Лабораторная работа 3: Параллельные методы решения систем линейных уравнений

Цель лабораторной работы ... 1
Упражнение 1 – Определение задачи решения системы линейных уравнений ... 2
Упражнение 2 - Изучение последовательного алгоритма Гаусса решения систем линейных уравнений ... 2
Прямой ход алгоритма Гаусса ... 3
Обратный ход алгоритма Гаусса ... 4
Упражнение 3 – Реализация последовательного алгоритма Гаусса ... 5
Задание 1 – Открытие проекта SerialGauss ... 5
Задание 2 – Ввод размеров матрицы и вектора ... 6
Задание 3 – Ввод данных ... 7
Задание 4 – Завершение процесса вычислений ... 9
Задание 5 – Реализация прямого хода метода Гаусса ... 9
Задание 6 – Выполнение обратного хода алгоритма Гаусса ... 13
Задание 7 – Проведение вычислительных экспериментов ... 14
Упражнение 4 – Разработка параллельного алгоритма Гаусса ... 15
Определение подзадач ... 15
Выделение информационных зависимостей ... 15
Масштабирование и распределение подзадач по процессорам ... 15
Упражнение 5 - Реализация параллельного алгоритма Гаусса решения систем линейных уравнений ... 16
Задание 1 – Открытие проекта ParallelGauss ... 16
Задание 2 – Определение размеров объектов и ввод исходных данных ... 17
Задание 3 – Завершение процесса вычислений ... 19
Задание 4 – Распределение данных между процессами ... 20
Задание 5 – Параллельное выполнение прямого хода алгоритма Гаусса ... 22
Задание 7 – Параллельное выполнение обратного хода алгоритма Гаусса ... 26
Задание 8 – Сбор результатов ... 27
Задание 9 – Проверка правильности работы программы ... 28
Задание 10 – Проведение вычислительных экспериментов ... 29
Контрольные вопросы ... 30
Задания для самостоятельной работы ... 30
Приложение 1. Программный код последовательного алгоритма Гаусса решения линейных систем ... 31
Приложение 2. Программный код параллельного алгоритма Гаусса решения линейных систем ... 34

Системы линейных уравнений возникают при решении ряда прикладных задач, описываемых дифференциальными, интегральными или системами нелинейных (трансцендентных) уравнений. Они могут появляться также в задачах математического программирования, статистической обработки данных, аппроксимации функций, при дискретизации краевых дифференциальных задач методом конечных разностей или методом конечных элементов и др.

В данной лабораторной работе рассматривается один из прямых методов решения систем линейных уравнений – метод Гаусса и его параллельное обобщение.

Цель лабораторной работы

Целью данной лабораторной работы является разработка параллельной программы, которая выполняет решение системы линейных уравнений методом Гаусса. Выполнение лабораторной работы включает:

- Упражнение 1 – Определение задачи решения системы линейных уравнений
- Упражнение 2 – Изучение последовательного алгоритма Гаусса решения систем линейных уравнений
- Упражнение 3 – Реализация последовательного алгоритма Гаусса
- Упражнение 4 – Разработка параллельного алгоритма Гаусса
- Упражнение 5 - Реализация параллельного алгоритма Гаусса решения систем линейных уравнений

Примерное время выполнения лабораторной работы: **90 минут**.

При выполнении лабораторной работы предполагается знание раздела 4 "Параллельное программирование на основе MPI", раздела 6 "Принципы разработки параллельных методов" и раздела 9 "Параллельные методы решения систем линейных уравнений" учебных материалов курса. Кроме того, предполагается, что выполнена ознакомительная лабораторная работа "Параллельное программирование с использованием MPI", лабораторная работа 1 "Параллельные алгоритмы матрично-векторного умножения" и лабораторная работа 2 "Параллельные алгоритмы матричного умножения".

Упражнение 1 – Определение задачи решения системы линейных уравнений

Линейное уравнение с n неизвестными $x_0, x_1, \ldots, x_{n-1}$ может быть определено при помощи выражения

$$a_0 x_0 + a_1 x_1 + \ldots + a_{n-1} x_{n-1} = b \tag{3.1}$$

где величины $a_0, a_1, \ldots, a_{n-1}$ и b представляют собой постоянные значения.

Множество n линейных уравнений

$$\begin{aligned} a_{0,0}x_0 \quad &+a_{0,1}x_1 \quad &+\ldots+a_{0,n-1}x_{n-1} \quad &= b_0 \\ a_{1,0}x_0 \quad &+a_{1,1}x_1 \quad &+\ldots+a_{1,n-1}x_{n-1} \quad &= b_1 \\ \ldots & & & \\ a_{n-1,0}x_0 \quad &+a_{n-1,1}x_1 \quad &+\ldots+a_{n-1,n-1}x_{n-1} \quad &= b_{n-1} \end{aligned} \tag{3.2}$$

называется *системой линейных уравнений* или *линейной системой*. В более кратком (*матричном*) виде система может представлена как

$$Ax = b\ ,$$

где $A=(a_{i,j})$ есть вещественная матрица размера $n \times n$, а вектора b и x состоят из n 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.

Упражнение 2 - Изучение последовательного алгоритма Гаусса решения систем линейных уравнений

Метод Гаусса является широко известным *прямым* алгоритмом решения систем линейных уравнений, для которых матрицы коэффициентов являются *плотными*. Если система линейных уравнений является *невырожденной*, то метод Гаусса гарантирует нахождение решения с погрешностью, определяемой точностью машинных вычислений. Основная идея метода состоит в приведении матрицы A посредством эквивалентных преобразований (не меняющих решение системы (3.2)) к треугольному виду, после чего значения искомых неизвестных может быть получено непосредственно в явном виде.

В упражнении дается общая характеристика метода Гаусса, достаточная для начального понимания алгоритма и позволяющая рассмотреть возможные способы параллельных вычислений при решении систем линейных уравнений.

Метод Гаусса основывается на возможности выполнения преобразований линейных уравнений, которые не меняют при этом решение рассматриваемой системы (такие преобразования носят наименование *эквивалентных*). К числу таких преобразований относятся:

- Умножение любого из уравнений на ненулевую константу,
- Перестановка уравнений,
- Прибавление к уравнению любого другого уравнения 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

$$Ux = c,$$

где матрица коэффициентов получаемой системы имеет вид

$$U = \begin{pmatrix} u_{0,0} & u_{0,1} & \dots & u_{0,n-1} \\ 0 & u_{1,1} & \dots & u_{1,n-1} \\ & & \dots & \\ 0 & 0 & \dots & u_{n-1,n-1} \end{pmatrix}.$$

На *обратном ходе* метода Гаусса (второй этап алгоритма) осуществляется определение значений неизвестных. Из последнего уравнения преобразованной системы может быть вычислено значение переменной x_{n-1}, после этого из предпоследнего уравнения становится возможным определение переменной x_{n-2} и т.д.

Прямой ход алгоритма Гаусса

Прямой ход метода Гаусса состоит в последовательном исключении неизвестных в уравнениях решаемой системы линейных уравнений. На итерации i, $0 \le i < n-1$, метода производится исключение неизвестной i для всех уравнений с номерами k, больших i (т.е. $i < k \le n-1$,). Для этого из этих уравнений осуществляется вычитание строки i, умноженной на константу (a_{ki}/a_{ii}) с тем, чтобы результирующий коэффициент при неизвестной x_i в строках оказался нулевым – все необходимые вычисления могут быть определены при помощи соотношений:

$$\begin{aligned} a'_{kj} &= a_{kj} - (a_{ki} / a_{ii}) \cdot a_{ij}, \\ b'_k &= b_k - (a_{ki} / a_{ii}) \cdot b_i, \end{aligned} \quad i \le j \le n-1, i < k \le n-1, 0 \le i < n-1 \qquad (3.3)$$

(следует отметить, что аналогичные вычисления выполняются и над элементами вектора b).

Поясним выполнение прямого хода метода Гаусса на примере системы линейных уравнений вида:

$$\begin{aligned} x_0 &+3x_1 &+2x_2 &= 1 \\ 2x_0 &+7x_1 &+5x_2 &= 18. \\ x_0 &+4x_1 &+6x_2 &= 26 \end{aligned}$$

На первой итерации производится исключение неизвестной x_0 из второй и третьей строки. Для этого из этих строк нужно вычесть первую строку, умноженную соответственно на 2 и 1. После этих преобразований система уравнений принимает вид:

$$\begin{aligned} x_0 &+3x_1 &+2x_2 &= 1 \\ &x_1 &+x_2 &= 16. \\ &x_1 &+4x_2 &= 25 \end{aligned}$$

В результате остается выполнить последнюю итерацию и исключить неизвестную x_1 из третьего уравнения. Для этого необходимо вычесть вторую строку и в окончательной форме система имеет следующий вид:

$$\begin{aligned} x_0 &+3x_1 &+2x_2 &= 1 \\ &x_1 &+x_2 &= 16. \\ & &3x_2 &= 9 \end{aligned}$$

На рис. 3.1 представлена общая схема состояния данных на *i*-ой итерации прямого хода алгоритма Гаусса. Все коэффициенты при неизвестных, расположенные ниже главной диагонали и левее столбца *i*, уже являются нулевыми. На *i*-ой итерации прямого хода метода Гаусса осуществляется обнуление коэффициентов столбца *i*, расположенных ниже главной диагонали, путем вычитания строки *i*, умноженной на нужную ненулевую константу. После проведения *(n-1)* подобной итерации матрица, определяющая систему линейных уравнений, становится приведенной к верхнему треугольному виду.

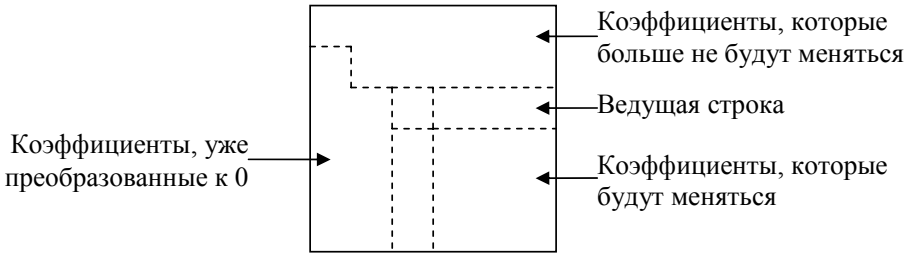

Рис. 3.1. Итерация прямого хода алгоритма Гаусса

При выполнении прямого хода метода Гаусса строка, которая используется для исключения неизвестных, носит наименование *ведущей*, а диагональный элемент ведущей строки называется *ведущим элементом*. Как можно заметить, выполнение вычислений является возможным только, если ведущий элемент имеет ненулевое значение. Более того, если ведущий элемент $a_{i,i}$ имеет малое значение, то деление и умножение строк на этот элемент может приводить к накоплению вычислительной погрешности и вычислительной неустойчивости алгоритма.

Возможный способ избежать подобной проблемы может состоять в следующем – при выполнении каждой очередной итерации прямого хода метода Гаусса следует определить коэффициент с максимальным значением по абсолютной величине в столбце, соответствующем исключаемой неизвестной, т.е.

$$y = \max_{i \le k \le n-1} |a_{ki}|,$$

и выбрать в качестве ведущей строку, в которой этот коэффициент располагается (данная схема выбора ведущего значения носит наименование *метода главных элементов*).

Вычислительная сложность прямого хода алгоритма Гаусса с выбором ведущей строки имеет порядок $O(n^3)$.

Обратный ход алгоритма Гаусса

После приведения матрицы коэффициентов к верхнему треугольному виду становится возможным определение значений неизвестных. Из последнего уравнения преобразованной системы может быть вычислено значение переменной x_{n-1}, после этого из предпоследнего уравнения становится возможным определение переменной x_{n-2} и т.д. В общем виде, выполняемые вычисления при обратном ходе метода Гаусса могут быть представлены при помощи соотношений:

$$x_{n-1} = b_{n-1} / a_{n-1,n-1},$$
$$x_i = (b_i - \sum_{j=i+1}^{n-1} a_{ij} x_j) / a_{ii}, \; i = n-2, n-3, ..., 0. \qquad (3.4)$$

Поясним, как и ранее, выполнение обратного хода метода Гаусса на примере рассмотренной в предыдущем подразделе системы линейных уравнений

$$\begin{aligned} x_0 + 3x_1 + 2x_2 &= 1 \\ x_1 + x_2 &= 16. \\ 3x_2 &= 9 \end{aligned}$$

Из последнего уравнения системы можно определить, что неизвестная x_2 имеет значение 3. В результате становится возможным разрешение второго уравнения и определение значение неизвестной x_1=13, т.е.

$$\begin{aligned} x_0 + 3x_1 + 2x_2 &= 1 \\ x_1 &= 13. \\ x_2 &= 3 \end{aligned}$$

На последней итерации обратного хода метода Гаусса определяется значение неизвестной x_0, равное -44.

С учетом последующего параллельного выполнения можно отметить, учет получаемых значений неизвестных может выполняться сразу во всех уравнениях системы (и эти действия могут выполняться в уравнениях одновременно и независимо друг от друга). Так, в рассматриваемом примере после определения значения неизвестной x_2 система уравнений может быть приведена к виду

$$
\begin{aligned}
x_0 \quad +3x_1 \quad &= \quad -5 \\
x_1 \quad &= \quad 13 \\
x_2 \quad &= \quad 3
\end{aligned}
$$

Вычислительная сложность обратного хода алгоритма Гаусса составляет $O(n^2)$.

Упражнение 3 – Реализация последовательного алгоритма Гаусса

При выполнении этого упражнения необходимо реализовать последовательный алгоритм Гаусса решения систем линейных уравнений. Начальный вариант будущей программы представлен в проекте *SerailGauss*, который содержит часть исходного кода и в котором заданы необходимые параметры проекта. В ходе выполнения упражнения необходимо дополнить имеющийся вариант программы операциями ввода размера матриц, задание исходных данных, реализации алгоритма Гаусса и вывода результатов.

Задание 1 – Открытие проекта SerialGauss

Откройте проект **SerialGauss**, последовательно выполняя следующие шаги:

- Запустите приложение **Microsoft Visual Studio 2005**, если оно еще не запущено,
- В меню **File** выполните команду **Open→Project/Solution,**
- В диалоговом окне **Open Project** выберите папку **c:\MsLabs\SerialGauss,**
- Дважды щелкните на файле **SerialGauss.sln** или выбрав файл выполните команду **Open.**

После открытия проекта в окне Solution Explorer (Ctrl+Alt+L) дважды щелкните на файле исходного кода **SerialGauss.cpp**, как это показано на рис. 3.2. После этих действий код, который предстоит в дальнейшем расширить, будет открыт в рабочей области Visual Studio.

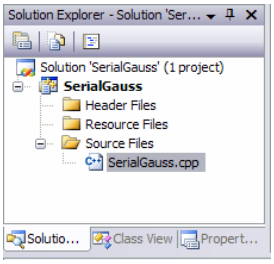

Рис. 3.2. Открытие файла **SerialGauss.cpp**

В файле **SerialGauss.cpp** подключаются необходимые библиотеки, а также содержится начальный вариант основной функции программы – функции *main*. Эта заготовка содержит объявление переменных и вывод на печать начального сообщения программы.

Рассмотрим переменные, которые используются в основной функции (*main*) нашего приложения. Первые две из них (*pMatrix* и *pVector*) – это, соответственно, матрица системы линейных уравнений и вектор правых частей системы. Третья переменная *pResult* – вектор, который должен быть получен в результате решения системы линейных уравнений. Переменная *Size* определяет размер матрицы и векторов.

```
  double* pMatrix;  // The matrix of linear system
  double* pVector;  // The right parts of the linear system
  double* pResult;  // The result vector
  int Size;         // Sizes of the initial matrix and the vector
```

Как и в предыдущих лабораторных работах, для хранения матрицы используется одномерный массив, в котором матрица хранятся построчно. Таким образом, элемент, расположенный на пересечении *i*-ой строки и *j*-ого столбца матрицы, в одномерном массиве имеет индекс *i*Size+j*.

Программный код, который следует за объявлением переменных, это вывод начального сообщения и ожидание нажатия любой клавиши перед завершением выполнения приложения:

```
  printf("Serial Gauss algorithm for solving linear systems\n");
  getch();
```

Теперь можно осуществить первый запуск приложения. Выполните команду **Rebuild Solution** в меню **Build** – эта команда позволяет скомпилировать приложение. Если приложение скомпилировано успешно (в нижней части окна Visual Studio появилось сообщение "`Rebuild All: 1 succeeded, 0`

failed, 0 skipped"), нажмите клавишу **F5** или выполните команду **Start Debugging** пункта меню **Debug**.

Сразу после запуска кода, в командной консоли появится сообщение: "Serial Gauss algorithm for solving linear systems ". Для того, чтобы завершить выполнение программы, нажмите любую клавишу.

Задание 2 – Ввод размеров матрицы и вектора

Для задания исходных данных последовательного алгоритма Гаусса решения системы линейных уравнений реализуем функцию *ProcessInitialization*. Эта функция предназначена для определения размера матрицы и векторов, выделения памяти для исходных матрицы *pMatrix* и вектора *pVector*, и вектора-результата *pResult*, а также для задания значений элементов исходных объектов. Значит, функция должна иметь следующий интерфейс:

```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pMatrix, double* &pVector,
  double* &pResult, int &Size);
```

На первом этапе необходимо определить размер матриц (задать значение переменной *Size*). В тело функции *ProcessInitialization* добавьте выделенный фрагмент кода:

```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pMatrix, double* &pVector,
  double* &pResult, int &Size) {
  // Setting the size of the matrix and the vector
  printf("\nEnter the size of the matrix and the vector: ");
  scanf("%d", &Size);
  printf("\nChosen size = %d", Size);
}
```

Пользователю предоставляется возможность ввести размер матриц, который затем считывается из стандартного потока ввода *stdin* и сохраняется в целочисленной переменной *Size*. Далее печатается значение переменной *Size* (рис. 3.3).

После строки, выводящей на экран приветствие, добавьте вызов функции инициализации процесса вычислений *ProcessInitialization* в тело основной функции последовательного приложения:

```
void main() {
  double* pMatrix;  // The matrix of the linear system
  double* pVector;  // The right parts of the linear system
  double* pResult;  // The result vector
  int Size;         // The sizes of the initial matrix and the vector
  time_t start, finish;
  double duration;

  printf ("Serial Gauss algorithm for solving linear systems \n");
  ProcessInitialization(pMatrix, pVector, pResult, Size);
  getch();
}
```

Скомпилируйте и запустите приложение. Убедитесь в том, что значение переменной *Size* задается корректно.

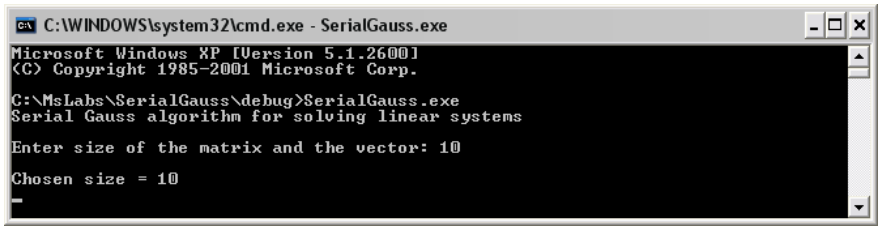

Рис. 3.3. Задание размера объектов

Как и при выполнении предыдущих лабораторных работ, выполним контроль правильности ввода. Организуем проверку размера и, в случае ошибки (заданный размер является нулевым или отрицательным), продолжим запрашивать размер матриц до тех пор, пока не будет введено положительное число. Для реализации такого поведения поместим фрагмент кода, который производит ввод размера матриц, в цикл с постусловием:

```
  // Setting the size of the matrix and the vector
  do {
    printf("\nEnter the size of the matrix and the vector: ");
    scanf("%d", &Size);
    printf("\nChosen size = %d\n", Size);

    if (Size <= 0)
    printf("\nSize of objects must be greater than 0!\n");
  } while (Size <= 0);
```

Снова скомпилируйте и запустите приложение. Попытайтесь ввести неположительное число в качестве размера объектов. Убедитесь в том, что ошибочные ситуации обрабатываются корректно.

Задание 3 – Ввод данных

Функция инициализации процесса вычислений должна осуществлять также выделение памяти для хранения объектов (добавьте выделенный код в тело функции *ProcessInitialization*):

```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pAMatrix, double* &pBMatrix,
  double* &pCMatrix, int &Size) {
  // Setting the size of the matrix and the vector
  do {
    <…>
  }
  while (Size <= 0);

  // Memory allocation
  pMatrix = new double [Size*Size];
  pVector = new double [Size];
  pResult = new double [Size];
}
```

Далее необходимо задать значения элементов матрицы системы линейных уравнений *pMatrix* и вектора правых частей *pVector*. Заметим, что матрица системы линейных уравнений не может быть задана произвольным образом. Решение системы линейных уравнений существует только в случае, когда матрица системы линейных уравнений *невырожденная* (то есть для нее существует обратная). Проводить проверку матрицы, сгенерированной случайным образом, на невырожденность нецелесообразно (это слишком "дорогостоящая" операция). Поэтому реализуем функции генерации исходных данных таким образом, чтобы матрица изначально являлась невырожденной. Будем генерировать нижнюю треугольную матрицу, то есть матрицу, у которой все ненулевые элементы, расположены либо на главной диагонали, либо ниже ее. Для задания значений элементов матрицы *pMatrix* и вектора *pVector* реализуем функцию *DummyDataInitialization*. Интерфейс и реализация этой функции представлены ниже:

```
// Function for simple initialization of the matrix and the vector elements
void DummyDataInitialization (double* pMatrix, double* pVector, int Size) {
  int i, j;  // Loop variables
  for (i=0; i<Size; i++) {
    pVector[i] = i+1;
    for (j=0; j<Size; j++) {
      if (j <= i)
        pMatrix[i*Size+j] = 1;
      else
        pMatrix[i*Size+j] = 0;
     }
  }
}
```

Как видно из представленного фрагмента кода, данная функция осуществляет задание элементов матрицы и вектора простым образом: значения всех элементов матрицы *pMatrix* , расположенные выше главной диагонали, равны 0, остальные элементы равны 1. Вектор *pVector* состоит из последовательных целых положительных чисел от 1 до *Size*. То есть в случае, когда пользователь выбрал размер объектов, например, равный 4, будут определены следующие матрица и вектор:

$$pMatrix = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix},\ pVector = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}.$$

Вызов функции *DummyDataInitialization* необходимо выполнить после выделения памяти внутри функции *ProcessInitialization*:

```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pAMatrix, double* &pBMatrix,
  double* &pCMatrix, int &Size) {
    <…>
  // Memory allocation
  <…>

  // Initialization of the matrix and the vector elements
  DummyDataInitialization(pMatrix, pVector, Size);
}
```

Для контроля ввода данных воспользуемся функциями форматированного вывода объектов *PrintMatrix* и *PrintVector*, которые были разработаны при выполнении лабораторной работы 1 и текст которых уже имеется в проекте (подробнее о функциях *PrintMatrix* и *PrintVector* см. задание 3 упражнение 2 лабораторной работы 1). Добавим вызов этих функций для печати объектов *pMatrix* и *pVector* в основную функцию приложения:

```
  // Memory allocation and data initialization
  ProcessInitialization(pAMatrix, pBMatrix, pCMatrix, Size);

  // Matrix and vector output
  printf ("Initial Matrix \n");
  PrintMatrix(pMatrix, Size, Size);
  printf("Initial Vector \n");
  PrintVector(pVector, Size);
```

Скомпилируйте и запустите приложение. Убедитесь в том, что ввод данных происходит по описанным правилам (рис. 3.4). Выполните несколько запусков приложения, задавайте различные размеры матриц.

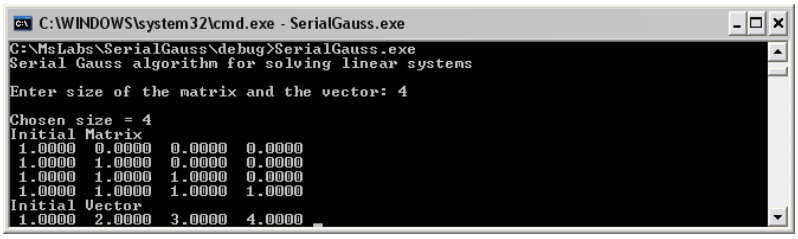

Рис. 3.4. Результат работы программы при завершении задания 3

Следует отметить, что если матрица системы линейных уравнений и вектор правых частей задаются по описанным выше правилам, то такая система имеет простое решение, все элементы искомого вектора *pResult* должны быть равны 1.

Реализуем еще одну функцию генерации исходных данных, в которой по-прежнему будет задаваться нижнеяя треугольная матрица, но элементы этой матрицы и вектора правых частей будут определятся с помощью датчика случайных чисел (датчик случайных чисел инициализируется текущим значением времени):

```
// Function for random initialization of the matrix and the vector elements
void RandomDataInitialization(double* pMatrix, double* pVector, int Size) {
  int i, j;  // Loop variables
  srand(unsigned(clock()));
  for (i=0; i<Size; i++) {
    pVector[i] = rand()/double(1000);
    for (j=0; j<Size; j++) {
      if (j <= i)
        pMatrix[i*Size+j] = rand()/double(1000);
```

```
      else
        pMatrix[i*Size+j] = 0;
    }
  }
}
```

Замените вызов функции простой генерации исходных данных *DummyDataInitialization* вызовом функции случайной генерации *RandomDataInitialization*. Скомпилируйте и запустите приложение. Убедитесь в том, что данные задаются согласно описанным правилам.

Задание 4 – Завершение процесса вычислений

Перед выполнением матрично-векторного умножения сначала разработаем функцию для корректного завершения процесса вычислений. Для этого необходимо освободить память, выделенную динамически в процессе выполнения программы. Реализуем соответствующую функцию *ProcessTermination*. Память выделялась для хранения исходных матрицы *pMatrix* и вектора *pVector*, а также для хранения вектора - результата решения системы линейных уравнений *pResult*. Следовательно, эти объекты необходимо передать в функцию *ProcessTermination* в качестве аргументов:

```
// Function for computational process termination
void ProcessTermination (double* pMatrix,double* pVector,double* pResult) {
  delete [] pMatrix;
  delete [] pVector;
  delete [] pResult;
}
```

Вызов функции *ProcessTermination* необходимо выполнить непосредственно перед завершением программы:

```
  // Memory allocation and definition of the objects elements
  ProcessInitialization(pMatrix, pVector, pResult, Size);

  // Matrix and vector output
  printf ("Initial Matrix \n");
  PrintMatrix(pMatrix, Size, Size);
  printf("Initial Vector \n");
  PrintVector(pVector, Size);

  // Process termination
  ProcessTermination(pMatrix, pVector, pResult);
```

Скомпилируйте и запустите приложение. Убедитесь в том, что оно выполняется корректно.

Задание 5 – Реализация прямого хода метода Гаусса

Выполним теперь разработку основной вычислительной части программы. Для решения системы линейных уравнений при помощи последовательного алгоритма Гаусса реализуем функцию *SerialResultCalculation*, которая принимает на вход исходные матрицу *pMatrix* и вектор *pVector*, размер этих объектов *Size*, а также указатель на вектор-результат *pResult*.

В соответствии с алгоритмом, изложенным в упражнении 2, выполнение алгоритма Гаусса состоит из двух этапов: прямого хода метода Гаусса и обратного хода метода Гаусса. На этапе выполнения прямого хода метода Гаусса система линейных уравнений путем эквивалентных преобразований приводится к верхнему треугольному виду. Для выполнения этого этапа реализуем функцию *SerialGaussianElimination*. При выполнении обратного хода алгоритма Гаусса определяются значения искомого вектора путем приведения матрицы к диагональному виду. Для выполнения этого этапа реализуем функцию *SerialBackSubstitution*. Таким образом, код функции *SerialResultCalculation* должен выглядеть следующим образом:

```
// Function for the execution of Gauss algorithm
void SerialResultCalculation(double* pMatrix, double* pVector,
  double* pResult, int Size) {
  // Gaussian elimination
  SerialGaussianElimination (pMatrix, pVector, Size);
  // Back substitution
  SerialBackSubstitution (pMatrix, pVector, pResult, Size);
}
```

В данном задании будет выполнена реализация прямого хода метода Гаусса. Обратный ход метода Гаусса будет выполнен в следующем задании лабораторной работы.

Прямой ход метода Гаусса путем эквивалентных преобразований приводит матрицу системы линейных уравнений к верхнему треугольному виду. На каждой итерации выполняемых преобразований для выбора ведущей строки применяют *метод главных элементов* (см. упражнение 2), в соответствии с которым в качестве ведущей выбирается строка, содержащая максимальный по абсолютному значению элемент очередного столбца матрицы.

Для запоминания порядка выбора ведущих строк введем массив *pSerialPivotPos*, в *i*-ом элементе которого будем хранить номер строки, которая была выбрана в качестве ведущей при выполнении *i*-ой итерации прямого хода алгоритма Гаусса. Кроме того, определим еще один дополнительный массив – *pSerialPivotIter*, в каждом элементе которого *pSerialPivotIter[i]* будем хранить номер итерации, на которой строка с номером *i* выбиралась в качестве ведущей. Изначально массив *pSerialPivotIter* заполним элементами, равными -1 (т.е. значение -1 в элементе массива *pSerialPivotIter[i]* означает, что строка с номером *i* еще не выбиралась в качестве ведущей).

Объявим соответствующие массивы как глобальные переменные, выделим память для этих массивов перед началом выполнения функции *GaussianElimination*, освободим выделенную память после завершения выполнения обратного хода метода Гаусса (функции *BackSubstitution*):

```
int* pSerialPivotPos;  // The number of pivot rows selected at the
                       // iterations
int* pSerialPivotIter; // The iterations, at which the rows were pivots

// Function for the execution of Gauss algorithm
void SerialResultCalculation(double* pMatrix, double* pVector,
  double* pResult, int Size) {

  // Memory allocation
  pSerialPivotPos  = new int [Size];
  pSerialPivotIter = new int [Size];
  for (int i=0; i<Size; i++) {
    pSerialPivotIter[i] = -1;
  }
  // Gaussian elimination
  SerialGaussianElimination (pMatrix, pVector, Size);
  // Back substitution
  SerialBackSubstitution (pMatrix, pVector, pResult, Size);

  // Memory deallocation
  delete [] pSerialPivotPos;
  delete [] pSerialPivotIter;
}
```

Согласно вычислительной схеме прямого хода алгоритма Гаусса, на каждой итерации необходимо определить ведущую строку матрицы, то есть строку, которая содержит максимальный по абсолютной величине элемент в столбце с номером, равным номеру текущей итерации, среди тех строк, которые ранее не были выбраны в качестве ведущих. Номер ведущей строки запоминается в переменной *Pivot* и записывается в соответствующий элемент массива *pSerialPivotPos*. Кроме того, значение элемента массива *pSerialPivotIter*, соответствующего выбранной строке, устанавливается равным номеру текущей итерации.

Реализуем функцию *FindPivotRow* для выбора ведущей строки. В качестве аргументов этой функции необходимо передать матрицу системы линейных уравнений *pMatrix*, размер матрицы *Size* и номер текущей итерации *Iter*. Эта функция должна просмотреть все строки, которые ранее не были выбраны в качестве ведущих, выбрать среди них ту, которая содержит максимальный элемент в позиции *Iter*, и вернуть номер выбранной строки:

```
// Finding the pivot row
int FindPivotRow(double* pMatrix, int Size, int Iter) {
  int PivotRow = -1; // The index of the pivot row
  double MaxValue =  0; // The value of the pivot element
  int i;             // Loop variable

  // Choose the row, that stores the maximum element
  for (i=0; i<Size; i++) {
    if ((pSerialPivotIter[i] == -1) &&
```

```
      (fabs(pMatrix[i*Size+Iter]) > MaxValue)) {
      PivotRow = i;
      MaxValue = fabs(pMatrix[i*Size+Iter]);
    }
  }
  return PivotRow;
}
```

Добавим вызов функции *FindPivotRow* в тело функции, выполняющей прямой ход метода Гаусса, запомним найденное значение в соответствующем элементе массива *pSerialPivotPos* и напечатаем номера выбираемых ведущих строк для проверки правильности вычислений:

```
// Gaussian elimination
void SerialGaussianElimination(double* pMatrix,double* pVector,int Size) {
  int Iter;       // The Number of the iteration of the gaussian
                  // elimination
  int PivotRow;   // The Number of the current pivot row
  for (Iter=0; Iter<Size; Iter++) {
    // Finding the pivot row
    PivotRow = FindPivotRow(pMatrix, Size, Iter);
    pSerialPivotPos[Iter] = PivotRow;
    pSerialPivotIter[PivotRow] = Iter;
  }
  printf ("Indices of the pivot rows: \n");
  for (int i=0; i<Size; i++)
    printf("%d ", pSerialPivotPos[i]);
}
```

В функции *SerialResultCalculation* закомментируqйте вызов функции, выполняющей обратный ход метода Гаусса. Добавьте вызов функции *SerialResultCalculation* в главную функцию приложения:

```
void main() {
  <…>
  // Memory allocation and data initialization
  ProcessInitialization(pMatrix, pVector, pResult, Size);

  // The matrix and the vector output
  <…>
  // Execution of Gauss algorithm
  SerialResultCalculation(pMatrix, pVector, pResult, Size);

  // Computational process termination
  ProcessTermination(pMatrix, pVector, pResult);
  getch();
}
```

Скомпилируйте и запустите приложение. Убедитесь в том, что ведущие строки выбираются правильно. При использовании функции *DummyDataInitialization* номера ведущих строк должны совпадать с номерами итераций, на которых эти строки выбирались. При использовании функции *RandomDataInitialization* общий вид результатов печати показан на рис. 3.5 (для наглядности значения ведущих элементов выделены красным цветом).

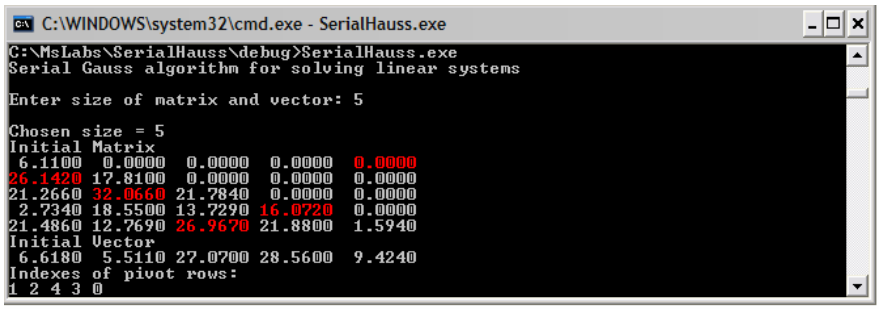

Рис. 3.5. Выбор ведущих строк: сначала выбираем строку, которая содержит максимальный элемент в первом столбце, затем среди всех строк, кроме второй, выбираем строку, содержащую максимальный элемент во втором столбце, и т.д.

Далее после выбора ведущих строк выполняется вычитание этих строк, умноженных на соответствующие множители, из всех строк, которые еще не выбирались в качестве ведущих, и таким образом зануляются элементы соответствующих столбцов. Для выполнения вычитания реализуем функцию *SerialEliminateColumns*, которая принимает на вход матрицу системы линейных уравнений *pMatrix*, вектор правых частей *pVector*, номер текущей ведущей строки *Pivot*, номер текущей итерации *Iter* и размер *Size*. Для всех строк матрицы *pMatrix* функция *EliminateRows* выполняет следующие действия: проверяет при помощи значений, записанных в массиве *pSerialPivotIter*, не была ли данная строка выбрана в качестве ведущей на одной из предшествующих итераций, и, если результат проверки отрицательный, то над этой строкой выполняются преобразования, согласно формуле (3.3):

```
// Column elimination
void SerialColumnElimination (double* pMatrix, double* pVector, int Pivot,
  int Iter, int Size) {
  double PivotValue, PivotFactor;
  PivotValue = pMatrix[Pivot*Size+Iter];
  for (int i=0; i<Size; i++) {
    if (pSerialPivotIter[i] == -1) {
      PivotFactor = pMatrix[i*Size+Iter] / PivotValue;
      for (int j=Iter; j<Size; j++) {
        pMatrix[i*Size + j] -= PivotFactor * pMatrix[Pivot*Size+j];
      }
      pVector[i] -= PivotFactor * pVector[Pivot];
    }
  }
}
```

Добавьте вызов функции *SerialColumnElimination* в код функции, выполняющей прямой ход алгоритма Гаусса. Вместо печати массива *pPivotPos* распечатайте матрицу системы линейных уравнений *pMatrix*. Она должна быть приведена к верхнему треугольному виду с точностью до перестановки строк (то есть должна существовать возможность перестановки строк матрицы так, чтобы получилась верхняя треугольная матрица) (см. рис. 3.6).

```
void SerialGaussianElimination(double* pMatrix,double* pVector,int Size) {
  int Iter;       // The Number of the iteration of the gaussian
                  // elimination stage
  int PivotRow;   // The Number of the current pivot row
  for (Iter=0; Iter<Size; Iter++) {
    // Finding the pivot row
    PivotRow = FindPivotRow(pMatrix, Size,Iter);
    pSerialPivotPos[Iter] = PivotRow;
    pSerialPivotIter[PivotRow] = Iter;
    SerialColumnElimination(pMatrix, pVector, PivotRow, Iter, Size);
  }
  // printf ("Indices of the pivot rows: \n");
  // for (int i=0; i<Size; i++)
  //  printf("%d ", pSerialPivotPos[i]);
  printf ("The matrix of the linear system after the elimination: \n");
  PrintMatrix(pMatrix, Size, Size);
}
```

Скомпилируйте и запустите приложение. Убедитесь в том, что прямой ход метода Гаусса выполняется правильно.

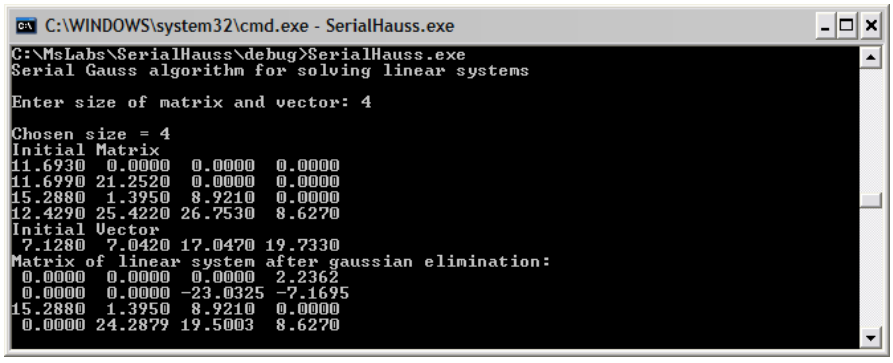

Рис. 3.6. Результат выполнения прямого хода алгоритма Гаусса

Задание 6 – Выполнение обратного хода алгоритма Гаусса

Для выполнения обратного хода алгоритма Гаусса реализуем функцию *SerialBackSubstitution*. На вход этой функции передадим матрицу системы линейных уравнений *pMatrix*, вектор правых частей *pVector*, вектор результата *pResult* и размер *Size*:

```
// Back substution
void SerialBackSubstitution (double* pMatrix, double* pVector,
  double* pResult, int Size);
```

Алгоритм обратного хода алгоритма Гаусса подробно описан в упражнении 2. Выполнение обратного хода начинается с той строки матрицы, которая была выбрана ведущей на последней итерации прямого хода. Номер этой строки можно узнать, обратившись к последнему элементу массива *pSerialPivotPos* (аналогично номер строки, которая была выбрана ведущей на предпоследней итерации прямого хода, хранится в предпоследнем элементе массива *pSerialPivotPos* и так далее). Используя эту строку можно вычислить один из элементов результирующего вектора, а затем, используя этот элемент, упростить остальные строки матрицы:

```
// Back substution
void SerialBackSubstitution (double* pMatrix, double* pVector,
  double* pResult, int Size) {
  int RowIndex, Row;
  for (int i=Size-1; i>=0; i--) {
    RowIndex = pSerialPivotPos[i];
    pResult[i] = pVector[RowIndex]/pMatrix[Size*RowIndex+i];
    for (int j=0; j<i; j++) {
      Row = pSerialPivotPos[j];
      pVector[j] -= pMatrix[Row*Size+i]*pResult[i];
      pMatrix[Row*Size+i] = 0;
    }
  }
}
```

Удалите печать матрицы после выполнения прямого хода метода Гаусса. Для генерации исходных данных снова используйте метод *DummyDataInitialization*. Раскомментируйте вызов функции выполнения обратного хода метода Гаусса. Вызовите печать результирующего вектора после выполнения последовательного алгоритма Гаусса в основной функции приложения:

```
void main() {
 <…>

  // The Execution of Gauss algorithm
  SerialResultCalculation(pMatrix, pVector, pResult, Size);

  // Printing the result vector
  printf ("\n Result Vector: \n");
  PrintVector(pResult, Size);

  // Computational process termination
  ProcessTermination(pMatrix, pVector, pResult);
  getch();
}
```

Скомпилируйте и запустите приложение. Если алгоритм реализован верно, все элементы результирующего вектора должны быть равны 1 (рис. 3.7).

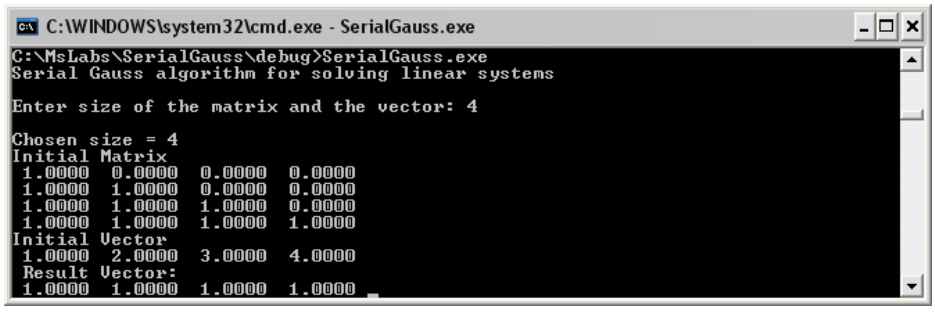

Рис. 3.7. Результат выполнения последовательного алгоритма Гаусса

Задание 7 – Проведение вычислительных экспериментов

Для последующего тестирования ускорения работы параллельного алгоритма необходимо провести эксперименты по вычислению времени выполнения последовательного алгоритма. Анализ времени выполнения алгоритма разумно проводить для достаточно больших размеров системы линейных уравнений. Задавать элементы больших матриц и векторов будем при помощи датчика случайных чисел (функция *RandomDataInitialization*).

Для определения времени добавьте в получившуюся программу вызовы функций, позволяющие узнать время выполнения вычислений. Мы, как и ранее, будем пользоваться функцией:

```
time_t clock(void);
```

Добавим в программный код вычисление и вывод времени выполнения метода Гаусса, для этого поставим замеры времени до и после вызова функции *SerialResultCalculation*:

```
  // The execution of Gauss algorithm
  start = clock();
  SerialResultCalculation(pMatrix, pVector, pResult, Size);
  finish = clock();
  duration = (finish-start)/double(CLOCKS_PER_SEC);

  // Printing the result vector
  printf ("\n Result Vector: \n");
  PrintVector(pResult, Size);

  // Printing the execution time of Gauss method
  printf("\n Time of execution: %f\n", duration);
```

Скомпилируйте и запустите приложение. Для проведения вычислительных экспериментов с большими объектами отключите печать исходных матрицы и вектора и печать результирующего вектора (закомментируйте соответствующие строки кода). Проведите вычислительные эксперименты, результаты занесите в таблицу:

Таблица 3.1. Время выполнения последовательного алгоритма Гаусса

Номер теста	Размер матрицы	Время работы (сек)
1	10	
2	100	
3	500	
4	1000	
5	1500	
6	2000	
7	2500	
8	3000	

В результате анализа выполняемых в методе Гаусса вычислений можно показать, что теоретическое время выполнения последовательного алгоритма Гаусса может быть вычислено в соответствии с выражением (см. раздел 9 "Параллельные методы решения систем линейных уравнений"):

$$T_1 = \left(2Size^3 / 3 + Size^2\right) \cdot \tau \qquad (3.5)$$

где τ есть время выполнения одной базовой вычислительной операции.

Заполним таблицу сравнения реального времени выполнения со временем, которое может быть получено по формуле (3.5). Для вычисления времени выполнения одной операции τ, как и при выполнении предыдущих лабораторных работ, выберем один из экспериментов в качестве образца, время выполнения этого эксперимента поделим на число выполненных операций (число операций может быть вычислено по формуле (3.5)). Таким образом, вычислим время выполнения одной операции. Далее, используя это значение, вычислим теоретическое время выполнения для всех оставшихся экспериментов. Напомним, что время выполнения одной операции, вообще говоря, зависит от размера объектов, поэтому при выборе эксперимента для образца следует ориентироваться на некоторый средний случай.

Вычислите теоретическое время выполнения алгоритма Гаусса. Результаты занесите в таблицу:

Таблица 3.2. Сравнение реального времени выполнения последовательного алгоритма Гаусса со временем, вычисленным теоретически

Время выполнения одной операции τ (сек):			
Номер теста	Размер матрицы	Время работы (сек)	Теоретическое время (сек)
1	10		
2	100		
3	500		
4	1000		
5	1500		
6	2000		
7	2500		
8	3000		

Упражнение 4 – Разработка параллельного алгоритма Гаусса

Определение подзадач

При внимательном рассмотрении метода Гаусса можно заметить, что все вычисления сводятся к однотипным вычислительным операциям над строками матрицы коэффициентов системы линейных уравнений. Как результат, в основу параллельной реализации алгоритма Гаусса может быть положен принцип распараллеливания по данным. В качестве *базовой подзадачи* можно принять тогда все вычисления, связанные с обработкой одной строки матрицы A и соответствующего элемента вектора b.

Выделение информационных зависимостей

Рассмотрим общую схему параллельных вычислений и возникающие при этом информационные зависимости между базовыми подзадачами.

Для выполнения **прямого хода** метода Гаусса необходимо осуществить $(n-1)$ итерацию по исключению неизвестных для преобразования матрицы коэффициентов A к верхнему треугольному виду.

Выполнение итерации $i,\ 0 \le i < n-1,$ прямого хода метода Гаусса включает ряд последовательных действий. Прежде всего, в самом начале итерации необходимо выбрать ведущую строку, которая при использовании метода главных элементов определяется поиском строки с наибольшим по абсолютной величине значением среди элементов столбца i, соответствующего исключаемой переменной x_i. Поскольку строки матрицы A распределены по подзадачам, для поиска максимального значения подзадачи с номерами $k,\ k > i,$ должны обменяться своими элементами при исключаемой переменной x_i. После сбора всех необходимых данных в каждой подзадаче может быть определено, какая из подзадач содержит ведущую строку и какое значение является ведущим элементом.

Далее для продолжения вычислений ведущая подзадача должна разослать свою строку матрицы A и соответствующий элемент вектора b всем остальным подзадачам с номерами $k,\ k > i$. Получив ведущую строку, подзадачи выполняют вычитание строк, обеспечивая тем самым исключение соответствующей неизвестной x_i.

При выполнении **обратного хода** метода Гаусса подзадачи выполняют необходимые вычисления для нахождения значения неизвестных. Как только какая-либо подзадача $i,\ 0 \le i < n-1,$ определяет значение своей переменной x_i, это значение должно быть разослано всем подзадачам с номерами $k,\ k < i$. Далее подзадачи подставляют полученное значение новой неизвестной и выполняют корректировку значений для элементов вектора b.

Масштабирование и распределение подзадач по процессорам

Выделенные базовые подзадачи характеризуются одинаковой вычислительной трудоемкостью и сбалансированным объемом передаваемых данных. В случае, когда размер матрицы, описывающей систему линейных уравнений, оказывается большим, чем число доступных процессоров (т.е., $p<n$), базовые подзадачи можно укрупнить, объединив в рамках одной подзадачи несколько строк матрицы. Воспользуемся уже знакомой ленточной схемой разделения данных: каждому процессу выделяется непрерывная последовательность строк матрицы линейных уравнений.

Распределение подзадач между процессорами должно учитывать характер выполняемых в методе Гаусса коммуникационных операций. Основным видом информационного взаимодействия подзадач является операция передачи данных от одного процессора всем процессорам вычислительной системы. Как результат, для эффективной реализации требуемых информационных взаимодействий между базовыми подзадачами топология сети передачи данных должны иметь структуру гиперкуба или полного графа.

Упражнение 5 - Реализация параллельного алгоритма Гаусса решения систем линейных уравнений

При выполнении этого упражнения Вам будет предложено разработать параллельный алгоритм Гаусса для решения систем линейных уравнений. При работе с этим упражнением Вы

- Получите опыт разработки сложных параллельных программ,
- Познакомитесь с коллективными операциями передачи данных в MPI.

Задание 1 – Открытие проекта ParallelGauss

Откройте проект **ParallelGauss**, последовательно выполняя следующие шаги:

- Запустите приложение **Microsoft Visual Studio 2005**, если оно еще не запущено,
- В меню File выполните команду **Open→Project/Solution**,
- В диалоговом окне **Open Project** выберите папку **c:\MsLabs\ParallelGauss**,
- Дважды щелкните на файле **ParallelGauss.sln** или подсветите его выполните команду **Open**.

После того, как Вы открыли проект, в окне Solution Explorer (Ctrl+Alt+L) дважды щелкните на файле исходного кода **ParallelGauss.cpp**, как это показано на рисунке 3.8. После этих действий код, который вам предстоит модифицировать, будет открыт в рабочей области Visual Studio.

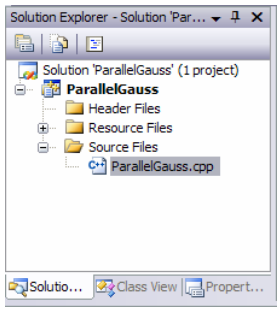

Рис. 3.8. Открытие файла **ParallelGauss.cpp** с использованием Solution Explorer

В файле **ParallelGauss.cpp** подключаются необходимые библиотеки, также в этом файле расположена главная функция (*main*) будущего параллельного приложения, которая содержит строки объявления необходимых переменных, вызовы функций инициализации и остановки среды выполнения MPI-программ, функции для определения числа доступных процессов и рангов процессов:

```
int ProcNum = 0;      // The number of the available processes
int ProcRank = 0;     // The rank of the current process

void main(int argc, char* argv[]) {
  double* pMatrix;        // The matrix of the linear system
  double* pVector;        // The right parts of the linear system
  double* pResult;        // The result vector
  int     Size;           // The size of the matrix and the vectors
  double Start, Finish, Duration;

  setvbuf(stdout, 0, _IONBF, 0);

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
  MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);

  if (ProcRank == 0)
    printf("Parallel Gauss algorithm for solving linear systems\n");
```

```
  MPI_Finalize();
}
```

Заметим, что переменные *ProcNum* и *ProcRank*, как и в предыдущих лабораторных работах, были объявлены глобальными.

Также в файле **ParallelGauss.cpp** расположены функции и переменные, перенесенные сюда из проекта, содержащего последовательный алгоритм Гаусса: глобальная переменная *pSerialPivotPos*, функции *DummyDataInitialization*, *RandomDataInitialization*, *SerialResultCalculation*, *SerialGaussianElimination*, *SerialBackSubstitution*, *SerialEliminateRows*, *FindPivotRow* (использование этих переменных и функций подробно описано в упражнении 3 данной лабораторной работы). Первые две из этих функций будут использоваться в параллельном приложении для инициализации исходных объектов. Остальные функции нужны для того, чтобы иметь возможность выполнить последовательный алгоритм и сравнить результаты выполнения последовательного и параллельного алгоритмов Гаусса.

В данном параллельном приложении для печати матриц и векторов, как и ранее, будем пользоваться функциями *PrintMatrix* и *PrintVector*, реализация этих функций также перенесена в данное параллельное приложение. Кроме того, помещены заготовки для функций инициализации процесса вычислений (*ProcessInitialization*) и завершения процесса (*ProcessTermination*).

Скомпилируйте и запустите приложение стандартными средствами Visual Studio. Убедитесь в том, что в командную консоль выводится приветствие: " `Parallel Gauss algorithm for solving linear systems` ".

Задание 2 – Определение размеров объектов и ввод исходных данных

На следующем этапе разработки параллельного приложения необходимо задать размеры матрицы системы линейных уравнений, вектора правых частей, вектора результатов и выделить память для их хранения. Согласно схеме параллельных вычислений, исходные объекты существуют только на ведущем процессе (процесс с нулевым рангом), на каждом процессе в каждый момент времени располагается полоса матрицы системы линейных уравнений, блок вектора правых частей и блок вектора результатов. Определим переменные для хранения блоков и размера этих блоков:

```
void main(int argc, char* argv[]) {
  double* pMatrix;        // The matrix of the linear system
  double* pVector;        // The right parts of the linear system
  double* pResult;        // The result vector
  double *pProcRows;      // The Rows of matrix A on the process
  double *pProcVector;    // The Elements of vector b on the process
  double *pProcResult;    // The Elements of vector x on the process
  int     Size;           // The Sizes of the initial matrix and vector
  int     RowNum;         // The Number of  the matrix rows on the  current
                          // process
  double Start, Finish, Duration;
```

Для определения размеров матрицы и векторов, вычисления количества строк матрицы, которые будут обрабатываться данным процессом, выделения памяти для хранения исходных объектов и их блоков, а также для определения элементов исходных матрицы и вектора реализуем функцию *ProcessInitialization*.

```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pMatrix, double* &pVector,
  double* &pResult, double* &pProcRows, double* &pProcVector,
  double* &pProcResult, int &Size, int &RowNum)
```

Для определения размера объектов *Size*, как и при выполнении предыдущих лабораторных работ, организуем диалог с пользователем. Приложение, которое будет разрабатываться в рамках данного упражнения, ориентировано на наиболее общий случай: не требуется, чтобы размер объектов был кратен числу доступных процессов, единственное ограничение состоит в том, что размер *Size* должен быть не меньше числа процессов *ProcNum* для того, чтобы на каждый процесс приходилась по крайней мере одна строка матрицы системы линейных уравнений. Если пользователь вводит некорректное число, ему предлагается повторить ввод. Диалог осуществляется только на *ведущем процессе*. Когда размер матрицы и векторов корректно определен, значение переменной *Size* рассылается на все процессы:

```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pMatrix, double* &pVector,
  double* &pResult, double* &pProcRows, double* &pProcVector,
  double* &pProcResult, int &Size, int &RowNum) {
```

```
  if (ProcRank == 0) {
    do {
      printf("\nEnter the size of the matrix and the vector: ");
      scanf("%d", &Size);
      if (Size < ProcNum) {
        printf ("Size must be greater than number of processes! \n");
      }
    } while (Size < ProcNum);
  }
  MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);
}
```

После того, как размер объектов *Size* определен, появляется возможность определить количество строк матрицы, которые будут обрабатываться каждым процессом, а также выделить память для хранения исходных матрицы и вектора, вектора результата, полосы матрицы и блоков векторов. Для определения числа строк *RowNum*, которые будет обрабатывать данный процесс, воспользуемся методикой, описанной в лабораторной работе 1 при разработке параллельного приложения для умножения матрицы на вектор в случае, когда размер объектов не кратен числу процессов:

```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pMatrix, double* &pVector,
  double* &pResult, double* &pProcRows, double* &pProcVector,
  double* &pProcResult, int &Size, int &RowNum) {
  <…>
  MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);

  int RestRows = Size;
  for (int i=0; i<ProcRank; i++)
    RestRows = RestRows-RestRows/(ProcNum-i);
  RowNum = RestRows/(ProcNum-ProcRank);

  pProcRows = new double [RowNum*Size];
  pProcVector = new double [RowNum];
  pProcResult = new double [RowNum];

  if (ProcRank == 0) {
    pMatrix = new double [Size*Size];
    pVector = new double [Size];
    pResult = new double [Size];
  }
}
```

Для определения элементов матрицы системы линейных уравнений *pMatrix* и вектора правых частей *pVector* будем использовать функцию *DummyDataInitialization*, которая была разработана нами при реализации последовательного алгоритма Гаусса:

```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pMatrix, double* &pVector,
  double* &pResult, double* &pProcRows, double* &pProcVector,
  double* &pProcResult, int &Size, int &RowNum) {
  <…>
  if (ProcRank == 0) {
    pMatrix = new double [Size*Size];
    pVector = new double [Size];
    pResult = new double [Size];
    // Initialization of the matrix and the vector elements
    DummyDataInitialization (pMatrix, pVector, Size);
  }
}
```

Вызовем функцию *ProcessInitialization* из основной функции параллельного приложения. Для контроля правильности ввода исходных данных воспользуемся функцией форматированного вывода матрицы *PrintMatrix* и вектора *PrintVector*, распечатаем матрицу системы линейных уравнений и вектор правых частей на ведущем процессе.

```
void main(int argc, char* argv[]) {
  <…>
```

```
  // Memory allocation and definition of object elements
  ProcessInitialization(pMatrix, pVector, pResult, pProcRows, pProcVector,
    pProcResult, Size, RowNum);
  if (ProcRank == 0) {
    printf("Initial matrix \n");
    PrintMatrix(pMatrix, Size, Size);
    printf("Initial vector \n");
    PrintVector(pVector, Size);
  }
  MPI_Finalize();
}
```

Скомпилируйте и запустите приложение. Убедитесь в том, что диалог для ввода размеров объектов позволяет ввести только корректное значение размеров объектов. Проанализируйте значения элементов матрицы и вектора. Если данные задаются верно, то матрица линейной системы должна быть нижнеq треугольной, все элементы расположенные ниже главной диагонали равны 1, элементы вектора правых частей – целые положиетльные числа от 1 до *Size*. (рис. 3.9).

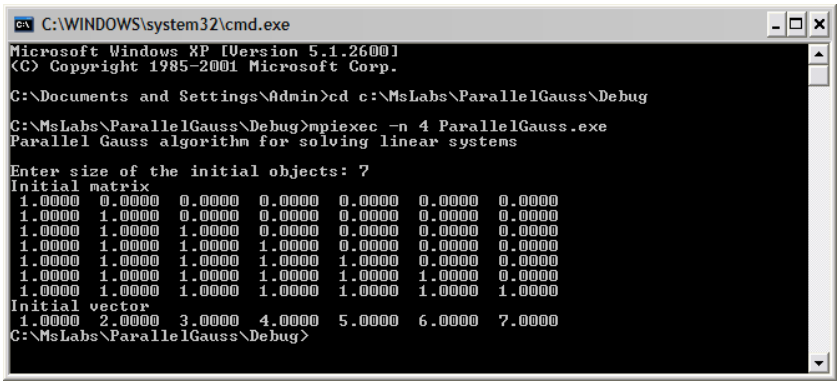

Рис. 3.9. Задание исходных данных

Задание 3 – Завершение процесса вычислений

Для того, чтобы на каждом этапе разработки приложение было завершенным, разработаем функцию для корректной остановки процесса вычислений. Для этого необходимо освободить память, выделенную динамически в процессе выполнения программы. Реализуем соответствующую функцию *ProcessTermination*. На ведущем процессе выделялась память для хранения исходных матрицы *pMatrix* и вектора *pVector* и память для хранения результирующего вектора *pResult*, на всех процессах выделялась память для хранения полосы матрицы *pProcRows* и блоков вектора правых частей *pProcVector* и вектора результата *pProcResult*. Все эти объекты необходимо передать в функцию *ProcessTermination* в качестве аргументов:

```
// Function for computational process termination
void ProcessTermination (double* pMatrix, double* pVector, double* pResult,
  double* pProcRows, double* pProcVector, double* pProcResult) {
  if (ProcRank == 0) {
    delete [] pMatrix;
    delete [] pVector;
    delete [] pResult;
  }
  delete [] pProcRows;
  delete [] pProcVector;
  delete [] pProcResult;
}
```

Вызов функции остановки процесса вычислений необходимо выполнить непосредственно перед завершением параллельной программы:

```
  // Process termination
  ProcessTermination (pMatrix, pVector, pResult, pProcRows, pProcVector,
    pProcResult);
  MPI_Finalize();
}
```

Скомпилируйте и запустите приложение. Убедитесь в том, что приложение работает корректно.

Задание 4 – Распределение данных между процессами

В соответствии со схемой параллельных вычислений, изложенной в предыдущем упражнении, система линейных уравнений должна быть распределена между процессами горизонтальными полосами (разделена на непрерывные последовательности строк).

За разделение данных отвечает функция *DataDistribution*. Ей на вход в качестве аргументов необходимо передать исходные матрицу *pMatrix* и вектор *pVector*, адреса буферов для хранения горизонтальных полос матрицы *pProcRows* и соответствующего блока вектора правых частей *pProcVector*, а также размеры объектов (размер матрицы и вектора *Size* и число полос в горизонтальной полосе *RowNum*):

```
// Data distribution among the processes
void DataDistribution(double* pMatrix, double* pProcRows, double* pVector,
  double* pProcVector, int Size, int RowNum);
```

Для разделения матрицы на горизонтальные полосы и рассылки этих полос воспользуемся процедурой, описанной в лабораторной работе 1 в ходе разработки параллельного приложения умножения матрицы на вектор в случае, когда размер матрицы не кратен числу процессов.

```
// Data distribution among the processes
void DataDistribution(double* pMatrix, double* pProcRows, double* pVector,
  double* pProcVector, int Size, int RowNum) {

  int *pSendNum;     // The number of the elements sent to the process
  int *pSendInd;     // The index of the first data element sent
                     // to the process
  int RestRows=Size; // The number of rows, that have not been
                     // distributed yet
  int i;             // Loop variable

  // Alloc memory for temporary objects
  pSendInd = new int [ProcNum];
  pSendNum = new int [ProcNum];

  // Define the disposition of the matrix rows for the current process
  RowNum = (Size/ProcNum);
  pSendNum[0] = RowNum*Size;
  pSendInd[0] = 0;
  for (i=1; i<ProcNum; i++) {
    RestRows -= RowNum;
    RowNum = RestRows/(ProcNum-i);
    pSendNum[i] = RowNum*Size;
    pSendInd[i] = pSendInd[i-1]+pSendNum[i-1];
  }

  // Scatter the rows
  MPI_Scatterv(pMatrix, pSendNum, pSendInd, MPI_DOUBLE, pProcRows,
    pSendNum[ProcRank], MPI_DOUBLE, 0, MPI_COMM_WORLD);

    // Free the memory
  delete [] pSendNum;
  delete [] pSendInd;
}
```

Для разделения вектора применим ту же последовательность действий. Сделаем лишь небольшое дополнение: при выполнении параллельного алгоритма Гаусса нужно будет иметь возможность по номеру строки определить, на каком из процессов вычислительной системы расположена эта строка и какой номер имеет эта строка в полосе матрицы *pProcRows* этого процесса. Для того, чтобы эффективно решать эту задачу, заведем два глобальных массива: *pProcInd* и *pProcNum*. В каждом из них должно быть расположено по *ProcNum* элементов. Элемент первого массива *pProcInd*[*i*] определяет номер первой строки, расположенной на процессе с рангом *i*. Элемент второго массива *pProcNum*[*i*] определяет количество строк линейной системы, которые обрабатываются процессом с рангом *i*. Объявим соответствующие глобальные переменные, выделим память для этих массивов внутри функции *DataDistribution*, заполним массивы значениями. Заметим, что эти массивы можно использовать для рассылки вектора правых частей при помощи функции *MPI_Scatter*.

```
int* pProcInd;
int* pProcNum;

// Data distribution among the processes
void DataDistribution(double* pMatrix, double* pProcRows, double* pVector,
  double* pProcVector, int Size, int RowNum) {
  <…>
  pProcInd = new int [ProcNum];
  pProcNum = new int [ProcNum];

  RestRows = Size;
  pProcInd[0] = 0;
  pProcNum[0] = Size/ProcNum;
  for (i=1; i<ProcNum; i++) {
    RestRows -= pProcNum[i-1];
    pProcNum[i] = RestRows/(ProcNum-i);
    pProcInd[i] = pProcInd[i-1]+pProcNum[i-1];
  }

  MPI_Scatterv(pVector, pProcNum, pProcInd, MPI_DOUBLE, pProcVector,
    pProcNum[ProcRank], MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
```

Соответственно, вызывать функцию распределения данных из основной программы нужно непосредственно после вызова функции инициализации вычислительного процесса *ProcessInitialization*, перед тем, как приступить к выполнению алгоритма Гаусса:

```
  // Memory allocation and definition of object elements
  ProcessInitialization(pMatrix, pVector, pResult, pProcRows, pProcVector,
    pProcResult, Size, RowNum);

  // Distributing the initial data between the processes
  DataDistribution(pMatrix, pProcRows, pVector, pProcVector, Size, RowNum);
```

Удалим печать исходных объектов после выполнения функции *ProcessInitialization*. Выполним проверку правильности разделения данных между процессами. Для этого после выполнения функции *DataDistribution* распечатаем исходные матрицу и вектор, а затем полосы матрицы и блоки векторов, содержащиеся на каждом из процессов. Добавим в код приложения еще одну функцию, которая служит для проверки правильности выполнения этапа распределения данных, и назовем ее *TestDistribution*.

Для того, чтобы организовать форматированный вывод матрицы и вектора, воспользуемся методами *PrintMatrix* и *PrintVector*:

```
// Function for testing the data distribution
void TestDistribution(double* pMatrix, double* pVector, double* pProcRows,
  double* pProcVector, int Size, int RowNum) {
  if (ProcRank == 0) {
    printf("Initial Matrix: \n");
    PrintMatrix(pMatrix, Size, Size);
    printf("Initial Vector: \n");
    PrintVector(pVector, Size);
  }
  for (int i=0; i<ProcNum; i++) {
    if (ProcRank == i) {
      printf("\nProcRank = %d \n", ProcRank);
      printf(" Matrix Stripe:\n");
      PrintMatrix(pProcRows, RowNum, Size);
      printf(" Vector: \n");
      PrintVector(pProcVector, RowNum);
    }
    MPI_Barrier(MPI_COMM_WORLD);
  }
}
```

Добавьте вызов функции проверки распределения непосредственно после функции *DataDistribution*:

```
  // Distributing the initial data between the processes
  DataDistribution(pMatrix, pProcRows, pVector, pProcVector, Size, RowNum);
```

```
  TestDistribution(pMatrix, pVector, pProcRows, pProcVector, Size, RowNum);
```

Скомпилируйте приложение. Если в приложении обнаружились ошибки, исправьте их, сверяя свой код с кодом, представленным в лабораторной работе. Запустите приложение. Убедитесь в том, что данные распределяются правильно (рис. 3.10).

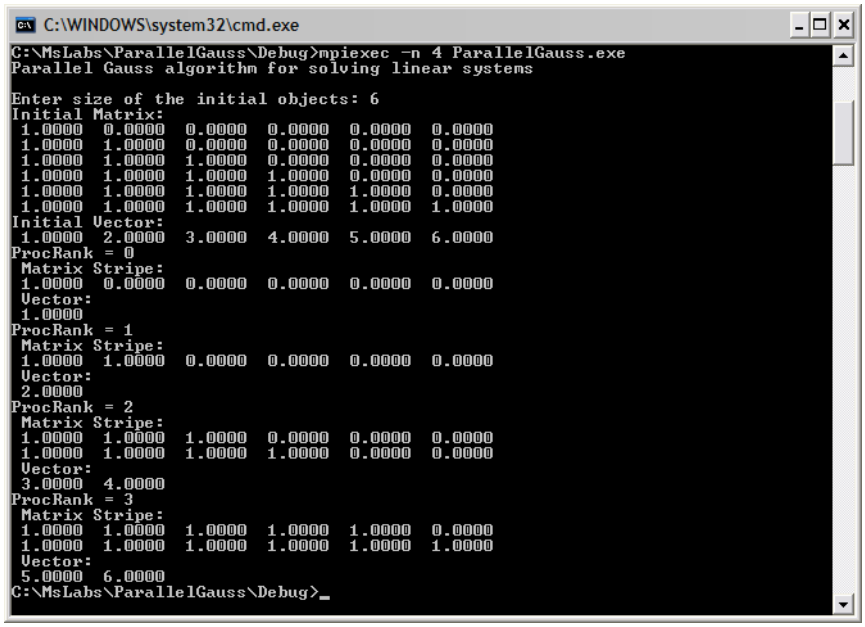

Рис. 3.10. Распределение данных в случае, когда приложение запускается на четырех процессах, а порядок системы уравнений равен шести

Задание 5 – Параллельное выполнение прямого хода алгоритма Гаусса

Согласно вычислительной схеме алгоритма Гаусса решения систем линейных уравнений, метод состоит из двух этапов: прямого (*Gaussian Elimination*) и обратного (*Back Substitution*) хода. Для выполнения параллельного алгоритма Гаусса реализуем функцию *ParallelResultCalculation*, которая содержит вызовы функций для выполнения прямого и обратного хода алгоритма:

```
// Function for execution of the parallel Gauss algorithm
void ParallelResultCalculation(double* pProcRows, double* pProcVector,
  double* pProcResult, int Size, int RowNum) {
  // Gaussian elimination
  ParallelGaussianElimination (pProcRows, pProcVector, Size, RowNum);
  // Back substitution
  ParallelBackSubstitution (pProcRows, pProcVector, pProcResult, Size,
    RowNum);
}
```

Для реализации параллельной версии алгоритма Гаусса нам потребуются два дополнительных массива: *pParallelPivotPos* и *pProcPivotIter*.

Элементы массива *pParallelPivotPos* определяют номера строк матрицы, выбираемых в качестве ведущих, по итерациям прямого хода метода Гаусса. Именно в этом порядке должны выполняться затем итерации обратного хода для определения значений неизвестных системы линейных уравнений. Массив *pParallelPivotPos* является глобальным и любое его изменение в одном из процессов требует выполнения операции рассылки измененных данных всем остальным процессам программы.

Элементы массива *pProcPivotIter* определяют номера итераций прямого хода метода Гаусса, на которых строки процесса использовались в качестве ведущих (т.е., строка *i* процесса выбиралась ведущей на итерации *pProcPivotIter[i]*). Начальное значение элементов массива устанавливается равным -1 и, тем самым, такое значение элемента массива *pProcPivotIter[i]* является признаком того, что строка *i* процесса все еще подлежит обработке. Кроме того, важно отметить, что запоминаемые в элементах массива *pProcPivotIter* номера итераций дополнительно означают и номера неизвестных, для определения которых будут использованы соответствующие строкам уравнения. Массив *pProcPivotIter* является локальным для каждого процесса.

Объявим соответствующие глобальные переменные:

```
int *pParallelPivotPos; // The number of rows selected as the pivot ones
```

```
int *pProcPivotIter;    // The number of iterations, at which the processor
                        // rows were used as the pivot ones
```

Выделим память для хранения этих объектов до начала выполнения этапов параллельного метода Гаусса, после завершения обратного хода метода Гаусса освободим выделенную память:

```
// Function for the execution of the parallel Gauss algorithm
 void ParallelResultCalculation(double* pProcRows, double* pProcVector,
  double* pProcResult, int Size, int RowNum) {

  // Memory allocation
  pParallelPivotPos = new int [Size];
  pProcPivotIter = new int [RowNum];
  for (int i=0; i<RowNum; i++)
    pProcPivotIter[i] = -1;

  // Gaussian elimination
  ParallelGaussianElimination (pProcRows, pProcVector, Size, RowNum);
  // Back substitution
  ParallelBackSubstitution (pProcRows, pProcVector, pProcResult, Size,
    RowNum);

  // Memory deallocation
  delete [] pParallelPivotPos;
  delete [] pProcPivotIter;
}
```

Далее в данном задании будет выполнена реализация прямого хода метода Гаусса. Обратный ход метода Гаусса будет выполнен в следующем задании лабораторной работы.

Итак, для параллельного выполнения прямого хода метода Гаусса предназначена функция *ParallelGaussianElimination*. В качестве аргументов этой функции передаются полоса матрицы системы линейных уравнений, которую обрабатывает данный процесс (*pProcRows*), и блок вектора правых частей *pProcVector*, размер исходных объектов *Size* и количество строк в полосе *RowNum*.

```
  // Gaussian elimination
void ParallelGaussianElimination (double* pProcRows, double* pProcVector,
  int Size, int RowNum);
```

Назначение этой функции – путем эквивалентных преобразований привести матрицу системы линейных уравнений к верхнему треугольному виду.

Количество итераций прямого хода алгоритма Гаусса равно порядку системы линейных уравнений. На каждой итерации выбирается ведущая строка с использованием метода главных элементов. Поскольку строки матрицы системы линейных уравнений распределены по подзадачам, для поиска максимального значения подзадачи должны обменяться своими элементами при исключаемой переменной (на итерации *i* прямого хода исключается *i*-ая неизвестная). После сбора всех необходимых данных в каждой подзадаче может быть определено, какая из подзадач содержит ведущую строку и какое значение является ведущим элементом.

Реализуем процедуру выбора ведущей строки за два этапа. На первом этапе выбираются локальные ведущие строки на каждом процессе. Для этого нужно просмотреть все строки, которые подлежат обработке (строка с номером *i* подлежит обработке, если значение элемента *pProcPivotIter[i]* равно -1), и выбрать среди них ту, которая содержит максимальный по абсолютному значению коэффициент при исключаемой неизвестной:

```
  // Gaussian elimination
void ParallelGaussianElimination (double* pProcRows, double* pProcVector,
  int Size, int RowNum) {
  double MaxValue;   // The value of the pivot element of the process
  int    PivotPos;   // The Position of the pivot row in the stripe
                     // of the process
  // The iterations of the Gaussian elimination
  for (int i=0; i<Size; i++) {
    // Calculating the local pivot row
    for (int j=0; j<RowNum; j++) {
      if ((pProcPivotIter[j] == -1) &&
         (MaxValue < fabs(pProcRows[j*Size+i]))) {
        MaxValue = fabs(pProcRows[j*Size+i]);
```

```
        PivotPos = j;
      }
    }
  }
}
```

После определения ведущей строки на каждом процессе, необходимо выбрать максимальный среди полученных ведущих элементов и определить, на каком процессе он располагается. Для выполнения таких действий в библиотеке MPI предназначена функция *MPI_Allreduce*. Функция имеет следующий интерфейс:

```
int MPI_Allreduce(void *sendbuf, void *recvbuf,int count,MPI_Datatype type,
  MPI_Op op,MPI_Comm comm),
где
   - sendbuf - буфер памяти с отправляемым сообщением,
   - recvbuf – буфер памяти для результирующего сообщения,
   - count   - количество элементов в сообщениях,
   - type    – тип элементов сообщений,
   - op      - операция, которая должна быть выполнена над данными,
   - comm    - коммуникатор, в рамках которого выполняется операция.
```

Выполним редукцию данных для определения значения ведущего элемента и процесса, на котором расположена ведущая строка:

```
  // Gaussian elimination
void ParallelGaussianElimination (double* pProcRows, double* pProcVector,
  int Size, int RowNum) {
  double MaxValue;  // The value of the pivot element of the process
  int PivotPos;     // The position of the pivot row in the stripe
                    // of the process
  struct { double MaxValue; int ProcRank; } ProcPivot, Pivot;
  // The iterations of the Gaussian elimination
  for (int i=0; i<Size; i++) {
    <…>
    // Finding the global pivot row
    ProcPivot.MaxValue = MaxValue;
    ProcPivot.ProcRank = ProcRank;
    // Finding the pivot process
    MPI_Allreduce(&ProcPivot, &Pivot, 1, MPI_DOUBLE_INT, MPI_MAXLOC,
      MPI_COMM_WORLD);
  }
}
```

После выполнения операции редукции данных в структуре *Pivot* будет хранится значение ведущего элемента и номер процесса, на котором расположена соответствующая ведущая строка.

На процессе, где расположена ведущая строка, заполним соответствующий элемент массива *pProcPivotIter*. Кроме того, занесем номер ведущей строки в глобальный массив *pPrallelPivotPos* (нам известен номер процесса, на котором расположена ведущая строка, и номер строки в рамках полосы, которая расположена на данном процессе, по этим данным можно определить номер строки в системе уравнений, пользуясь значениями в массивах *pProcInd* и *pProcNum*).

```
  // Gaussian elimination
void ParallelGaussianElimination (double* pProcRows, double* pProcVector,
  int Size, int RowNum) {
  double MaxValue;  // The value of the pivot element of the process
  int PivotPos;     // The position of the pivot row in the stripe
                    // of the process
  struct { double MaxValue; int ProcRank; } ProcPivot, Pivot;
  // The iterations of the Gaussian elimination stage
  for (int i=0; i<Size; i++) {
    <…>
    MPI_Allreduce(&ProcPivot, &Pivot, 1, MPI_DOUBLE_INT, MPI_MAXLOC,
      MPI_COMM_WORLD);
    // Storing the number of the pivot row
    if ( ProcRank == Pivot.ProcRank ){
      pProcPivotIter[PivotPos]= i;
      pParallelPivotPos[i]= pProcInd[ProcRank] + PivotPos;
```

```
    }
    MPI_Bcast(&pParallelPivotPos[i], 1, MPI_INT, Pivot.ProcRank,
      MPI_COMM_WORLD);
  }
}
```

Для выполнения преобразований оставшихся строк матрицы необходимо разослать ведущую строку и соответствующий элемент вектора правых частей на все процессы. Заведем буфер для хранения ведущей строки, на процессе, ранг которого был определен в ходе выполнения редукции (*Pivot.ProcRank*) скопируем строку в буфер и выполним широковещательную рассылку:

```
  // Gaussian elimination
void ParallelGaussianElimination (double* pProcRows, double* pProcVector,
  int Size, int RowNum) {
  double MaxValue;  // The value of the pivot element of the process
  int PivotPos;     // Position of  the pivot row in the stripe
                    // of the process
  struct { double MaxValue; int ProcRank; } ProcPivot, Pivot;
  double *pPivotRow;   // Pivot row of the current iteration
  pPivotRow = new double [Size+1];
  // The iterations of the Gaussian elimination stage
  for (int i=0; i<Size; i++) {
    <…>
    MPI_Bcast(&pParallelPivotPos[i], 1, MPI_INT, Pivot.ProcRank,
      MPI_COMM_WORLD);
    // Broadcasting the pivot row
    if ( ProcRank == Pivot.ProcRank ){
      // Fill the pivot row
      for (int j=0; j<Size; j++) {
        pPivotRow[j] = pProcRows[PivotPos*Size + j];
      }
      pPivotRow[Size] = pProcVector[PivotPos];
    }
    MPI_Bcast(pPivotRow, Size+1, MPI_DOUBLE, Pivot.ProcRank,
      MPI_COMM_WORLD);
  }
  delete [] pPivotRow;
}
```

Получив ведущую строку, подзадачи выполняют вычитание строк, обеспечивая тем самым исключение соответствующей неизвестной. Реализуем вычитание с помощью функции *ParallelEliminateRows*:

```
// Fuction for column elimination
void ParallelEliminateColumns(double* pProcRows, double* pProcVector,
  double* pPivotRow, int Size, int RowNum, int Iter) {
  double PivotFactor;
  for (int i=0; i<RowNum; i++) {
    if (pProcPivotIter[i] == -1) {
      PivotFactor = pProcRows[i*Size+Iter] / pPivotRow[Iter];
      for (int j=Iter; j<Size; j++) {
        pProcRows[i*Size + j] -= PivotFactor* pPivotRow[j];
      }
      pProcVector[i] -= PivotFactor * pPivotRow[Size];
    }
  }
}
```

Вызовем функцию вычитания из функции, выполняющей параллельный алгоритм прямого хода метода Гаусса:

```
  // Gaussian elimination
void ParallelGaussianElimination (double* pProcRows, double* pProcVector,
  int Size, int RowNum) {
  <…>
  for (int i=0; i<Size; i++) {
    <…>
```

```
    MPI_Bcast(pPivotRow, Size+1, MPI_DOUBLE, Pivot.ProcRank,
      MPI_COMM_WORLD);
    // Column elimination
    ParallelEliminateColumns(pProcRows, pProcVector, pPivotRow, Size,
      RowNum, i);
  }
  delete [] pPivotRow;
}
```

Удалите вызов функции тестирования этапа распределения данных. Закомментируйте вызов функции, выполняющей обратный ход метода Гаусса *ParallelBackSubstitution*. Для контроля правильности выполнения прямого хода метода Гаусса вызовите функцию *TestDistribution* непосредственно после *ParallelResultCalculation*:

```
  // The execution of the parallel Gauss algorithm
  ParallelResultCalculation (pProcRows, pProcVector, pProcResult Size,
    RowNum);
  TestDistribution(pMatrix, pVector, pProcRows, pProcVector, Size, RowNum);
```

Скомпилируйте и запустите приложение. Напомним, что после выполнения прямого хода метода Гаусса матрица должна быть приведена к верхнему треугольному виду. Запустите приложение. Убедитесь в том, что реализованный алгоритм работает корректно (рис. 3.11).

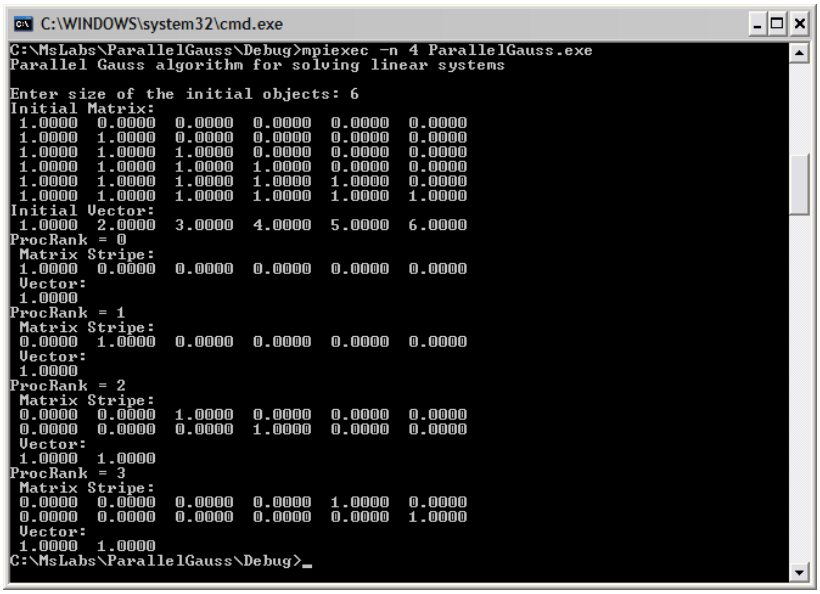

Рис. 3.11. Результат выполнения прямого хода метода Гаусса

Задание 7 – Параллельное выполнение обратного хода алгоритма Гаусса

При выполнении обратного хода метода Гаусса процессы выполняют необходимые вычисления для нахождения значения неизвестных. Как только какой-либо процесс определяет значение своей переменной, это значение должно быть разослано всем процессам для того, чтобы они могли подставить полученное значение новой неизвестной и выполнить корректировку значений для элементов вектора правых частей.

Выполнение обратного хода состоит из *Size* итераций. На каждой итерации необходимо определить строку, из которой можно вычислить значение очередного элемента результирующего вектора. Номер этой строки хранится в массиве *pParallelPivotIter*. По номеру необходимо определить номер процесса, на котором эта строка хранится, и номер этой строки в полосе *pProcRows* этого процесса. Реализуем для выполнения этих действий функцию *FindBackPivotRow*:

```
// Function for finding the pivot row of the back substitution
void FindBackPivotRow(int RowIndex, int Size,int &IterProcRank,
  int &IterPivotPos) {
  for (int i=0; i<ProcNum-1; i++) {
    if ((pProcInd[i]<=RowIndex) && (RowIndex<pProcInd[i+1]))
      IterProcRank = i;
  }
  if (RowIndex >= pProcInd[ProcNum-1])
```

```
    IterProcRank = ProcNum-1;
  IterPivotPos = RowIndex - pProcInd[IterProcRank];
}
```

На вход этой функции передается номер строки *RowIndex*, для которой определяется расположение, а также размер исходных объектов *Size*. Функция передает в переменную *IterProcRank* ранг того процесса, на котором располагается строка *RowIndex*, а в переменную *IterPivotPos* – номер этой строки в буфере *pProcRows*.

После того, как положение строки определено, процесс, содержащий эту строку, вычисляет значение соответствующего элемента результирующего вектора и рассылает его всем процессам, далее процессы выполняют преобразование своих строк матриц:

```
  // Back substitution
void ParallelBackSubstitution (double* pProcRows, double* pProcVector,
  double* pProcResult, int Size, int RowNum) {
  int IterProcRank;    // The rank of the process with the current
                       // pivot row
  int IterPivotPos;    // The position of the pivot row of the process
  double IterResult;   // The calculated value of the current unknown
  double val;

  // The Iterations of the back substitution
  for (int i=Size-1; i>=0; i--) {

    // Calculating the rank of the process, which holds the pivot row
    FindBackPivotRow(pParallelPivotPos[i],Size,IterProcRank,IterPivotPos);

    // Calculating the unknown
    if (ProcRank == IterProcRank) {
      IterResult = pProcVector[IterPivotPos] /
                   pProcRows[IterPivotPos*Size+i];
      pProcResult[IterPivotPos] = IterResult;
    }
    // Broadcasting the value of the current unknown
    MPI_Bcast(&IterResult, 1, MPI_DOUBLE, IterProcRank, MPI_COMM_WORLD);

    // Updating the values of the vector
    for (int j=0; j<RowNum; j++)
      if ( pProcPivotIter[j] < i ) {
        val = pProcRows[j*Size + i] * IterResult;
        pProcVector[j]=pProcVector[j] - val;
    }
  }
}
```

Раскомментируйте вызов функции, выполняющей обратный ход алгоритма Гаусса. После выполнения параллельного алгоритма Гаусса, распечатайте блоки результирующего вектора с каждого процесса параллельного приложения. Скомпилируйте и запустите приложение. Оцените правильность работы алгоритма: если для генерации исходных данных используется функция *DummyDataInitialization*, то все элементы результирующего вектора должны быть равны 1.

Задание 8 – Сбор результатов

После выполнения обратного хода алгоритма Гаусса на каждом процессе расположены блоки результирующего вектора. Необходимо собрать результирующий вектор на ведущем процессе. Выполним сбор при помощи функции *MPI_Gatherv*. Массивы, необходимые для вызова этой функции уже были определены нами при выполнении функции *DataDistribution*. Значит, функция, выполняющая сбор, имеет очень простую реализацию:

```
// Function for gathering the result vector
void ResultCollection(double* pProcResult, double* pResult) {
  //Gathering the result vector on the pivot processor
  MPI_Gatherv(pProcResult, pProcNum[ProcRank], MPI_DOUBLE, pResult,
    pProcNum, pProcInd, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
```

Добавьте вызов функции сбора результирующего вектора в основную функцию параллельного приложения.

```
// The execution of the parallel Gauss algorithm
  ParallelResultCalculation(pProcRows, pProcVector, pProcResult,
    Size, RowNum);

  // Gathering the result vector
  ResultCollection(pProcResult, pResult);
```

Собранный вектор состоит из элементов вектора неизвестных, расположенных в порядке выбора ведущих строк на этапе выполнения прямого хода метода Гаусса. Порядок выбора ведущих строк хранится в глобальном массиве *pParallelPivotPos*. При печати результирующего вектора необходимо учитывать этот порядок. Разработаем функцию печати результирующего вектора *PrintResultVector*:

```
// Function for formatted result vector output
void PrintResultVector (double* pResult, int Size) {
  int i;
  for (i=0; i<Size; i++)
    printf("%7.4f ", pResult[pParallelPivotPos[i]]);
}
```

Распечатайте результирующий вектор на ведущем процессе при помощи функции *PrintResultVector*:

```
  // Gathering the result vector
  ResultCollection(pProcResult, pResult);
  if (ProcRank == 0) {
    printf ("Result vector \n");
    PrintResultVector(pResult, Size);
  }
```

Скомпилируйте и запустите приложение. Проверьте правильность выполнения алгоритма: если метод Гаусса реализован верно, все элементы результирующего вектора должны быть равны 1 (при использовании функции *DummyDataInitialization* для генерации исходных данных).

Задание 9 – Проверка правильности работы программы

Теперь, после выполнения функции сбора, необходимо проверить правильность выполнения алгоритма. Для этого разработаем функцию *TestResult*. Для проверки правильности выполнения алгоритма необходимо умножить матрицу линейной системы на полученный вектор неизвестных (с учетом порядка элементов в нем), а затем поэлементно сравнить вектор, полученный в результате такого умножения с заданным вектором правых частей *pVector*. Получение каждого элемента результирующего вектора требует выполнения последовательности умножений и сложений дробных чисел. Порядок выполнения этих действий может повлиять на наличие и величину машинной погрешности. Поэтому в данном случае нельзя проверять элементы двух векторов (*pRightPartVector* и *pVector*) на равенство. Введем допустимую величину расхождения результатов *Accuracy*. Вектора будем считать равными в том случае, когда соответствующие элементы отличаются не более чем на величину допустимой погрешности *Accuracy*.

Функция *TestResult* должна иметь доступ к матрице системы линейных уравнений *pMatrix* и вектору правых частей *pVector*, а значит может быть выполнена только на ведущем процессе:

```
// Function for testing the result
void TestResult (double* pMatrix, double* pVector, double* pResult,
  int Size) {
  /* Buffer for storing the vector, that is a result of multiplication
     of the linear system matrix by the vector of unknowns */
  double* pRightPartVector;
  // Flag, that shows wheather the right parts vectors are identical or not
  int equal = 0;
  double Accuracy = 1.e-6; // Comparison accuracy

  if (ProcRank == 0) {
    pRightPartVector = new double [Size];
      for (int i=0; i<Size; i++) {
        pRightPartVector[i] = 0;
        for (int j=0; j<Size; j++) {
          pRightPartVector[i] +=
```

```
            pMatrix[i*Size+j]*pResult[pParallelPivotPos[j]];
        }
      }

    for (int i=0; i<Size; i++) {
        if (fabs(pRightPartVector[i]-pVector[i]) > Accuracy))
        equal = 1;
    }
    if (equal == 1)
      printf("The result of the parallel Gauss algorithm is NOT correct."
             "Check your code.");
    else
      printf("The result of the parallel Gauss algorithm is correct.");
      delete [] pRightPartVector;
  }
}
```

Результатом работы этой функции является печать диагностического сообщения. Используя эту функцию, можно проверять результат работы параллельного алгоритма независимо от того, насколько велики исходные объекты при любых значениях исходных данных.

Закомментируйте вызовы функций, использующих отладочную печать, которые ранее использовались для контроля правильности выполнения этапов параллельного приложения. Вместо функции *DummyDataInitialization*, которая генерирует систему уравнений простого вида, вызовите функцию *RandomDataInitialization*, которая генерирует систему уравнений с нижней треугольной матрицей, где ненулевые элементы задаются при помощи датчика случайных чисел. Скомпилируйте и запустите приложение. Задавайте различные объемы исходных данных. Убедитесь в том, что приложение работает правильно.

Задание 10 – Проведение вычислительных экспериментов

Определим время выполнения параллельного алгоритма. Для этого добавим в программный код замеры времени. Поскольку параллельный алгоритм включает этап распределения данных, вычисления блока частичных результатов на каждом процессе и сбора результата, то отсчет времени должен начинаться непосредственно перед вызовом функции *DataDistribution*, и останавливаться сразу после выполнения функции *ResultCollection*:

```
  <…>
  Start = MPI_Wtime();

  // Distributing the initial data between the processes
  DataDistribution(pMatrix, pProcRows, pVector, pProcVector, Size, RowNum);
  // The execution of the parallel Gauss algorithm
  ParallelResultCalculation(pProcRows, pProcVector, pProcResult,
    Size, RowNum);
  // Gathering the result vector
  ResultCollection(pProcResult, pResult, Size, RowNum);

  Finish = MPI_Wtime();
  Duration = Finish-Start;

  // Testing the result
  TestResult(pMatrix, pVector, pResult, Size);

  // Printing the time spent by parallel Gauss algorithm
  if (ProcRank == 0)
    printf("\n Time of execution: %f\n", Duration);
```

Очевидно, что таким образом будет распечатано то время, которое было затрачено на выполнение вычислений нулевым процессом. Возможно, что время выполнения алгоритма другими процессами немного от него отличается. Но на этапе разработки параллельного алгоритма мы особое внимание уделили равномерной загрузке (*балансировке*) процессов, поэтому теперь у нас есть о снования полагать, что время выполнения алгоритма другими процессами несущественно отличается от приведенного.

Добавьте выделенный фрагмент кода в тело основной функции приложения. Скомпилируйте и запустите приложение. Заполните таблицу:

Таблица 3.3. Время выполнения параллельного алгоритма Гаусса решения систем линейных уравнений и ускорение

Номер теста	Порядок системы	Последовательный алгоритм	Параллельный алгоритм					
			2 процесса		4 процесса		8 процессов	
			Время	Ускорение	Время	Ускорение	Время	Ускорение
1	10							
2	100							
3	500							
4	1000							
5	1500							
6	2000							
7	2500							
8	3000							

В графу "Последовательный алгоритм" внесите время выполнения последовательного алгоритма, замеренное при проведении тестирования последовательного приложения в упражнении 3. Для того, чтобы вычислить ускорение, разделите время выполнения последовательного алгоритма на время выполнения параллельного алгоритма. Результат поместите в соответствующую графу таблицы.

Для того, чтобы оценить теоретическое время выполнения параллельного алгоритма, реализованного согласно вычислительной схеме, приведенной в упражнении 4, можно воспользоваться следующим соотношением:

$$T_p = \frac{1}{p}\sum_{i=2}^{n}(3i + 2i^2)\,\tau + (n-1)\cdot\log_2 p\cdot(3\alpha + w(n+2)/\beta) \qquad (3.6)$$

(подробный вывод этой формулы приведен в разделе 9 "Параллельные методы решения систем линейных уравнений" учебных материалов курса). Здесь n – размер системы линейных уравнений, p – количество процессов, τ – время выполнения одной скалярной операции (значение было нами вычислено при тестировании последовательного алгоритма), α – латентность а β – пропускная способность сети передачи данных. В качестве значений латентности и пропускной способности следует использовать величины, полученные при выполнении лабораторной работы "Выполнение заданий под управлением Microsoft Compute Cluster Server 2003".

Вычислите теоретическое время выполнения параллельного алгоритма по формуле (3.6). Результаты занесите в таблицу 3.4:

Таблица 3.4. Сравнение реального времени выполнения параллельного алгоритма со временем, вычисленным теоретически

Номер теста	Порядок системы	2 процесса		4 процесса		8 процессов	
		Модель	Эксперимент	Модель	Эксперимент	Модель	Эксперимент
1	10						
2	100						
3	500						
4	1000						
5	1500						
6	2000						
7	2500						
8	3000						

Контрольные вопросы

- Насколько сильно отличаются время, затраченное на выполнение последовательного и параллельного алгоритма? Почему?
- Получилось ли ускорение в случае, когда порядок системы уравнений равен 10? Почему?
- Насколько хорошо совпадают время, полученное теоретически, и реальное время выполнения алгоритма? В чем может состоять причина несовпадений?

Задания для самостоятельной работы

1. Изучите метод сопряженных градиентов решения систем линейных уравнений. Выполните реализацию последовательного и параллельного вариантов этого метода.

Приложение 1. Программный код последовательного алгоритма Гаусса решения линейных систем

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <time.h>
#include <math.h>

int* pSerialPivotPos;   // The Number of pivot rows selected at the
iterations
int* pSerialPivotIter; // The Iterations, at which the rows were pivots

// Function for simple initialization of the matrix and the vector elements
void DummyDataInitialization (double* pMatrix, double* pVector, int Size) {
  int i, j;  // Loop variables

  for (i=0; i<Size; i++) {
    pVector[i] = i+1;
      for (j=0; j<Size; j++) {
      if (j <= i)
          pMatrix[i*Size+j] = 1;
        else
          pMatrix[i*Size+j] = 0;
      }
  }
}

// Function for random initialization of the matrix and the vector elements
void RandomDataInitialization (double* pMatrix, double* pVector, int Size)
{
  int i, j;  // Loop variables
  srand(unsigned(clock()));
  for (i=0; i<Size; i++) {
    pVector[i] = rand()/double(1000);
      for (j=0; j<Size; j++) {
        if (j <= i)
        pMatrix[i*Size+j] = rand()/double(1000);
        else
        pMatrix[i*Size+j] = 0;
      }
  }
}

// Function for memory allocation and definition of the objects elements
void ProcessInitialization (double* &pMatrix, double* &pVector,
    double* &pResult, int &Size) {
  // Setting the size of the matrix and the vector
  do {
    printf("\nEnter size of the matrix and the vector: ");
    scanf("%d", &Size);
    printf("\nChosen size = %d \n", Size);

      if (Size <= 0)
      printf("\nSize of objects must be greater than 0!\n");
  } while (Size <= 0);

  // Memory allocation
  pMatrix = new double [Size*Size];
  pVector = new double [Size];
  pResult = new double [Size];

  // Initialization of the matrix and the vector elements
```

```
  DummyDataInitialization(pMatrix, pVector, Size);
  //RandomDataInitialization(pMatrix, pVector, Size);
}

// Function for formatted matrix output
void PrintMatrix (double* pMatrix, int RowCount, int ColCount) {
  int i, j; // Loop variables
  for (i=0; i<RowCount; i++) {
    for (j=0; j<ColCount; j++)
        printf("%7.4f ", pMatrix[i*RowCount+j]);
      printf("\n");
  }
}

// Function for formatted vector output
void PrintVector (double* pVector, int Size) {
  int i;
  for (i=0; i<Size; i++)
    printf("%7.4f ", pVector[i]);
}

// Finding the pivot row
int FindPivotRow(double* pMatrix, int Size, int Iter) {
  int PivotRow = -1; // The index of the pivot row
  int MaxValue =  0; // The value of the pivot element
  int i;             // Loop variable

  // Choose the row, that stores the maximum element
  for (i=0; i<Size; i++) {
    if ((pSerialPivotIter[i] == -1) &&
       (fabs(pMatrix[i*Size+Iter]) > MaxValue)) {
      PivotRow = i;
      MaxValue = fabs(pMatrix[i*Size+Iter]);
    }
  }
  return PivotRow;
}

// Column elimination
void SerialColumnElimination (double* pMatrix, double* pVector, int Pivot,
  int Iter, int Size) {
  double PivotValue, PivotFactor;
  PivotValue = pMatrix[Pivot*Size+Iter];
  for (int i=0; i<Size; i++) {
    if (pSerialPivotIter[i] == -1) {
      PivotFactor = pMatrix[i*Size+Iter] / PivotValue;
      for (int j=Iter; j<Size; j++) {
        pMatrix[i*Size + j] -= PivotFactor * pMatrix[Pivot*Size+j];
      }
      pVector[i] -= PivotFactor * pVector[Pivot];
    }
  }
}

// Gaussian elimination
void SerialGaussianElimination(double* pMatrix,double* pVector,int Size) {
  int Iter;       // The number of the iteration of the gaussian
                  // elimination
  int PivotRow;   // The number of the current pivot row
  for (Iter=0; Iter<Size; Iter++) {
    // Finding the pivot row
    PivotRow = FindPivotRow(pMatrix, Size, Iter);
    pSerialPivotPos[Iter] = PivotRow;
```

```
    pSerialPivotIter[PivotRow] = Iter;
    SerialColumnElimination(pMatrix, pVector, PivotRow, Iter, Size);
  }
}

// Back substution
void SerialBackSubstitution (double* pMatrix, double* pVector,
  double* pResult, int Size) {
  int RowIndex, Row;
  for (int i=Size-1; i>=0; i--) {
    RowIndex = pSerialPivotPos[i];
    pResult[i] = pVector[RowIndex]/pMatrix[Size*RowIndex+i];
    for (int j=0; j<i; j++) {
      Row = pSerialPivotPos[j];
      pVector[j] -= pMatrix[Row*Size+i]*pResult[i];
      pMatrix[Row*Size+i] = 0;
    }
  }
}



// Function for the execution of Gauss algorithm
void SerialResultCalculation(double* pMatrix, double* pVector,
  double* pResult, int Size) {

  // Memory allocation
  pSerialPivotPos  = new int [Size];
  pSerialPivotIter = new int [Size];
  for (int i=0; i<Size; i++) {
    pSerialPivotIter[i] = -1;
  }
  // Gaussian elimination
  SerialGaussianElimination (pMatrix, pVector, Size);
  // Back substitution
  SerialBackSubstitution (pMatrix, pVector, pResult, Size);

  // Memory deallocation
  delete [] pSerialPivotPos;
  delete [] pSerialPivotIter;
}

// Function for computational process termination
void ProcessTermination (double* pMatrix, double* pVector, double* pResult)
{
  delete [] pMatrix;
  delete [] pVector;
  delete [] pResult;
}


void main() {
  double* pMatrix;  // The matrix of the linear system
  double* pVector;  // The right parts of the linear system
  double* pResult;  // The result vector
  int Size;         // The sizes of the initial matrix and the vector
  time_t start, finish;
  double duration;

  printf("Serial Gauss algorithm for solving linear systems\n");
  // Memory allocation and definition of objects' elements
  ProcessInitialization(pMatrix, pVector, pResult, Size);
```

```
  // The matrix and the vector output
  printf ("Initial Matrix \n");
  PrintMatrix(pMatrix, Size, Size);
  printf("Initial Vector \n");
  PrintVector(pVector, Size);

  // Execution of Gauss algorithm
  start = clock();
  SerialResultCalculation(pMatrix, pVector, pResult, Size);
  finish = clock();
  duration = (finish-start)/double(CLOCKS_PER_SEC);

  // Printing the result vector
  printf ("\n Result Vector: \n");
  PrintVector(pResult, Size);

  // Printing the execution time of Gauss method
  printf("\n Time of execution: %f\n", duration);

  // Computational process termination
  ProcessTermination(pMatrix, pVector, pResult);
  getch();
}
```

Приложение 2. Программный код параллельного алгоритма Гаусса решения линейных систем

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <time.h>
#include <math.h>
#include <mpi.h>


int ProcNum;            // Number of the available processes
int ProcRank;           // Rank of the current process
int *pParallelPivotPos; // Number of rows selected as the pivot ones
int *pProcPivotIter;    // Number of iterations, at which the processor
                        // rows were used as the pivot ones

int* pProcInd; // Number of the first row located on the processes
int* pProcNum; // Number of the linear system rows located on the processes

// Function for simple definition of matrix and vector elements
void DummyDataInitialization (double* pMatrix, double* pVector, int Size) {
  int i, j;  // Loop variables

  for (i=0; i<Size; i++) {
    pVector[i] = i+1;
      for (j=0; j<Size; j++) {
      if (j <= i)
          pMatrix[i*Size+j] = 1;
        else
          pMatrix[i*Size+j] = 0;
      }
  }
}

// Function for random definition of matrix and vector elements
void RandomDataInitialization(double* pMatrix, double* pVector, int Size) {
  int i, j;  // Loop variables
  srand(unsigned(clock()));
```

```
  for (i=0; i<Size; i++) {
    pVector[i] = rand()/double(1000);
      for (j=0; j<Size; j++) {
        if (j <= i)
        pMatrix[i*Size+j] = rand()/double(1000);
        else
        pMatrix[i*Size+j] = 0;
      }
  }
}

// Function for memory allocation and data initialization
void ProcessInitialization (double* &pMatrix, double* &pVector,
  double* &pResult, double* &pProcRows, double* &pProcVector,
  double* &pProcResult, int &Size, int &RowNum) {

  int RestRows; // Number of rows, that haven't been distributed yet
  int i;        // Loop variable

  if (ProcRank == 0) {
    do {
      printf("\nEnter the size of the matrix and the vector: ");
      scanf("%d", &Size);
      if (Size < ProcNum) {
        printf("Size must be greater than number of processes! \n");
      }
    }
    while (Size < ProcNum);
  }
  MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);

  RestRows = Size;
  for (i=0; i<ProcRank; i++)
    RestRows = RestRows-RestRows/(ProcNum-i);
  RowNum = RestRows/(ProcNum-ProcRank);

  pProcRows = new double [RowNum*Size];
  pProcVector = new double [RowNum];
  pProcResult = new double [RowNum];

  pParallelPivotPos = new int [Size];
  pProcPivotIter = new int [RowNum];

  pProcInd = new int [ProcNum];
  pProcNum = new int [ProcNum];

  for (int i=0; i<RowNum; i++)
    pProcPivotIter[i] = -1;

  if (ProcRank == 0) {
    pMatrix = new double [Size*Size];
    pVector = new double [Size];
    pResult = new double [Size];
      // DummyDataInitialization (pMatrix, pVector, Size);
    RandomDataInitialization(pMatrix, pVector, Size);
  }
}

// Function for the data distribution among the processes
void DataDistribution(double* pMatrix, double* pProcRows, double* pVector,
  double* pProcVector, int Size, int RowNum) {

  int *pSendNum;     // Number of the elements sent to the process
```

```
  int *pSendInd;     // Index of the first data element sent
                     // to the process
  int RestRows=Size; // Number of rows, that have not been
                     // distributed yet
  int i;             // Loop variable

  // Alloc memory for temporary objects
  pSendInd = new int [ProcNum];
  pSendNum = new int [ProcNum];

  // Define the disposition of the matrix rows for the current process
  RowNum = (Size/ProcNum);
  pSendNum[0] = RowNum*Size;
  pSendInd[0] = 0;
  for (i=1; i<ProcNum; i++) {
    RestRows -= RowNum;
    RowNum = RestRows/(ProcNum-i);
    pSendNum[i] = RowNum*Size;
    pSendInd[i] = pSendInd[i-1]+pSendNum[i-1];
  }

  // Scatter the rows
  MPI_Scatterv(pMatrix, pSendNum, pSendInd, MPI_DOUBLE, pProcRows,
    pSendNum[ProcRank], MPI_DOUBLE, 0, MPI_COMM_WORLD);

  // Define the disposition of the matrix rows for current process
  RestRows = Size;
  pProcInd[0] = 0;
  pProcNum[0] = Size/ProcNum;
  for (i=1; i<ProcNum; i++) {
    RestRows -= pProcNum[i-1];
    pProcNum[i] = RestRows/(ProcNum-i);
    pProcInd[i] = pProcInd[i-1]+pProcNum[i-1];
  }

  MPI_Scatterv(pVector, pProcNum, pProcInd, MPI_DOUBLE, pProcVector,
    pProcNum[ProcRank], MPI_DOUBLE, 0, MPI_COMM_WORLD);

  // Free the memory
  delete [] pSendNum;
  delete [] pSendInd;
}

// Function for gathering the result vector
void ResultCollection(double* pProcResult, double* pResult) {
   //Gather the whole result vector on every processor
  MPI_Gatherv(pProcResult, pProcNum[ProcRank], MPI_DOUBLE, pResult,
    pProcNum, pProcInd, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}

// Function for formatted matrix output
void PrintMatrix (double* pMatrix, int RowCount, int ColCount) {
  int i, j; // Loop variables
  for (i=0; i<RowCount; i++) {
    for (j=0; j<ColCount; j++)
        printf("%7.4f ", pMatrix[i*ColCount+j]);
      printf("\n");
  }
}

// Function for formatted vector output
void PrintVector (double* pVector, int Size) {
  int i;
```

```
  for (i=0; i<Size; i++)
    printf("%7.4f ", pVector[i]);
}

// Function for formatted result vector output
void PrintResultVector (double* pResult, int Size) {
  int i;
  for (i=0; i<Size; i++)
        printf("%7.4f ", pResult[pParallelPivotPos[i]]);
}

// Fuction for the column elimination
void ParallelEliminateColumns(double* pProcRows, double* pProcVector,
  double* pPivotRow, int Size, int RowNum, int Iter) {
  double multiplier;
  for (int i=0; i<RowNum; i++) {
    if (pProcPivotIter[i] == -1) {
      multiplier = pProcRows[i*Size+Iter] / pPivotRow[Iter];
      for (int j=Iter; j<Size; j++) {
        pProcRows[i*Size + j] -= pPivotRow[j]*multiplier;
      }
      pProcVector[i] -= pPivotRow[Size]*multiplier;
    }
  }
}

// Function for the Gausian elimination
void ParallelGaussianElimination (double* pProcRows, double* pProcVector,
  int Size, int RowNum) {
  double MaxValue;   // Value of the pivot element of the process
  int    PivotPos;   // Position of the pivot row in the process stripe
  // Structure for the pivot row selection
  struct { double MaxValue; int ProcRank; } ProcPivot, Pivot;

  // pPivotRow is used for storing the pivot row and the corresponding
  // element of the vector b
  double* pPivotRow = new double [Size+1];

  // The iterations of the Gaussian elimination stage
  for (int i=0; i<Size; i++) {

    // Calculating the local pivot row
    double MaxValue = 0;
      for (int j=0; j<RowNum; j++) {
      if ((pProcPivotIter[j] == -1) &&
          (MaxValue < fabs(pProcRows[j*Size+i]))) {
        MaxValue = fabs(pProcRows[j*Size+i]);
        PivotPos = j;
        }
    }
    ProcPivot.MaxValue = MaxValue;
    ProcPivot.ProcRank = ProcRank;

    // Find the pivot process (process with the maximum value of MaxValue)
    MPI_Allreduce(&ProcPivot, &Pivot, 1, MPI_DOUBLE_INT, MPI_MAXLOC,
                  MPI_COMM_WORLD);

    // Broadcasting the pivot row
    if ( ProcRank == Pivot.ProcRank ){
      pProcPivotIter[PivotPos]= i; //iteration number
      pParallelPivotPos[i]= pProcInd[ProcRank] + PivotPos;
      }
    MPI_Bcast(&pParallelPivotPos[i], 1, MPI_INT, Pivot.ProcRank,
```

```
      MPI_COMM_WORLD);

    if ( ProcRank == Pivot.ProcRank ){
      // Fill the pivot row
      for (int j=0; j<Size; j++) {
        pPivotRow[j] = pProcRows[PivotPos*Size + j];
      }
      pPivotRow[Size] = pProcVector[PivotPos];
    }
    MPI_Bcast(pPivotRow, Size+1, MPI_DOUBLE, Pivot.ProcRank,
      MPI_COMM_WORLD);

    ParallelEliminateColumns(pProcRows, pProcVector, pPivotRow, Size,
      RowNum, i);
  }
}
// Function for finding the pivot row of the back substitution
void FindBackPivotRow (int RowIndex, int Size, int &IterProcRank,
  int &IterPivotPos) {
  for (int i=0; i<ProcNum-1; i++) {
    if ((pProcInd[i]<=RowIndex) && (RowIndex<pProcInd[i+1]))
        IterProcRank = i;
  }
  if (RowIndex >= pProcInd[ProcNum-1])
    IterProcRank = ProcNum-1;
  IterPivotPos = RowIndex - pProcInd[IterProcRank];
}

// Function for the back substitution
void ParallelBackSubstitution (double* pProcRows, double* pProcVector,
  double* pProcResult, int Size, int RowNum) {
  int IterProcRank;    // Rank of the process with the current pivot row
  int IterPivotPos;    // Position of the pivot row of the process
  double IterResult;   // Calculated value of the current unknown
  double val;

  // Iterations of the back substitution stage
  for (int i=Size-1; i>=0; i--) {

    // Calculating the rank of the process, which holds the pivot row
      FindBackPivotRow(pParallelPivotPos[i], Size, IterProcRank,
        IterPivotPos);

    // Calculating the unknown
    if (ProcRank == IterProcRank) {
      IterResult =
        pProcVector[IterPivotPos]/pProcRows[IterPivotPos*Size+i];
      pProcResult[IterPivotPos] = IterResult;
    }
    // Broadcasting the value of the current unknown
    MPI_Bcast(&IterResult, 1, MPI_DOUBLE, IterProcRank, MPI_COMM_WORLD);

    // Updating the values of the vector b
    for (int j=0; j<RowNum; j++)
      if ( pProcPivotIter[j] < i ) {
        val = pProcRows[j*Size + i] * IterResult;
        pProcVector[j]=pProcVector[j] - val;
    }
  }
}

void TestDistribution(double* pMatrix, double* pVector, double* pProcRows,
  double* pProcVector, int Size, int RowNum) {
```

```
  if (ProcRank == 0) {
    printf("Initial Matrix: \n");
    PrintMatrix(pMatrix, Size, Size);
    printf("Initial Vector: \n");
    PrintVector(pVector, Size);
  }
  MPI_Barrier(MPI_COMM_WORLD);
  for (int i=0; i<ProcNum; i++) {
    if (ProcRank == i) {
      printf("\nProcRank = %d \n", ProcRank);
      printf(" Matrix Stripe:\n");
      PrintMatrix(pProcRows, RowNum, Size);
      printf(" Vector: \n");
      PrintVector(pProcVector, RowNum);
    }
    MPI_Barrier(MPI_COMM_WORLD);
  }
}

// Function for the execution of the parallel Gauss algorithm
void ParallelResultCalculation(double* pProcRows, double* pProcVector,
  double* pProcResult, int Size, int RowNum) {
  ParallelGaussianElimination (pProcRows, pProcVector, Size, RowNum);
  ParallelBackSubstitution (pProcRows, pProcVector, pProcResult, Size,
    RowNum);
}

// Function for computational process termination
void ProcessTermination (double* pMatrix, double* pVector, double* pResult,
  double* pProcRows, double* pProcVector, double* pProcResult) {
  if (ProcRank == 0) {
    delete [] pMatrix;
    delete [] pVector;
    delete [] pResult;
  }
  delete [] pProcRows;
  delete [] pProcVector;
  delete [] pProcResult;

  delete [] pParallelPivotPos;
  delete [] pProcPivotIter;

  delete [] pProcInd;
  delete [] pProcNum;
}


// Function for testing the result
void TestResult(double* pMatrix, double* pVector, double* pResult, int
Size) {
  /* Buffer for storing the vector, that is a result of multiplication
     of the linear system matrix by the vector of unknowns */
  double* pRightPartVector;
  // Flag, that shows wheather the right parts vectors are identical or not
  int equal = 0;
  double Accuracy = 1.e-6; // Comparison accuracy

  if (ProcRank == 0) {
    pRightPartVector = new double [Size];
      for (int i=0; i<Size; i++) {
        pRightPartVector[i] = 0;
        for (int j=0; j<Size; j++) {
          pRightPartVector[i] +=
```

```
            pMatrix[i*Size+j]*pResult[pParallelPivotPos[j]];
        }
      }

    for (int i=0; i<Size; i++) {
        if (fabs(pRightPartVector[i]-pVector[i]) > Accuracy))
        equal = 1;
    }
    if (equal == 1)
      printf("The result of the parallel Gauss algorithm is NOT correct."
             "Check your code.");
    else
      printf("The result of the parallel Gauss algorithm is correct.");
    delete [] pRightPartVector;
  }
}

void main(int argc, char* argv[]) {
  double* pMatrix;        // Matrix of the linear system
  double* pVector;        // Right parts of the linear system
  double* pResult;        // Result vector
  double *pProcRows;      // Rows of the matrix A
  double *pProcVector;    // Block of the vector b
  double *pProcResult;    // Block of the vector x
  int     Size;           // Size of the matrix and vectors
  int     RowNum;         // Number of  the matrix rows
  double start, finish, duration;

  setvbuf(stdout, 0, _IONBF, 0);

  MPI_Init ( &argc, &argv );
  MPI_Comm_rank ( MPI_COMM_WORLD, &ProcRank );
  MPI_Comm_size ( MPI_COMM_WORLD, &ProcNum );

  if (ProcRank == 0)
        printf("Parallel Gauss algorithm for solving linear systems\n");

  // Memory allocation and data initialization
  ProcessInitialization(pMatrix, pVector, pResult,
    pProcRows, pProcVector, pProcResult, Size, RowNum);
  // The execution of the parallel Gauss algorithm
  start = MPI_Wtime();

  DataDistribution(pMatrix, pProcRows, pVector, pProcVector, Size, RowNum);

  ParallelResultCalculation(pProcRows, pProcVector, pProcResult, Size,
    RowNum);
  TestDistribution(pMatrix, pVector, pProcRows, pProcVector, Size, RowNum);

  ResultCollection(pProcResult, pResult);

  finish = MPI_Wtime();
  duration = finish-start;

  if (ProcRank == 0) {
    // Printing the result vector
    printf ("\n Result Vector: \n");
    PrintResultVector(pResult, Size);
  }
  TestResult(pMatrix, pVector, pResult, Size);

  // Printing the time spent by Gauss algorithm
  if (ProcRank == 0)
```

```
    printf("\n Time of execution: %f\n", duration);

  // Computational process termination
  ProcessTermination(pMatrix, pVector, pResult, pProcRows, pProcVector,
    pProcResult);
  MPI_Finalize();
}
```