

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 13.....	3
---	--------------	---

1 Work 13

Задание

Аналогично работе с OMP выполните следующее задание через MPI.

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 1.

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 уравнений 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 "mpi.h"
8 #include <iostream>
9
10 int NProc, ProcId;
11 MPI_Status st;
12
13 // Function for formatted matrix output
14 void PrintMatrix(double* pMatrix, int RowCount, int ColCount) {
15     for (int i = 0; i < RowCount; ++i) {
16         for (int j = 0; j < ColCount; ++j)
17             printf("%7.4f ", pMatrix[i * RowCount + j]);
18         printf("\n");
19     }
20 }
21
22 // Function for formatted vector output
23 void PrintVector(double* pVector, int Size) {
24     for (int i = 0; i < Size; ++i)
25         printf("%7.4f ", pVector[i]);
26 }
27
28
29 // Function for simple initialization of the matrix
30 // and the vector elements
31 void DummyDataInitialization(double* pMatrix, double* pVector, int Size) {
32     for (int i = 0; i < Size; ++i) {
33         pVector[i] = i + 1.0;

```

```

34         for (int j = 0; j < Size; ++j)
35             if (j <= i)
36                 pMatrix[i * Size + j] = 1;
37             else
38                 pMatrix[i * Size + j] = 0;
39     }
40 }
41
42 // Function for random initialization of the matrix
43 // and the vector elements
44 void RandomDataInitialization(double* pMatrix, double* pVector, int Size) {
45     srand(unsigned(clock()));
46     for (int i = 0; i < Size; ++i) {
47         pVector[i] = rand() / double(1000);
48         for (int j = 0; j < Size; ++j)
49             pMatrix[i * Size + j] = rand() / double(1000);
50     }
51 }
52
53 void MyDataInitialization(double* pMatrix, double* pVector, int Size) {
54     pMatrix[0] = 1; pMatrix[1] = 1; pMatrix[2] = 4; pMatrix[3] = 4; pMatrix[4] = 9; pVector[0] = -9;
55     pMatrix[5] = 2; pMatrix[6] = 2; pMatrix[7] = 17; pMatrix[8] = 17; pMatrix[9] = 82; pVector[1] = -146;
56     pMatrix[10] = 2; pMatrix[11] = 0; pMatrix[12] = 3; pMatrix[13] = -1; pMatrix[14] = 4; pVector[2] = -10;
57     pMatrix[15] = 0; pMatrix[16] = 1; pMatrix[17] = 4; pMatrix[18] = 12; pMatrix[19] = 27; pVector[3] =
58     ↪ -26;
59     pMatrix[20] = 1; pMatrix[21] = 2; pMatrix[22] = 2; pMatrix[23] = 10; pMatrix[24] = 0; pVector[4] = 37;
60 }
61
62 // Function for memory allocation and definition of the objectselements
63 void ProcessInitialization(double*& pMatrix, double*& pVector, double*& pResult,
64     double*& pPartialMatrix, double*& pPartialVector, int& Size, int& PartialSize,
65     int& AdditionalSize, double*& pAdditionalMatrix, double*& pAdditionalVector) {
66     MPI_Barrier(MPI_COMM_WORLD);
67
68     Size = 1000;
69
70     // Memory allocation
71     pMatrix = new double[Size * Size];
72     pVector = new double[Size];
73     pResult = new double[Size];
74
75     PartialSize = Size / NProc;
76
77     // Partial arrays memory allocation
78     pPartialMatrix = new double[Size * PartialSize];
79     pPartialVector = new double[PartialSize];
80
81     AdditionalSize = ((ProcId > 0) && (ProcId <= (Size % NProc)) && ((Size % NProc) > 0));
82
83     // Additional arrays memory allocation

```

```

83     pAdditionalMatrix = new double[Size * AdditionalSize];
84     pAdditionalVector = new double[AdditionalSize];
85
86     if (ProcId == 0) {
87         // Initialization of the matrix and the vector elements
88         //DummyDataInitialization(pMatrix, pVector, Size);
89         RandomDataInitialization(pMatrix, pVector, Size);
90         //MyDataInitialization(pMatrix, pVector, Size);
91     }
92
93     MPI_Scatter(pMatrix, Size * PartialSize, MPI_DOUBLE, pPartialMatrix, Size * PartialSize,
94     ↪ MPI_DOUBLE, 0, MPI_COMM_WORLD);
95     MPI_Scatter(pVector, PartialSize, MPI_DOUBLE, pPartialVector, PartialSize, MPI_DOUBLE, 0,
96     ↪ MPI_COMM_WORLD);
97
98     if (Size % NProc) {
99         if (ProcId == 0)
100             for (int i = 0; i < Size % NProc; ++i) {
101                 MPI_Send(pMatrix + Size * (PartialSize * NProc + i), Size, MPI_DOUBLE, i + 1,
102                 ↪ 0, MPI_COMM_WORLD);
103                 MPI_Send(pVector + (PartialSize * NProc + i), 1, MPI_DOUBLE, i + 1, 0,
104                 ↪ MPI_COMM_WORLD);
105             }
106         else if (AdditionalSize) {
107             MPI_Recv(pAdditionalMatrix, Size, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &st);
108             MPI_Recv(pAdditionalVector, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &st);
109         }
110     }
111
112     // Finding the pivot row
113     int ParallelFindPivotRow(double* pPartialMatrix, int PartialSize, double* pAdditionalMatrix, int
114     ↪ AdditionalSize, int Iter, int* pPivotIter, int Size) {
115         MPI_Barrier(MPI_COMM_WORLD);
116         int PivotRow = -1; // The index of the pivot row
117         int MaxValue = 0; // The value of the pivot element
118
119         for (int i = 0; i < PartialSize; ++i) {
120             if ((pPivotIter[i + PartialSize * ProcId] == -1) && (fabs(pPartialMatrix[i * Size + Iter]) >=
121             ↪ MaxValue)) {
122                 PivotRow = i + PartialSize * ProcId;
123                 MaxValue = fabs(pPartialMatrix[i * Size + Iter]);
124             }
125         }
126
127         if (AdditionalSize) {
128             if (pPivotIter[PartialSize * NProc + ProcId - 1] == -1 && fabs(pAdditionalMatrix[Iter]) >=
129             ↪ MaxValue) {
130                 PivotRow = PartialSize * NProc + ProcId - 1;
131                 MaxValue = fabs(pAdditionalMatrix[Iter]);
132             }
133         }
134     }

```

```

126         }
127     }
128
129     if (ProcId != 0) {
130         MPI_Send(&MaxValue, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
131         MPI_Send(&PivotRow, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
132     }
133     else {
134         for (int i = 1; i < NProc; ++i) {
135             int TMaxValue, TPivotRow;
136             MPI_Recv(&TMaxValue, 1, MPI_INT, i, 0, MPI_COMM_WORLD, &st);
137             MPI_Recv(&TPivotRow, 1, MPI_INT, i, 0, MPI_COMM_WORLD, &st);
138             if (TMaxValue > MaxValue) {
139                 MaxValue = TMaxValue;
140                 PivotRow = TPivotRow;
141             }
142         }
143     }
144
145     MPI_Bcast(&PivotRow, 1, MPI_INT, 0, MPI_COMM_WORLD);
146
147     return PivotRow;
148 }
149
150 // Column elimination
151 void ParallelColumnElimination(double* pPartialMatrix, double* pPartialVector, double* pAdditionalMatrix,
    ↪ double* pAdditionalVector, int Pivot, int Iter, int Size, int PartialSize, int AdditionalSize, int*
    ↪ pPivotIter) {
152     MPI_Barrier(MPI_COMM_WORLD);
153     double PivotValue, PivotFactor;
154     double* PivotRow = new double[Size - Iter];
155
156     if (Pivot <= NProc * PartialSize - 1) {
157         if (Pivot >= ProcId * PartialSize && Pivot < (ProcId + 1) * PartialSize) {
158             PivotValue = pPartialVector[(Pivot - ProcId * PartialSize)];
159             for (int i = 0; i < Size - Iter; ++i)
160                 PivotRow[i] = pPartialMatrix[(Pivot - ProcId * PartialSize) * Size + Iter + i];
161             for (int i = 0; i < NProc; ++i)
162                 if (i != ProcId) {
163                     MPI_Send(PivotRow, Size - Iter, MPI_DOUBLE, i, 0,
    ↪ MPI_COMM_WORLD);
164                     MPI_Send(&PivotValue, 1, MPI_DOUBLE, i, 1, MPI_COMM_WORLD);
165                 }
166         }
167         else {
168             MPI_Recv(PivotRow, Size - Iter, MPI_DOUBLE, MPI_ANY_SOURCE, 0,
    ↪ MPI_COMM_WORLD, &st);
169             MPI_Recv(&PivotValue, 1, MPI_DOUBLE, MPI_ANY_SOURCE, 1,
    ↪ MPI_COMM_WORLD, &st);
170         }

```

```

171     }
172     else
173         if (Pivot == NProc * PartialSize + ProcId - 1) {
174             PivotValue = pAdditionalVector[0];
175             for (int i = 0; i < Size - Iter; ++i)
176                 PivotRow[i] = pAdditionalMatrix[Iter + i];
177             for (int i = 0; i < NProc; ++i)
178                 if (i != ProcId) {
179                     MPI_Send(PivotRow, Size - Iter, MPI_DOUBLE, i, 0,
180                             ↪ MPI_COMM_WORLD);
181                     MPI_Send(&PivotValue, 1, MPI_DOUBLE, i, 1, MPI_COMM_WORLD);
182                 }
183             }
184             else {
185                 MPI_Recv(PivotRow, Size - Iter, MPI_DOUBLE, MPI_ANY_SOURCE, 0,
186                         ↪ MPI_COMM_WORLD, &st);
187                 MPI_Recv(&PivotValue, 1, MPI_DOUBLE, MPI_ANY_SOURCE, 1,
188                         ↪ MPI_COMM_WORLD, &st);
189             }
190         }
191         for (int i = 0; i < PartialSize; ++i) {
192             if (pPivotIter[i + PartialSize * ProcId] == -1) {
193                 PivotFactor = pPartialMatrix[i * Size + Iter] / PivotRow[0];
194                 for (int j = Iter; j < Size; ++j)
195                     pPartialMatrix[i * Size + j] -= PivotFactor * PivotRow[j - Iter];
196                 pPartialVector[i] -= PivotFactor * PivotValue;
197             }
198         }
199         if (AdditionalSize) {
200             if (pPivotIter[NProc * PartialSize + ProcId - 1] == -1) {
201                 PivotFactor = pAdditionalMatrix[Iter] / PivotRow[0];
202                 for (int j = Iter; j < Size; ++j)
203                     pAdditionalMatrix[j] -= PivotFactor * PivotRow[j - Iter];
204                 pAdditionalVector[0] -= PivotFactor * PivotValue;
205             }
206         }
207     }
208     // Gaussian elimination
209     void ParallelGaussianElimination(double* pPartialMatrix, double* pPartialVector, int PartialSize,
210                                     double* pAdditionalMatrix, double* pAdditionalVector, int AdditionalSize, int Size, int* pPivotPos,
211                                     ↪ int* pPivotIter) {
212         //MPI_Barrier(MPI_COMM_WORLD);
213         int PivotRow; // The number of the current pivot row
214         for (int Iter = 0; Iter < Size; ++Iter) {
215             // Finding the pivot row
216             PivotRow = ParallelFindPivotRow(pPartialMatrix, PartialSize, pAdditionalMatrix,
217                                             ↪ AdditionalSize, Iter, pPivotIter, Size);
218             pPivotPos[Iter] = PivotRow;

```

```

216         pPivotIter[PivotRow] = Iter;
217
218         /*if (ProcId == 0) {
219             for (int i = 0; i < Size; ++i)
220                 std::cout << pPivotPos[i] << " ";
221             std::cout << "\n";
222         }*/
223
224         ParallelColumnElimination(pPartialMatrix, pPartialVector, pAdditionalMatrix,
225             ↪ pAdditionalVector, PivotRow, Iter, Size, PartialSize, AdditionalSize, pPivotIter);
226     }
227
228     // Back substitution
229     void ParallelBackSubstitution(double* pMatrix, double* pVector, double* pResult, int Size, int* pPivotPos) {
230         int RowIndex, Row;
231         for (int i = Size - 1; i >= 0; --i) {
232             RowIndex = pPivotPos[i];
233             pResult[i] = pVector[RowIndex] / pMatrix[Size * RowIndex + i];
234             for (int j = 0; j < i; ++j) {
235                 Row = pPivotPos[j];
236                 pVector[Row] -= pMatrix[Row * Size + i] * pResult[i];
237                 pMatrix[Row * Size + i] = 0;
238             }
239         }
240     }
241
242     // Function for the execution of Gauss algorithm
243     void ParallelResultCalculation(double* pMatrix, double* pVector, double* pResult, int Size,
244         double* pPartialMatrix, double* pPartialVector, int PartialSize,
245         double* pAdditionalMatrix, double* pAdditionalVector, int AdditionalSize, int*& pPivotPos, int*&
246         ↪ pPivotIter) {
247         MPI_Barrier(MPI_COMM_WORLD);
248         // Memory allocation
249         pPivotPos = new int[Size];
250         pPivotIter = new int[Size];
251
252         for (int i = 0; i < Size; pPivotIter[i] = -1, ++i);
253
254         // Gaussian elimination
255         ParallelGaussianElimination(pPartialMatrix, pPartialVector, PartialSize, pAdditionalMatrix,
256             ↪ pAdditionalVector, AdditionalSize, Size, pPivotPos, pPivotIter);
257
258         MPI_Barrier(MPI_COMM_WORLD);
259
260         MPI_Gather(pPartialMatrix, Size * PartialSize, MPI_DOUBLE, pMatrix, Size * PartialSize,
261             ↪ MPI_DOUBLE, 0, MPI_COMM_WORLD);
262         MPI_Gather(pPartialVector, PartialSize, MPI_DOUBLE, pVector, PartialSize, MPI_DOUBLE, 0,
263             ↪ MPI_COMM_WORLD);

```

```

261     if (ProcId == 0) {
262         for (int i = 0; i < Size % NProc; ++i) {
263             MPI_Recv(pMatrix + PartialSize * NProc * Size + i * Size, Size, MPI_DOUBLE, i + 1, 0,
264                     ↪ MPI_COMM_WORLD, &st);
265             MPI_Recv(pVector + PartialSize * NProc + i, 1, MPI_DOUBLE, i + 1, 0,
266                     ↪ MPI_COMM_WORLD, &st);
267         }
268     }
269     else if (AdditionalSize) {
270         MPI_Send(pAdditionalMatrix, Size, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
271         MPI_Send(pAdditionalVector, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
272     }
273
274     /*if (ProcId == 0)
275         PrintMatrix(pMatrix, Size, Size);*/
276
277     // Back substitution
278     if (ProcId == 0)
279         ParallelBackSubstitution(pMatrix, pVector, pResult, Size, pPivotPos);
280 }
281
282 // Function for testing the result
283 void TestResult(double* pMatrix, double* pVector, double* pResult, int Size) {
284     /* Buffer for storing the vector, that is a result of multiplication
285     of the linear system matrix by the vector of unknowns */
286     double* pRightPartVector;
287
288     // Flag, that shows wheather the right parts vectors are identical or not
289     int equal = 0;
290     double Accuracy = 1e-3; // Comparison accuracy
291     pRightPartVector = new double[Size];
292     for (int i = 0; i < Size; ++i) {
293         pRightPartVector[i] = 0;
294         for (int j = 0; j < Size; ++j)
295             pRightPartVector[i] += pMatrix[i * Size + j] * pResult[j];
296     }
297
298     for (int i = 0; i < Size; i++)
299         if (fabs(pRightPartVector[i] - pVector[i]) > Accuracy)
300             equal = 1;
301
302     if (equal == 1)
303         printf("\nThe result of the parallel Gauss algorithm is NOT correct. Check your code.");
304     else
305         printf("\nThe result of the parallel Gauss algorithm is correct.");
306
307     delete[] pRightPartVector;
308 }
309
310 int main() {

```

```

309 double* pMatrix; // The matrix of the linear system
310 double* pVector; // The right parts of the linear system
311 double* pResult; // The result vector
312 int Size; // The sizes of the initial matrix and the vector
313 double start, finish, duration;
314
315 int PartialSize;
316 double* pPartialMatrix;
317 double* pPartialVector;
318
319 int AdditionalSize;
320 double* pAdditionalMatrix;
321 double* pAdditionalVector;
322
323 int* pPivotPos; // The Number of pivot rows selected at the iterations
324 int* pPivotIter; // The Iterations, at which the rows were pivots
325
326 MPI_Init(NULL, NULL);
327 MPI_Comm_size(MPI_COMM_WORLD, &NProc);
328 MPI_Comm_rank(MPI_COMM_WORLD, &ProcId);
329
330 if (ProcId == 0)
331     printf("Parallel Gauss algorithm for solving linear systems\n");
332
333 MPI_Barrier(MPI_COMM_WORLD);
334 // Memory allocation and definition of objects' elements
335 ProcessInitialization(pMatrix, pVector, pResult,
336     pPartialMatrix, pPartialVector, Size, PartialSize,
337     AdditionalSize, pAdditionalMatrix, pAdditionalVector);
338
339 //if (ProcId == 0) {
340 //    // The matrix and the vector output
341 //    printf("Initial Matrix \n");
342 //    PrintMatrix(pMatrix, Size, Size);
343 //    printf("Initial Vector \n");
344 //    PrintVector(pVector, Size);
345 //}
346
347 // Execution of Gauss algorithm
348 start = clock();
349 ParallelResultCalculation(pMatrix, pVector, pResult, Size,
350     pPartialMatrix, pPartialVector, PartialSize,
351     pAdditionalMatrix, pAdditionalVector, AdditionalSize, pPivotPos, pPivotIter);
352 finish = clock();
353 duration = (finish - start) / CLOCKS_PER_SEC;
354
355 MPI_Barrier(MPI_COMM_WORLD);
356 if (ProcId == 0) {
357     // Testing the result
358     TestResult(pMatrix, pVector, pResult, Size);

```

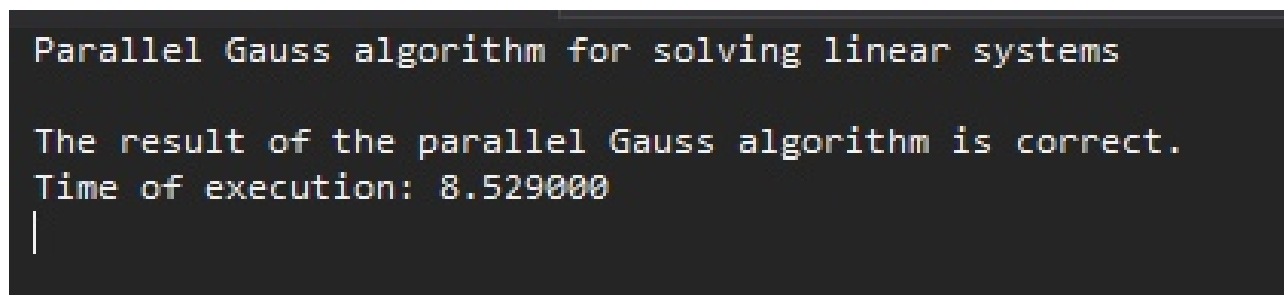


```

359
360     /// Printing the result vector
361     //printf("\nResult Vector: \n");
362     //PrintVector(pResult, Size);
363
364     // Printing the execution time of Gauss method
365     printf("\nTime of execution: %f\n", duration);
366 }
367
368 MPI_Finalize();
369
370 return 0;
371 }

```

Результат работы



```

Parallel Gauss algorithm for solving linear systems

The result of the parallel Gauss algorithm is correct.
Time of execution: 8.529000
|

```

Рисунок 1 – Work-13

Таблица сравнения

Номер теста	Порядок системы	Последовательный алгоритм	Параллельный алгоритм	
			Время	Ускорение
1	10	0.000000	0.001000	≈ 0
2	100	0.001000	0.001000	≈ 1
3	500	0.115000	0.050000	≈ 2.3
4	1000	0.934000	0.291000	≈ 3.21
5	1500	3.215000	1.095000	≈ 2.94
6	2000	7.592000	2.612000	≈ 2.91
7	2500	14.792000	4.880000	≈ 3.03
8	3000	25.848000	8.307000	≈ 3.11

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