставить inf для наглядности.

Список используемых источников

1. https://docs.microsoft.com/ru-ru/cpp/parallel/openmp/reference/openmp-library-reference?view=msvc-160- документация Microsoft по OpenMP;
2. https://software.intel.com/content/www/ru/ru/develop/articles/more-work-sharing-with-openmp.html - статья на Intel;
3. “Параллельное программирование с использованием технологии OpenMP”- Антонов А.С. 2009 (Главы).