Student Name: Галактионов Кирилл
Student ID: st067889

Современные технологии программирования
в научных исследованиях
Лабораторная работа 2

---

1 Постановка задачи

Задача заключалась в оптимизации алгоритма нахождения максимального по модулю собственного числа заданной, вещественной, симметричной матрицы **A**, при помощи метода прямых итераций. При этом, требовалось исследовать зависимость масштабируемости параллельной версии программы от ее вычислительной трудоемкости и проверить закон Амдала, изучив зависимости ускорения от числа потоков.

2 Описание тестового стенда

Вычисления проводились на выданном вычислительном узле, на котором доступно максимум 6 вычислительных потоков. Вычислительный узел:

- Операционная система: Ubuntu 20.04.4 LTS
- Процессор: Intel(R) Xeon(R) E-2136 CPU @ 3.30GHz, 6 ядер
- Доступная оперативная память: 62-63 ГБ
- Компилятор: gcc 10.2.0 (mpic++)

Время работы алгоритмов вычислялось при помощи команды MPI_Wtime(). Проводилось по три серии экспериментов, затем полученные значения времен усреднялись. Проверка правильности полученных значений осуществлялась сравнением с вычисленным последовательным методом.

3 Описание алгоритма решения задачи

Алгоритм прямых итераций, поиска максимального по модую собственного числа матрицы **A** заключается в итерационном применении следующей формулы: $v_{k+1} = \frac{Av_k}{||Av_k||}$, при которой последовательность векторов v_k сходится к собственному вектору, отвечающему максимальному по модулю собственному числу, а собственным числом является норма $||Av_k||$.

Так как итерации необходимо выполнять последовательно, то их нельзя запускать параллельно на разных потоках. Поэтому было принято решение реализовывать параллельное выполнение наиболее ресурсоёмкой операции - умножения матрицы на вектор. Реализовывалось параллельное матрично-векторное умножение при помощи технологии передачи информации между потоками MPI и языка C++. Были исследован способ реализации параллельного умножения матрицы на вектор, который представляет собой распараллеливание по строкам. То есть каждому потоку на выполнение передается часть строк для вычисления. Код функции представлен ниже.

```
void RowMatrixVectorMultiply(int dim, double* matrix_data, double* vector_data, double*
    result) {
    // Initialize variables and get size of communicator
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    // Initialize local matrix and vector
    double* localresult = new double[dim / size]{};
    double* matrix = new double[dim * dim / size]{};    //local matrix

    MPI_Barrier(MPI_COMM_WORLD);
    //Scatter the Matrix from master to slaves
    MPI_Scatter(matrix_data, (dim * dim) / size, MPI_DOUBLE, matrix, (dim * dim) / size,
        MPI_DOUBLE, 0, MPI_COMM_WORLD);
    // Send same vector from master to slaves
    MPI_Bcast(vector_data, dim, MPI_DOUBLE, 0, MPI_COMM_WORLD);// Broadcast the Vector

    //Calculate the results
    for (int i = 0; i < (dim / size); i++)
        for (int j = 0; j < dim; j++)
            localresult[i] += vector_data[j] * matrix[i * dim + j];
    // Gather separetely calculatet parts together
    MPI_Gather(localresult, (dim) / size, MPI_DOUBLE, result, (dim) / size, MPI_DOUBLE,
        0, MPI_COMM_WORLD); // Gather the results
}
```

4 Программный код

Полный код программы, используемый для решения задачи в разных условиях представлен в Приложении 1 в разделе 7.

5 Результаты измерений

На графиках приведена зависимость ускорения от параметров задачи. Ускорение считалось как $K(N) = \frac{t_1}{t_N}$, где t_1 - время затрачиваемое на работу алгоритма в последовательном режиме, t_N - время работы алгоритма, при использовании N потоков.

5.1 Изучение масштабируемости

Для изучения масштабируемости строились графики зависимости ускорения от размерности задачи (размерности матрицы (n)) для разных значений количества потоков, и отвечающих вычислению на предоставленном вычислительном узле. Графики приведены на Рисунках 1 - 4.

5.2 Проверка закона Амдала

Для проверки закона Амдала:

$$K(N) = \frac{N}{S * N + (1 - S)}$$

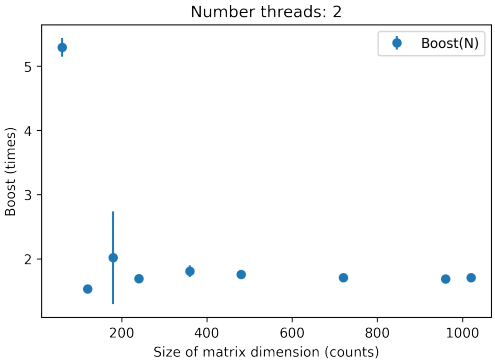

Рис. 1: 2 параллельных потока

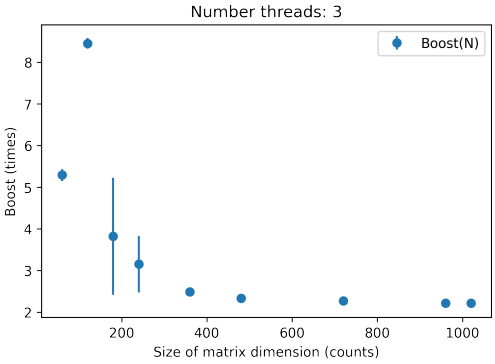

Рис. 2: 3 параллельных потоков

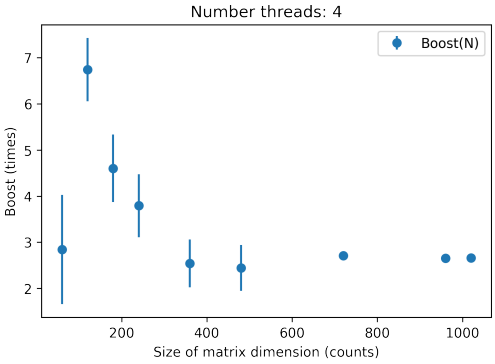

Рис. 3: 4 параллельных потоков

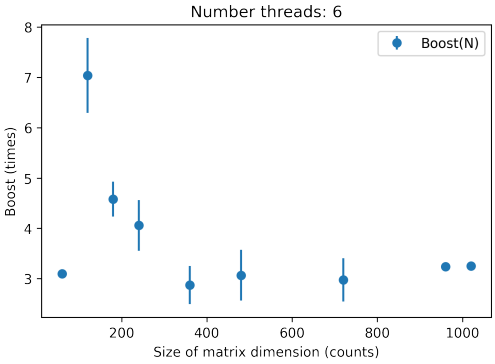

Рис. 4: 6 параллельных потоков

где, K - достигаемое ускорение, N - число потоков, S - доля последовательной части алгорима, который описывает как должно расти максимально возможное ускорение с ростом числа потоков, исследовалась зависимость ускорения от числа потоков, при фиксированной размерности задачи. Полученные зависимости представлены на рисунках 5 - 8.

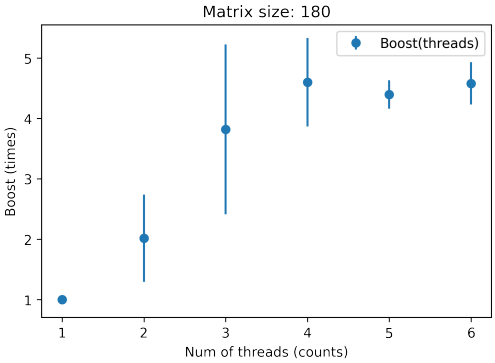

Рис. 5: Матрица размером в 180

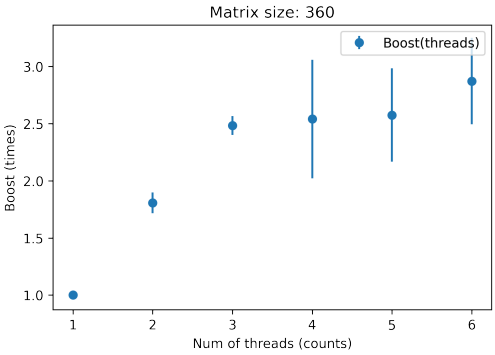

Рис. 6: Матрица размером в 360

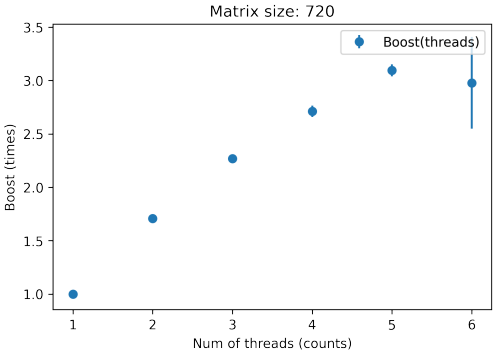

Рис. 7: Матрица размером в 720

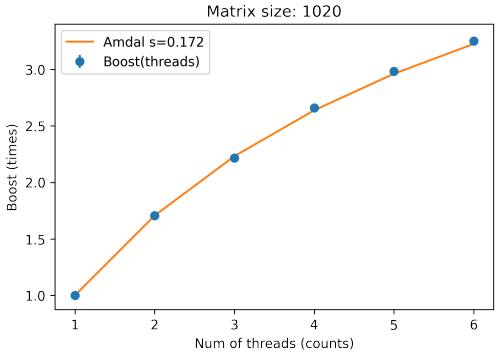

Рис. 8: Матрица размером в 1020

6 Обсуждение результатов и вывод

Были успешно проведены несколько серий экспериментов по оценке масштабируемости алгорима прямых итераций. По рисункам 1 - 4 можно заметить, что в области матриц малой размерности (до линейного размера в 400) ускорение ведет себя хаотично, в силу того, исследуются матрицы совсем малого размера, где большой вклад вносят случайные факторы, но при увеличении размера матриц, ускорение становится постоянным. Для всех значений количества потоков зависимости имеют схожий харктер.

На рисунках 5 - 8 зависимости ускорения от числа потоков, наблюдается рост ускорения при увеличении числа потоков, при этом, чем больше размер матрицы, тем меньше разброс результатов. На Рисунке 8 также приведен график закона Амдала с коэффициентом последовательной части алгоритма $S = 0.172$, который хорошо согласуется с полученной экспериментальной зависимостью.

Подводя итог, были изучены характеристики параллельного алгоритма вычисления максимального по модулю собственного числа матрицы, методом прямых итераций. Успешно проверен закон Амдала, показана его согласованность с экспериментальными даннымы и исследована масштабируемость параллельной программы.

7 Приложение 1. Код программы

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <cmath>
#include <fstream>
#include <iomanip>
#include "mpi.h"


using namespace std;

void matrixVectorMult(int dim, double* mat, double* vec, double* result);
void RowMatrixVectorMultiply(int dim, double* mat, double* vec, double* result);
void read_mat_from_file(const char* s, int n_row, int n_col, double* in_matrix);
void vectorNormalize(int dim, double* vec, const double vecNorm);
```

```
16 double vectorNorm(int dim, double* vec);
17
18 int main(int argc, char* argv[]) {
19     const int n = 1020;                                                    ///
       !!! SET MATRIX SIZE !!!
20     int rank, size;                    //MPI RANK,SIZE
21     MPI_Init(&argc, &argv);                //Initializations
22     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
23     MPI_Comm_size(MPI_COMM_WORLD, &size);
24
25     if (n % size) {  //Valid Communicator Size?
26         MPI_Finalize();
27         return(0);
28     }
29
30     double tolerance = 1e-8;      // Set acceptable tolerance of iterations
31     srand(static_cast <unsigned> (time(0)));
32     double A[n * n];  //matrix
33
34     // Read matrix from prepared file
35     if (rank == 0) {
36         //cout << "Reading matrix" << endl;
37         read_mat_from_file("matr.txt", n, n, (double*)A);
38         // Print matrix
39         /*
40         for (int i = 0; i < n; i++) {
41             for (int j = 0; j < n; j++) {
42                 cout << A[i][j] << " ";
43             }
44             cout << endl;
45         }
46         */
47         //cout << "Matrix read" << endl;
48     }
49
50
51     ////////////////////////////////////////////////////// iterations
        ////////////////////////////////////////////
52     // Preparation
53     double eigVector[n] = { 0.0 };      // Main vector for iterations
54     for (int i = 0; i < n; i++) {
55         eigVector[i] = 1.0;
56     }
57     double residueVector[n] = { 0.0 };
58     double nextVector[n] = { 0.0 };
59
60     double residueValue, eigValue, start_time, total_time, referenceValue;
61
62
63     // Set initial values for eigvalue and eigvector
64     if (rank == 0) {
65         start_time = MPI_Wtime();
66         residueValue = 100.0;
67         eigValue = vectorNorm(n, eigVector);
```

```
    }
    vectorNormalize(n, eigVector, eigValue);
    RowMatrixVectorMultiply(n, (double*)A, eigVector, nextVector);

    // While residue: ||Av - lv||, is not less than tolerance, continue iterations
    bool continueIterations = true;
    while (continueIterations) {
        if (rank == 0) {
            for (int i = 0; i < n; i++) {
                eigVector[i] = nextVector[i];
            }
            eigValue = vectorNorm(n, eigVector);        // Dont forget to norm iterated
                vector
            vectorNormalize(n, eigVector, eigValue);
        }
        else {
            residueValue = 0.0;
        }

        RowMatrixVectorMultiply(n, (double*)A, eigVector, nextVector);  // Next vector is
             matrix * current vector
        if (rank == 0) {
            // calculate residue vector
            for (int i = 0; i < n; i++) {
                residueVector[i] = nextVector[i] - eigValue * eigVector[i];
            }
            // calculate residue value
            residueValue = vectorNorm(n, residueVector);
            continueIterations = (residueValue > tolerance);
        }

        // Send message to slaves if stop or not
        if (rank == 0) {
            // send for every slave process
            for (int i = 0; i < size; i++) {
                if (i != rank) {
                    MPI_Send(&continueIterations, 1, MPI_CXX_BOOL, i, 0, MPI_COMM_WORLD);
                }
            }
        }
        else {
            MPI_Recv(&continueIterations, 1, MPI_CXX_BOOL, 0, 0, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
        }
    }
    // Print time and results info
    if (rank == 0) {
        total_time = MPI_Wtime() - start_time;
        cout << "MPI processes: " << size << ";    Value:   " << eigValue << "; Time
            taken :   " << total_time << endl;
    }
```

```
118
119     MPI_Finalize();
120     return 0;
121 }
122
123 // Classic (no MPI) multiplication
124 void matrixVectorMult(int dim, double* mat, double* vec, double* result) {
125     for (int i = 0; i < dim; i++) {
126         result[i] = 0.0;
127         for (int j = 0; j < dim; j++) {
128             result[i] += mat[i * dim + j] * vec[j];
129         }
130     }
131 }
132
133 // MPI matrix-vector multiplication
134 void RowMatrixVectorMultiply(int dim, double* matrix_data, double* vector_data, double*
        result) {
135     // Initialize variables and get size of communicator
136     int rank, size;
137     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
138     MPI_Comm_size(MPI_COMM_WORLD, &size);
139     // Initialize local matrix and vector
140     double* localresult = new double[dim / size]{};
141     double* matrix = new double[dim * dim / size]{};    //local matrix
142
143     MPI_Barrier(MPI_COMM_WORLD);
144     //Scatter the Matrix from master to slaves
145     MPI_Scatter(matrix_data, (dim * dim) / size, MPI_DOUBLE, matrix, (dim * dim) / size,
            MPI_DOUBLE, 0, MPI_COMM_WORLD);
146     // Send same vector from master to slaves
147     MPI_Bcast(vector_data, dim, MPI_DOUBLE, 0, MPI_COMM_WORLD);// Broadcast the Vector
148
149     //Calculate the results
150     for (int i = 0; i < (dim / size); i++)
151         for (int j = 0; j < dim; j++)
152             localresult[i] += vector_data[j] * matrix[i * dim + j];
153     // Gather separetely calculatet parts together
154     MPI_Gather(localresult, (dim) / size, MPI_DOUBLE, result, (dim) / size, MPI_DOUBLE,
            0, MPI_COMM_WORLD); // Gather the results
155 }
156
157 // Read pre-generated matrix from file
158 void read_mat_from_file(const char* s, int n_row, int n_col, double* in_matrix) {
159     std::ifstream fin(s);
160     std::string line;
161     double data_in;
162
163     //open the input stream
164     if (!fin)
165     {
166         cout << "Unable to open " << s << " for reading.\n";
167         exit(0);
168     }
```

```
169
170     //  cout << " for file: " << s << "\n" << endl;
171     for (int i = 0; i < n_row; i++) {
172         std::getline(fin, line);
173         std::stringstream stream(line);
174         for (int j = 0; j < n_col; j++) {
175             stream >> data_in;       //now read the whitespace-separated floats
176             *(in_matrix + (i * n_col) + j) = data_in;
177         }
178     }
179 }
180
181 // L2 vector norm
182 double vectorNorm(int dim, double* vec) {
183     double res = 0.0;
184     for (int i = 0; i < dim; i++) {
185         res += vec[i] * vec[i];
186     }
187     return sqrt(res);
188 }
189
190 // Normalize vector to 1
191 void vectorNormalize(int dim, double* vec, const double vecNorm) {
192     for (int i = 0; i < dim; i++) {
193         vec[i] = vec[i] / vecNorm;
194     }
195 }
```

8 Приложение 2. Таблицы результатов для вычислений

n - размерность матрицы, Threads - число потоков, Time - время работы алгоритма (мс).

```
1  Max number of threads is 6
2
3  n = 60
4  Threads  Time_1    Time_2    Time_3
5  1        0.00102   0.00101   0.00104
6  2        0.00019   0.00020   0.00019
7  3        0.00019   0.00020   0.00019
8  4        0.00023   0.00028   0.00057
9  5        0.00047   0.00029   0.00045
10 6        0.00033   0.00034   0.00032
11
12 n = 120
13 Threads  Time_1    Time_2    Time_3
14 1        0.00328   0.00317   0.00319
15 2        0.00212   0.00210   0.00208
16 3        0.00038   0.00038   0.00038
17 4        0.00052   0.00050   0.00041
18 5        0.00045   0.00036   0.00047
19 6        0.00048   0.00050   0.00039
20
21 n = 180
```

```
Threads  Time_1    Time_2    Time_3
1        0.00364   0.00393   0.00347
2        0.00091   0.00227   0.00229
3        0.00071   0.00146   0.00072
4        0.00088   0.00089   0.00063
5        0.00082   0.00085   0.00084
6        0.00086   0.00075   0.00080

n = 240
Threads  Time_1    Time_2    Time_3
1        0.00439   0.00435   0.00453
2        0.00266   0.00257   0.00260
3        0.00183   0.00120   0.00118
4        0.00146   0.00105   0.00099
5        0.00095   0.00091   0.00092
6        0.00119   0.00118   0.00090

n = 360
Threads  Time_1    Time_2    Time_3
1        0.00557   0.00572   0.00563
2        0.00291   0.00318   0.00327
3        0.00222   0.00237   0.00222
4        0.00286   0.00191   0.00189
5        0.00244   0.00243   0.00170
6        0.00213   0.00160   0.00216

n = 480
Threads  Time_1    Time_2    Time_3
1        0.00921   0.00907   0.00931
2        0.00518   0.00523   0.00527
3        0.00395   0.00396   0.00394
4        0.00485   0.00322   0.00323
5        0.00297   0.00300   0.00295
6        0.00369   0.00265   0.00265

n = 720
Threads  Time_1    Time_2    Time_3
1        0.01860   0.01807   0.01884
2        0.01063   0.01085   0.01101
3        0.00816   0.00815   0.00817
4        0.00683   0.00675   0.00689
5        0.00593   0.00599   0.00601
6        0.00560   0.00557   0.00747

n = 960
Threads  Time_1    Time_2    Time_3
1        0.03318   0.03336   0.03344
2        0.01957   0.01980   0.01986
3        0.01514   0.01479   0.01515
4        0.01238   0.01278   0.01251
5        0.01117   0.01128   0.01097
6        0.01021   0.01026   0.01040

n = 1020
```

Threads	Time_1	Time_2	Time_3
1	0.03835	0.03853	0.03782
2	0.02250	0.02235	0.02236
3	0.01718	0.01738	0.01716
4	0.01447	0.01424	0.01439
5	0.01278	0.01282	0.01283
6	0.01176	0.01186	0.01165