

МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»**

**ЧИСЛЕННОЕ РЕШЕНИЕ СИСТЕМ ЛИНЕЙНЫХ АЛГЕБРАИЧЕСКИХ
УРАВНЕНИЙ МЕТОДОМ ИСКЛЮЧЕНИЯ ГАУССА И
ИТЕРАЦИОННЫМИ МЕТОДАМИ**
ОТЧЕТ О ПРАКТИКЕ

Студента 3 курса 311 группы
направления 02.03.02 — Фундаментальная информатика и информационные
технологии
факультета КНиИТ
Аношкина Андрея Алексеевича

Проверил
Старший преподаватель

М. С. Портенко

СОДЕРЖАНИЕ

1	Work 04.....	3
---	--------------	---

1 Work 04

Задание

Задайте элементы больших матриц и векторов при помощи датчика случайных чисел. Отключите печать исходных матрицы и вектора и печать результирующего вектора (закомментируйте соответствующие строки кода). Проведите вычислительные эксперименты, результаты занесите в таблицу ??.

Насколько сильно отличаются время, затраченное на выполнение последовательного и параллельного алгоритма? Для матрицы какого размера было получено наилучшее значение ускорения? Почему?

Определение задачи решения системы линейных уравнений

Множество n линейных уравнений:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2 \\ \dots \\ a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n = b_n \end{cases}$$

называется системой линейных уравнений или линейной системой.

В более кратком (матричном) виде система может быть представлена как $Ax = b$, где $A = (a_{ij})$ есть вещественная матрица размера $n \times n$, а вектора b и x состоят из элементов.

Под задачей решения системы линейных уравнений для заданных матрицы A и вектора b обычно понимается нахождение значения вектора неизвестных x , при котором выполняются все уравнения системы.

Метод Гаусса

- Основная идея: приведение матрицы A к верхнему треугольному виду с помощью эквивалентных преобразований.
- Эквивалентные преобразования:
 - умножение уравнения на ненулевую константу;
 - перестановка уравнений;
 - суммирование уравнения с любым другим уравнением системы.

Метод Гаусса включает последовательное выполнение двух этапов. На первом этапе — прямой ход метода Гаусса — исходная система линейных урав-

нений при помощи последовательного исключения неизвестных приводится к верхнему треугольному виду.

На обратном ходе метода Гаусса (второй этап алгоритма) осуществляется определение значений неизвестных. Из последнего уравнения преобразованной системы может быть вычислено значение переменной, после этого из предпоследнего уравнения становится возможным определение переменной x_{n-1} и т. д.

Прямой ход метода Гаусса

- На итерации i , $0 \leq i < n$, метода производится исключение неизвестной i для всех уравнений с номерами k , больших i ($i \leq k < n$). Для этого из этих уравнений осуществляется вычитание строки i , умноженной на константу (a_{ki}/a_{ii}) , чтобы результирующий коэффициент при неизвестной x_i в строках оказался нулевым.
- Все необходимые вычисления определяются при помощи соотношений:

$$\begin{cases} a'_{kj} = a_{kj} - (a_{ki}/a_{ii})a_{ij} \\ b'_k = b_k - (a_{ki}/a_{ii})b_i \\ i \leq j < n, i < k \leq n, 0 \leq i < n \end{cases}$$

Обратный ход метода Гаусса

После приведения матрицы коэффициентов к треугольному виду становится возможным определение значений неизвестных:

- Из последнего уравнения преобразованной системы может быть вычислено значение переменной x_n .
- Из предпоследнего уравнения становится возможным определение переменной x_{n-1} , и т. д.

В общем виде, выполняемые вычисления при обратном ходе метода Гаусса могут быть представлены при помощи соотношений:

$$\begin{cases} x_n = b_n/a_{nn}, \\ x_i = (b_i - \sum_{j=i+1}^n a_{ij}x_j)/a_{ii}, i = n-1, n-2, \dots, 0. \end{cases}$$

Выбор ведущего элемента

Описанный алгоритм применим, только если ведущие элементы отличны от нуля, т. е. $a_{ii} \neq 0$.

- Рассмотрим k -й шаг алгоритма. Пусть $s = \max |a_{kk}|, |a_{k+1k}|, \dots, |a_{nk}|$
- Тогда переставим s -ю и k -ю строки матрицы (выбор ведущего элемента по столбцу).
- В итоге получаем систему $PAx = Pb$, где P — матрица перестановки.

Последовательная реализация

Фрагмент кода решения приведен ниже:

```
1 // SerialGauss.cpp
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <conio.h>
5 #include <time.h>
6 #include <math.h>
7
8 int* pSerialPivotPos; // The Number of pivot rows selected at the iterations
9 int* pSerialPivotIter; // The Iterations, at which the rows were pivots
10
11 // Function for simple initialization of the matrix
12 // and the vector elements
13 void DummyDataInitialization(double* pMatrix, double* pVector, int
14     Size) {
15     for (int i = 0; i < Size; ++i) {
16         pVector[i] = i + 1.0;
17         for (int j = 0; j < Size; ++j)
18             if (j <= i)
19                 pMatrix[i * Size + j] = 1;
20             else
21                 pMatrix[i * Size + j] = 0;
22     }
23 }
24
25 // Function for random initialization of the matrix
26 // and the vector elements
27 void RandomDataInitialization(double* pMatrix, double* pVector,
28     int Size) {
29     srand(unsigned(clock()));
30     for (int i = 0; i < Size; ++i) {
31         pVector[i] = rand() / double(1000);
32         for (int j = 0; j < Size; ++j)
33             if (j <= i)
34                 pMatrix[i * Size + j] = rand() / double(1000);
35             else
```

```

36         pMatrix[i * Size + j] = 0;
37     }
38 }
39
40 // Function for memory allocation and definition of the objectselements
41 void ProcessInitialization(double*& pMatrix, double*
42     & pVector,
43     double*& pResult, int& Size) {
44     // Setting the size of the matrix and the vector
45     do {
46         printf("\nEnter size of the matrix and the vector: ");
47         scanf_s("%d", &Size);
48         printf("\nChosen size = %d \n", Size);
49         if (Size <= 0)
50             printf("\nSize of objects must be greater than 0!\n");
51     } while (Size <= 0);
52
53     // Memory allocation
54     pMatrix = new double[Size * Size];
55     pVector = new double[Size];
56     pResult = new double[Size];
57
58     // Initialization of the matrix and the vector elements
59     //DummyDataInitialization(pMatrix, pVector, Size);
60     RandomDataInitialization(pMatrix, pVector, Size);
61 }
62
63 // Function for formatted matrix output
64 void PrintMatrix(double* pMatrix, int RowCount, int ColCount) {
65     for (int i = 0; i < RowCount; ++i) {
66         for (int j = 0; j < ColCount; ++j)
67             printf("%7.4f ", pMatrix[i * RowCount + j]);
68         printf("\n");
69     }
70 }
71
72 // Function for formatted vector output
73 void PrintVector(double* pVector, int Size) {
74     for (int i = 0; i < Size; ++i)
75         printf("%7.4f ", pVector[i]);
76 }
77
78 // Finding the pivot row
79 int FindPivotRow(double* pMatrix, int Size, int Iter) {
80     int PivotRow = -1; // The index of the pivot row
81     int MaxValue = 0; // The value of the pivot element
82
83     // Choose the row, that stores the maximum element
84     for (int i = 0; i < Size; ++i) {
85         if ((pSerialPivotIter[i] == -1) &&

```

```

86         (fabs(pMatrix[i * Size + Iter]) > MaxValue)) {
87             PivotRow = i;
88             MaxValue = fabs(pMatrix[i * Size + Iter]);
89         }
90     }
91     return PivotRow;
92 }
93
94 // Column elimination
95 void SerialColumnElimination(double* pMatrix, double* pVector,
96     int Pivot, int Iter, int Size) {
97     double PivotValue, PivotFactor;
98     PivotValue = pMatrix[Pivot * Size + Iter];
99     for (int i = 0; i < Size; i++)
100         if (pSerialPivotIter[i] == -1) {
101             PivotFactor = pMatrix[i * Size + Iter] / PivotValue;
102             for (int j = Iter; j < Size; j++)
103                 pMatrix[i * Size + j] -= PivotFactor * pMatrix[Pivot * Size + j];
104             pVector[i] -= PivotFactor * pVector[Pivot];
105         }
106 }
107
108 // Gaussian elimination
109 void SerialGaussianElimination(double* pMatrix, double* pVector, int
110     Size) {
111     int PivotRow; // The number of the current pivot row
112     for (int Iter = 0; Iter < Size; ++Iter) {
113         // Finding the pivot row
114         PivotRow = FindPivotRow(pMatrix, Size, Iter);
115         pSerialPivotPos[Iter] = PivotRow;
116         pSerialPivotIter[PivotRow] = Iter;
117         SerialColumnElimination(pMatrix, pVector, PivotRow, Iter, Size);
118     }
119 }
120
121 // Back substitution
122 void SerialBackSubstitution(double* pMatrix, double* pVector,
123     double* pResult, int Size) {
124     int RowIndex, Row;
125     for (int i = Size - 1; i >= 0; --i) {
126         RowIndex = pSerialPivotPos[i];
127         pResult[i] = pVector[RowIndex] / pMatrix[Size * RowIndex + i];
128         for (int j = 0; j < i; ++j) {
129             Row = pSerialPivotPos[j];
130             pVector[Row] -= pMatrix[Row * Size + i] * pResult[i];
131             pMatrix[Row * Size + i] = 0;
132         }
133     }
134 }
135

```

```

136 // Function for the execution of Gauss algorithm
137 void SerialResultCalculation(double* pMatrix, double* pVector,
138     double* pResult, int Size) {
139     // Memory allocation
140     pSerialPivotPos = new int[Size];
141     pSerialPivotIter = new int[Size];
142
143     for (int i = 0; i < Size; pSerialPivotIter[i] = -1, ++i);
144
145     // Gaussian elimination
146     SerialGaussianElimination(pMatrix, pVector, Size);
147     // Back substitution
148     SerialBackSubstitution(pMatrix, pVector, pResult, Size);
149
150     // Memory deallocation
151     delete[] pSerialPivotPos;
152     delete[] pSerialPivotIter;
153 }
154
155 // Function for computational process termination
156 void ProcessTermination(double* pMatrix, double* pVector, double*
157     pResult) {
158     delete[] pMatrix;
159     delete[] pVector;
160     delete[] pResult;
161 }
162
163 int main() {
164     double* pMatrix; // The matrix of the linear system
165     double* pVector; // The right parts of the linear system
166     double* pResult; // The result vector
167     int Size; // The sizes of the initial matrix and the vector
168     double start, finish, duration;
169     printf("Serial Gauss algorithm for solving linear systems\n");
170
171     // Memory allocation and definition of objects' elements
172     ProcessInitialization(pMatrix, pVector, pResult, Size);
173
174     // The matrix and the vector output
175     printf("Initial Matrix \n");
176     PrintMatrix(pMatrix, Size, Size);
177     printf("Initial Vector \n");
178     PrintVector(pVector, Size);
179
180     // Execution of Gauss algorithm
181     start = clock();
182     SerialResultCalculation(pMatrix, pVector, pResult, Size);
183     finish = clock();
184     duration = (finish - start) / CLOCKS_PER_SEC;
185

```



```

186 // Printing the result vector
187 printf("\n Result Vector: \n");
188 PrintVector(pResult, Size);
189
190 // Printing the execution time of Gauss method
191 printf("\n Time of execution: %f\n", duration);
192
193 // Computational process termination
194 ProcessTermination(pMatrix, pVector, pResult);
195
196 return 0;
197 }

```

Параллельная реализация

Фрагмент кода решения приведен ниже:

```

1 // SerialGauss.cpp
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <conio.h>
5 #include <time.h>
6 #include <math.h>
7 #include <omp.h>
8
9 int* pPivotPos; // The Number of pivot rows selected at the iterations
10 int* pPivotIter; // The Iterations, at which the rows were pivots
11
12 typedef struct {
13     int PivotRow;
14     double MaxValue;
15 } TThreadPivotRow;
16
17 // Function for simple initialization of the matrix
18 // and the vector elements
19 void DummyDataInitialization(double* pMatrix, double* pVector, int Size) {
20     for (int i = 0; i < Size; ++i) {
21         pVector[i] = i + 1.0;
22         for (int j = 0; j < Size; ++j)
23             if (j <= i)
24                 pMatrix[i * Size + j] = 1;
25             else
26                 pMatrix[i * Size + j] = 0;
27     }
28 }
29
30 // Function for random initialization of the matrix
31 // and the vector elements
32 void RandomDataInitialization(double* pMatrix, double* pVector, int Size) {
33     srand(unsigned(clock()));

```

```

34     for (int i = 0; i < Size; ++i) {
35         pVector[i] = rand() / double(1000);
36         for (int j = 0; j < Size; ++j)
37             if (j <= i)
38                 pMatrix[i * Size + j] = rand() / double(1000);
39             else
40                 pMatrix[i * Size + j] = 0;
41     }
42 }
43
44 // Function for memory allocation and definition of the objectselements
45 void ProcessInitialization(double*& pMatrix, double*& pVector, double*& pResult, int& Size) {
46     // Setting the size of the matrix and the vector
47     do {
48         printf("\nEnter size of the matrix and the vector: ");
49         scanf_s("%d", &Size);
50         printf("\nChosen size = %d \n", Size);
51         if (Size <= 0)
52             printf("\nSize of objects must be greater than 0!\n");
53     } while (Size <= 0);
54
55     // Memory allocation
56     pMatrix = new double[Size * Size];
57     pVector = new double[Size];
58     pResult = new double[Size];
59
60     // Initialization of the matrix and the vector elements
61     //DummyDataInitialization(pMatrix, pVector, Size);
62     RandomDataInitialization(pMatrix, pVector, Size);
63 }
64
65 // Function for formatted matrix output
66 void PrintMatrix(double* pMatrix, int RowCount, int ColCount) {
67     for (int i = 0; i < RowCount; ++i) {
68         for (int j = 0; j < ColCount; ++j)
69             printf("%7.4f ", pMatrix[i * RowCount + j]);
70         printf("\n");
71     }
72 }
73
74 // Function for formatted vector output
75 void PrintVector(double* pVector, int Size) {
76     for (int i = 0; i < Size; ++i)
77         printf("%7.4f ", pVector[i]);
78 }
79
80 // Finding the pivot row
81 int ParallelFindPivotRow(double* pMatrix, int Size, int Iter) {
82     int PivotRow = -1; // The index of the pivot row
83     int MaxValue = 0; // The value of the pivot element

```

```

84
85     #pragma omp parallel
86     {
87         TThreadPivotRow ThreadPivotRow;
88         ThreadPivotRow.MaxValue = 0;
89         ThreadPivotRow.PivotRow = -1;
90         // Choose the row, that stores the maximum element
91         #pragma omp for
92         for (int i = 0; i < Size; ++i) {
93             if ((pPivotIter[i] == -1) && (fabs(pMatrix[i * Size + Iter]) > ThreadPivotRow.MaxValue))
94                 ↪ {
95                     ThreadPivotRow.PivotRow = i;
96                     ThreadPivotRow.MaxValue = fabs(pMatrix[i * Size + Iter]);
97                 }
98         }
99         #pragma omp critical
100        {
101            if (ThreadPivotRow.MaxValue > MaxValue) {
102                MaxValue = ThreadPivotRow.MaxValue;
103                PivotRow = ThreadPivotRow.PivotRow;
104            }
105        }
106    }
107    return PivotRow;
108 }
109
110 // Column elimination
111 void ParallelColumnElimination(double* pMatrix, double* pVector, int Pivot, int Iter, int Size) {
112     double PivotValue, PivotFactor;
113     PivotValue = pMatrix[Pivot * Size + Iter];
114     #pragma omp parallel for private(PivotFactor) schedule(dynamic, 1)
115     for (int i = 0; i < Size; ++i)
116         if (pPivotIter[i] == -1) {
117             PivotFactor = pMatrix[i * Size + Iter] / PivotValue;
118             for (int j = Iter; j < Size; ++j)
119                 pMatrix[i * Size + j] -= PivotFactor * pMatrix[Pivot * Size + j];
120             pVector[i] -= PivotFactor * pVector[Pivot];
121         }
122 }
123
124 // Gaussian elimination
125 void ParallelGaussianElimination(double* pMatrix, double* pVector, int Size) {
126     int PivotRow; // The number of the current pivot row
127     for (int Iter = 0; Iter < Size; ++Iter) {
128         // Finding the pivot row
129         PivotRow = ParallelFindPivotRow(pMatrix, Size, Iter);
130         pPivotPos[Iter] = PivotRow;
131         pPivotIter[PivotRow] = Iter;
132         ParallelColumnElimination(pMatrix, pVector, PivotRow, Iter, Size);

```

```

133     }
134 }
135
136 // Back substitution
137 void ParallelBackSubstitution(double* pMatrix, double* pVector, double* pResult, int Size) {
138     int RowIndex, Row;
139     for (int i = Size - 1; i >= 0; --i) {
140         RowIndex = pPivotPos[i];
141         pResult[i] = pVector[RowIndex] / pMatrix[Size * RowIndex + i];
142         #pragma omp parallel for private(Row)
143         for (int j = 0; j < i; ++j) {
144             Row = pPivotPos[j];
145             pVector[Row] -= pMatrix[Row * Size + i] * pResult[i];
146             pMatrix[Row * Size + i] = 0;
147         }
148     }
149 }
150
151 // Function for the execution of Gauss algorithm
152 void ParallelResultCalculation(double* pMatrix, double* pVector,
153     double* pResult, int Size) {
154     // Memory allocation
155     pPivotPos = new int[Size];
156     pPivotIter = new int[Size];
157
158     for (int i = 0; i < Size; pPivotIter[i] = -1, ++i);
159
160     // Gaussian elimination
161     ParallelGaussianElimination(pMatrix, pVector, Size);
162     // Back substitution
163     ParallelBackSubstitution(pMatrix, pVector, pResult, Size);
164
165     // Memory deallocation
166     delete[] pPivotPos;
167     delete[] pPivotIter;
168 }
169
170 // Function for computational process termination
171 void ProcessTermination(double* pMatrix, double* pVector, double*
172     pResult) {
173     delete[] pMatrix;
174     delete[] pVector;
175     delete[] pResult;
176 }
177
178 // Function for testing the result
179 void TestResult(double* pMatrix, double* pVector, double* pResult, int Size) {
180     /* Buffer for storing the vector, that is a result of multiplication
181     of the linear system matrix by the vector of unknowns */
182     double* pRightPartVector;

```

```

183
184 // Flag, that shows wheather the right parts vectors are identical or not
185 int equal = 0;
186 double Accuracy = 1e-2; // Comparison accuracy
187 pRightPartVector = new double[Size];
188 for (int i = 0; i < Size; ++i) {
189     pRightPartVector[i] = 0;
190     for (int j = 0; j < Size; ++j)
191         pRightPartVector[i] += pMatrix[i * Size + j] * pResult[j];
192 }
193
194 for (int i = 0; i < Size; i++)
195     if (fabs(pRightPartVector[i] - pVector[i]) > Accuracy)
196         equal = 1;
197
198 if (equal == 1)
199     printf("\nThe result of the parallel Gauss algorithm is NOT correct. Check your code.");
200 else
201     printf("The result of the parallel Gauss algorithm is correct.");
202
203 delete[] pRightPartVector;
204 }
205 int main() {
206     double* pMatrix; // The matrix of the linear system
207     double* pVector; // The right parts of the linear system
208     double* pResult; // The result vector
209     int Size; // The sizes of the initial matrix and the vector
210     double start, finish, duration;
211     printf("Parallel Gauss algorithm for solving linear systems\n");
212
213     // Memory allocation and definition of objects' elements
214     ProcessInitialization(pMatrix, pVector, pResult, Size);
215
216     // The matrix and the vector output
217     /*printf("Initial Matrix \n");
218     PrintMatrix(pMatrix, Size, Size);
219     printf("Initial Vector \n");
220     PrintVector(pVector, Size);*/
221
222     // Execution of Gauss algorithm
223     start = clock();
224     ParallelResultCalculation(pMatrix, pVector, pResult, Size);
225     finish = clock();
226     duration = (finish - start) / CLOCKS_PER_SEC;
227
228     // Testing the result
229     TestResult(pMatrix, pVector, pResult, Size);
230
231     // Printing the result vector
232     /*printf("\n Result Vector: \n");

```

```

233     PrintVector(pResult, Size);*/
234
235     // Printing the execution time of Gauss method
236     printf("\nTime of execution: %f\n", duration);
237
238     // Computational process termination
239     ProcessTermination(pMatrix, pVector, pResult);
240
241     return 0;
242 }

```

Результат работы

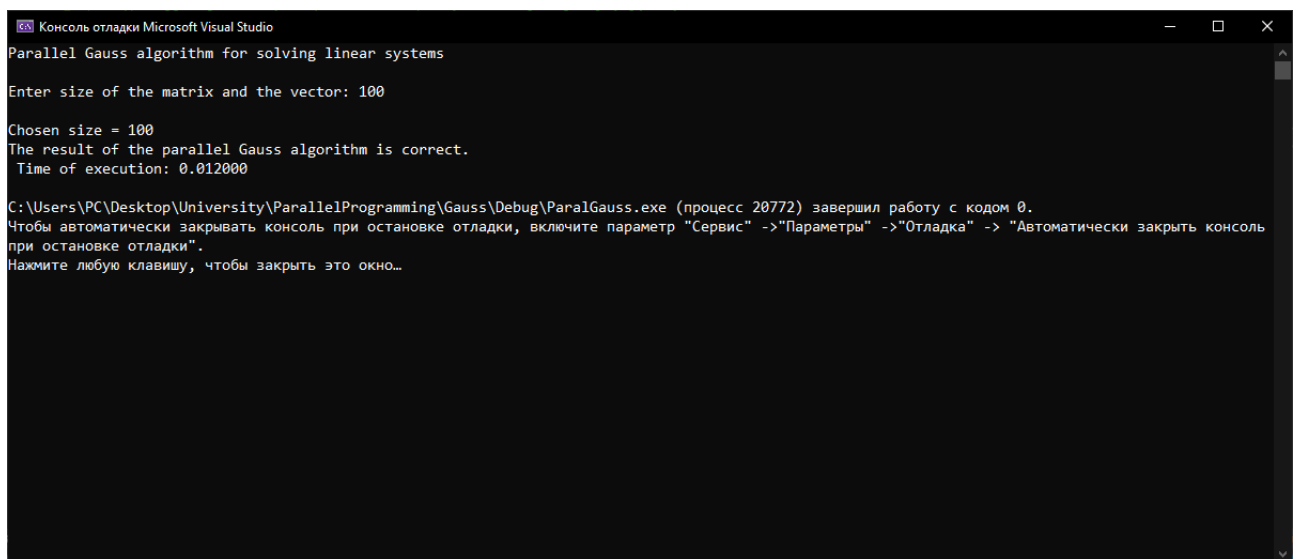


Рисунок 1 – Work-4

Таблица сравнения

Номер теста	Порядок системы	Последовательный алгоритм	Параллельный алгоритм	
			Время	Ускорение
1	10	0.000000	0.004000	$\approx \infty$
2	100	0.001000	0.007000	≈ 0.14
3	500	0.115000	0.058000	≈ 1.98
4	1000	0.934000	0.260000	≈ 3.59
5	1500	3.215000	0.899000	≈ 3.57
6	2000	7.592000	1.799000	≈ 4.22
7	2500	14.792000	3.388000	≈ 4.36
8	3000	25.848000	6.242000	≈ 4.14

Начиная с 500, ускорение стало достигать значений в 2 раза и более. При небольшом порядке системы уравнений последовательная реализация выигрывает в скорости перед параллельной в связи со временем, затрачиваемым параллельной реализацией для подготовки потоков.

Характеристики устройства

Процессор: Intel(R) Core(TM) i5-10400F

Ядер: 6

Оперативная память: 16 Гб