

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 14.....	3
---	--------------	---

1 Work 14

Задание

Аналогично работе с OMP выполните следующее задание через MPI.

Решите систему линейных уравнений согласно варианту параллельным методом Гаусса.

$$\begin{cases} x_1 + x_2 + 4x_3 + 4x_4 + 9x_5 + 9 = 0 \\ 2x_1 + 2x_2 + 17x_3 + 17x_4 + 82x_5 + 146 = 0 \\ 2x_1 + 3x_3 - x_4 + 4x_5 + 10 = 0 \\ x_2 + 4x_3 + 12x_4 + 27x_5 + 26 = 0 \\ x_1 + 2x_2 + 2x_3 + 10x_4 - 37 = 0 \end{cases}$$

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.

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 = 5;
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     AdditionalSize = ((ProcId > 0) && (ProcId <= (Size % NProc)) && ((Size % NProc) > 0));
81
82     // 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
128     if (AdditionalSize) {

```

```

123         if (pPivotIter[PartialSize * NProc + ProcId - 1] == -1 && fabs(pAdditionalMatrix[Iter]) >=
            ↪ MaxValue) {
124             PivotRow = PartialSize * NProc + ProcId - 1;
125             MaxValue = fabs(pAdditionalMatrix[Iter]);
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,
                    ↪ MPI_COMM_WORLD);
180                 MPI_Send(&PivotValue, 1, MPI_DOUBLE, i, 1, MPI_COMM_WORLD);
181             }
182     }
183     else {
184         MPI_Recv(PivotRow, Size - Iter, MPI_DOUBLE, MPI_ANY_SOURCE, 0,
            ↪ MPI_COMM_WORLD, &st);
185         MPI_Recv(&PivotValue, 1, MPI_DOUBLE, MPI_ANY_SOURCE, 1,
            ↪ MPI_COMM_WORLD, &st);
186     }
187
188     for (int i = 0; i < PartialSize; ++i) {
189         if (pPivotIter[i + PartialSize * ProcId] == -1) {
190             PivotFactor = pPartialMatrix[i * Size + Iter] / PivotRow[0];
191             for (int j = Iter; j < Size; ++j)
192                 pPartialMatrix[i * Size + j] -= PivotFactor * PivotRow[j - Iter];
193             pPartialVector[i] -= PivotFactor * PivotValue;
194         }
195     }
196
197     if (AdditionalSize) {
198         if (pPivotIter[NProc * PartialSize + ProcId - 1] == -1) {
199             PivotFactor = pAdditionalMatrix[Iter] / PivotRow[0];
200             for (int j = Iter; j < Size; ++j)
201                 pAdditionalMatrix[j] -= PivotFactor * PivotRow[j - Iter];
202             pAdditionalVector[0] -= PivotFactor * PivotValue;
203         }
204     }
205 }
206
207 // Gaussian elimination
208 void ParallelGaussianElimination(double* pPartialMatrix, double* pPartialVector, int PartialSize,
209     double* pAdditionalMatrix, double* pAdditionalVector, int AdditionalSize, int Size, int* pPivotPos,
        ↪ int* pPivotIter) {
210     //MPI_Barrier(MPI_COMM_WORLD);
211     int PivotRow; // The number of the current pivot row

```

```

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

```

```

258 MPI_Gather(pPartialMatrix, Size * PartialSize, MPI_DOUBLE, pMatrix, Size * PartialSize,
    ↪ MPI_DOUBLE, 0, MPI_COMM_WORLD);
259 MPI_Gather(pPartialVector, PartialSize, MPI_DOUBLE, pVector, PartialSize, MPI_DOUBLE, 0,
    ↪ MPI_COMM_WORLD);
260
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,
            ↪ MPI_COMM_WORLD, &st);
264         MPI_Recv(pVector + PartialSize * NProc + i, 1, MPI_DOUBLE, i + 1, 0,
            ↪ MPI_COMM_WORLD, &st);
265     }
266 }
267 else if (AdditionalSize) {
268     MPI_Send(pAdditionalMatrix, Size, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
269     MPI_Send(pAdditionalVector, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
270 }
271
272 /*if (ProcId == 0)
273     PrintMatrix(pMatrix, Size, Size);*/
274
275 // Back substitution
276 if (ProcId == 0)
277     ParallelBackSubstitution(pMatrix, pVector, pResult, Size, pPivotPos);
278 }
279
280 // Function for testing the result
281 void TestResult(double* pMatrix, double* pVector, double* pResult, int Size) {
282     /* Buffer for storing the vector, that is a result of multiplication
283     of the linear system matrix by the vector of unknowns */
284     double* pRightPartVector;
285
286     // Flag, that shows wheather the right parts vectors are identical or not
287     int equal = 0;
288     double Accuracy = 1e-3; // Comparison accuracy
289     pRightPartVector = new double[Size];
290     for (int i = 0; i < Size; ++i) {
291         pRightPartVector[i] = 0;
292         for (int j = 0; j < Size; ++j)
293             pRightPartVector[i] += pMatrix[i * Size + j] * pResult[j];
294     }
295
296     for (int i = 0; i < Size; i++)
297         if (fabs(pRightPartVector[i] - pVector[i]) > Accuracy)
298             equal = 1;
299
300     if (equal == 1)
301         printf("\nThe result of the parallel Gauss algorithm is NOT correct. Check your code.");
302     else
303         printf("\nThe result of the parallel Gauss algorithm is correct.");

```

```

304
305     delete[] pRightPartVector;
306 }
307
308 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;

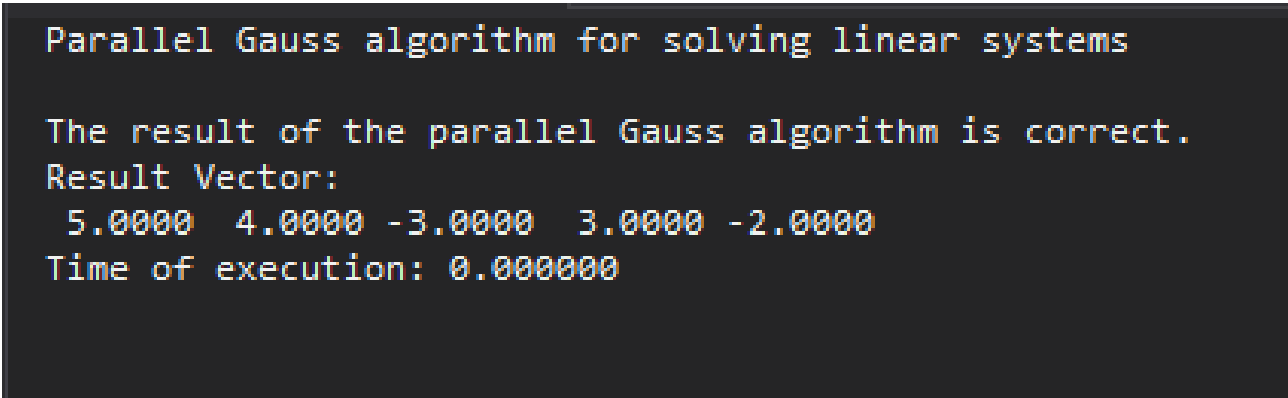
```

```

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.
Result Vector:
5.0000 4.0000 -3.0000 3.0000 -2.0000
Time of execution: 0.000000

```

Рисунок 1 – Work-14

Характеристики устройства

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

Ядер: 6

Оперативная память: 16 Гб