МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«САМАРСКИЙ ГОСУДАРСТВЕННЫЙ АЭРОКОСМИЧЕСКИЙ
УНИВЕРСИТЕТ имени академика С.П. КОРОЛЁВА»
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

А.А. БЕЛОУСОВ

УЧЕБНОЕ ПОСОБИЕ ПО МЕТОДАМ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ

Часть I

САМАРА
2013

УДК 004.432
ББК 32.973.26-018.1

Белоусов, А.А. Учебное пособие по методам параллельных вычислений Часть I[Текст] / А.А. Белоусов, – Самара: Издательство СНЦ РАН, 2013. – 200 с.

ISBN 978-5-906605-02-3

Учебное пособие посвящено изучению метододов параллельных вычислений и их реализации в кластерных системах с разделяемой памятью. При этом изложение построено не от теории к задачам, а от задач к теории и приёмам, которые требуются для решения задач.

Каждый из 6 разделов пособия направлен на изучение некоторой темы и состоит из задания, кода решения и комментария к решению, подробно объясняющего выбранные способ и средства решения. Решение каждой следующей задачи при этом основывается на приемах использовавшихся при решении предыдущих, что позволяет в ходе изучения постепенно перейти от основ построения параллельных алгоритмов к достаточно сложным алгоритмам и их реализациям с использованием стандарта MPI .

Пособие ориентировано на студентов, обучающихся по учебным планам бакалавров и специалистов, и может использоваться в качестве учебника по методам параллельных вычислений.

Рецензенты: д.т.н., профессор Клебанов Я.М.
к.т.н., доцент Симонова Е.В.

ISBN 978-5-906605-02-3

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

Настоящее пособие имеет целью показать как средства стандарта MPI для разработки параллельных алгоритмов в системах с разделяемой памятью применяются для решения конкретных задач и как программист должен выбирать и использовать эти средства при проектировании параллельных алгоритмов. Изучение данного пособия позволит сформировать единую картину в восприятии читателя из средств стандарта MPI и возможностями по их применению в конкретных случаях.

В пособие вошли 6 задач, которые более пяти лет предлагались студентам, изучавшим предметы «Методы параллельных вычислений», «Параллельное программирование на основе MPI» на факультете информатики в Самарском государственном аэрокосмическом университете. Задачи расположены в порядке изучения возможностей стандарта MPI и приемов, применяемых в построении параллельных алгоритмов, при этом каждая следующая задача опирается на решения предыдущих. Таким образом читатель, начиная практически с азов методов и функций MPI, знакомится со всё более сложными средствами и конструкциями.

Каждая задача посвящена отдельному алгоритму, который необходимо реализовать в системе с разделяемой памятью и содержит, собственно, описание последовательного алгоритма, вариантов его распараллеливания, а так же упражнения, в ходе выполнения которых, студент сможет реализовать в программном коде требуемые алгоритмы. Накопленный опыт работы со студентами позволил сосредоточиться на наиболее трудных для понимания местах и типовых ошибках. После каждой задачи приводится список вопросов, позволяющий проконтролировать усвоение теоретического материала по данной теме.

Пособие разработано при поддержке РФФИ (грант 12-07-31193 мол_а).

ЗАДАЧА 1: ПАРАЛЛЕЛЬНЫЕ АЛГОРИТМЫ МАТРИЧНО-ВЕКТОРНОГО УМНОЖЕНИЯ

Матрицы и матричные операции широко используются при математическом моделировании самых разнообразных процессов, явлений и систем. Матричные вычисления составляют основу многих научных и инженерных расчетов – среди областей приложений могут быть указаны вычислительная математика, физика, экономика и др.

Являясь вычислительно-трудоемкими, матричные вычисления представляют собой классическую область применения параллельных вычислений. С одной стороны, использование высокопроизводительных многопроцессорных систем позволяет существенно повысить сложность решаемых задач. С другой стороны, в силу своей достаточно простой формулировки матричные операции предоставляют прекрасную возможность для демонстрации многих приемов и методов параллельного программирования.

Обзор задачи

Целью данной задачи является разработка параллельной программы, которая выполняет умножение матрицы на вектор.

- Упражнение 1 – Постановка задачи матрично-векторного умножения
- Упражнение 2 – Реализация последовательного алгоритма умножения матрицы на вектор
- Упражнение 3 – Разработка параллельного алгоритма умножения матрицы на вектор
- Упражнение 4 - Реализация параллельного алгоритма матрично-векторного умножения

При выполнении задачи предполагается знание раздела "Параллельное программирование на основе MPI", "Принципы разработки параллельных методов" и "Параллельные методы умножения матрицы на вектор".

Упражнение 1 – Постановка задачи матрично-векторного умножения

В результате умножения матрицы A размерности $m \times n$ и вектора b, состоящего из n элементов, получается вектор c размера

m, каждый i-ый элемент которого есть результат скалярного умножения i-й строки матрицы A (обозначим эту строчку a_i) и вектора b (см. рис. 1.1):

$$c_i = (a_i, b) = \sum_{j=1}^{n} a_{ij} b_j, 1 \le i \le m \qquad (1.1)$$

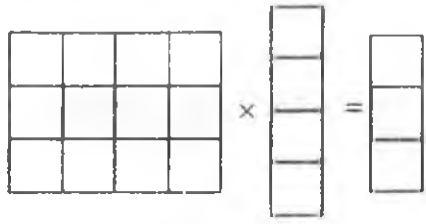

Рис. 1.1. Элемент результирующего вектора – это результат скалярного умножения строки матрицы на вектор

Так, например, при умножении матрицы, состоящей из 3 строк и 4 столбцов на вектор из 4 элементов, получается вектор размера 3:

$$\begin{pmatrix} 3 & 2 & 0 & -1 \\ 5 & -2 & 1 & 1 \\ 1 & 0 & -1 & -1 \end{pmatrix} \times \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix} = \begin{pmatrix} 3 \\ 8 \\ -6 \end{pmatrix}$$

Рис. 1.2. Умножение матрицы на вектор

Тем самым, получение результирующего вектора c предполагает повторение m однотипных операций по умножению строк матрицы A и вектора b. Каждая такая операция включает умножение элементов строки матрицы и вектора b и последующее суммирование полученных произведений.

Псевдокод для представленного алгоритма умножения матрицы на вектор может выглядеть следующим образом:

```
// Serial algorithm of matrix-vector multiplication
for (i = 0; i < m; i++){
 c[i] = 0;
 for (j = 0; j < n; j++){
 c[i] += A[i][j]*b[j]
 }
}
```

Упражнение 2 – Реализация последовательного алгоритма умножения матрицы на вектор

При выполнении этого упражнения необходимо реализовать последовательный алгоритм матрично- векторного умножения. Начальный вариант будущей программы представлен в проекте *SerailMatrixVecorMult*, который содержит часть исходного кода и и в котором заданы необходимые параметры проекта. В ходе выполнения упражнения необходимо дополнить имеющийся вариант программы операциями ввода размера объектов, инициализации матрицы и вектора, умножения матрицы на вектор и вывода результатов.

Задание 1 – Открытие проекта SerialMatrixVectorMult

Откройте проект **SerialMatrixVector**, 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\Serial Matrix Vector**,
- Дважды щелкните на файле **SerialMatrixVector.sln** или выбрав файл выполните команду **Open**.

После открытия проекта в окне Solution Explorer (Ctrl+Alt+L) дважды щелкните на файле исходного кода **SerialMV.cpp**, как это показано на рис. 1.3. После этих действий код, который предстоит в дальнейшем расширить будет открыт в рабочей области Visual Studio.

Рис. 1.3. Открытие файла SerialMV.cpp

В файле SerialMV.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 и вектор, которые участвуют в матрично-векторном умножении в качестве аргументов. Третья переменная *pResult* – вектор, который должен быть получен в результате матрично-векторного умножения. Переменная *Size* определяет размер матриц и векторов (предполагаем, что матрица *pMatrix* квадратная, имеет размерность *Size*×*Size*, умножается на вектор из *Size* элементов). Далее объявлены переменные циклов.

```
    double* pMatrix;  // The first argument - initial ma-
trix
    double* pVector;  // The second argument - initial
vector
    double* pResult;  // Result vector for matrix-vector
multiplication
    int Size;  // Sizes of initial matrix and vector
```

Заметим, что для хранения матрицы *pMatrix* используется одномерный массив, в котором матрица хранится 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 matrix-vector multiplication pro-
gram\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 matrix-vector multiplication program". Для того, чтобы завершить выполнение программы, нажмите любую клавишу.

Задание 2 – Ввод размеров объектов

Для задания исходных данных последовательного алгоритма умножения матрицы на вектор реализуем функцию *ProcessInitialization*. Эта функция предназначена для определения размеров объектов, выделения памяти для всех объектов, участвующих в умножении (исходных матрицы *pMatrix* и вектора *pVector*, и результата умножения *pResult*), а также для задания значений элементов исходных матрицы и вектора. Значит, функция должна иметь следующий интерфейс:

```
    // Function for memory allocation and definition of ob-
ject's elements void
    ProcessInitialization (double* &pMatrix, double*
&pVector,
     double* &pResult, int &Size);
```

На самом первом этапе необходимо определить размеры объектов (задать значение

Переменной *Size*). В тело функции *ProcessInitialization* добавьте выделенный фрагмент кода:

```
    // Function for memory allocation and definition of ob-
ject's elements void
    ProcessInitialization (double* &pMatrix, double*
&pVector,
```

```
    double* &pResult, int &Size) {
    // Setting the size of initial matrix and vector
    printf("\nEnter size of the initial objects: ");
    scanf("%d", &Size);
    printf("\nChosen objects size = %d", Size);
}
```

Пользователю предоставляется возможность ввести размер объектов (матрицы и вектора), который затем считывается из стандартного потока ввода *stdin* и сохраняется в целочисленной переменной *Size*. Далее печатается значение переменной *Size* (рис. 1.4).

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

```
void main() {
    double* pMatrix;      // The first argument - initial
matrix
    double* pVector;      // The second argument - initial
vector
    double* pResult;      // Result vector for matrix-
vector multiplication
    int Size;             // Sizes of initial matrix and
vector
    time_t start, finish;
    double duration;
    printf ("Serial matrix-vector multiplication pro-
gram\n");
    ProcessInitialization(pMatrix, pVector, pResult,
Size);
    getch();
}
```

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

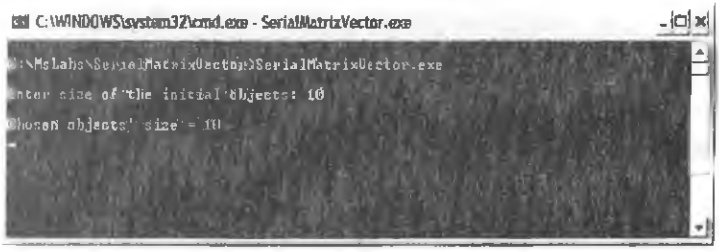

Рис. 1.4. Задание размера объектов

Теперь обратимся к вопросу контроля правильности ввода. Так, например, если в качестве размера объектов пользователь попытается ввести неположительное число, приложение должно либо завершить выполнение, либо продолжать запрашивать размер объектов до тех пор, пока не будет введеноположительное число. Реализуем второй вариант поведения, для этого тот фрагмент кода, который производит ввод размера объектов, поместим в цикл с постусловием:

```
// Setting the size of initial matrix and vector
do {
 printf("\nEnter size of the initial objects: ");
 scanf("%d", &Size);
 printf("\nChosen objects size = %d", 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 definition of ob-
jects' elements
  void ProcessInitialization (double* &pMatrix, double*
&pVector,
    double* &pResult, int Size) {
    // Setting the size of initial matrix and vector
    do {
    <..>
    }
    while (Size <= 0);

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

Далее необходимо задать значения всех элементов исходных объектов: матрицы *pMatrix* и вектора *pVector*. Для выполнения этих действий реализуем еще одну функцию *DummyDataInitialization*.

Интерфейс и реализация этой функции представлены ниже:

```
// 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] = 1;
 for (j=0; j<Size; j++)
 pMatrix[i*Size+j] = i;
 }
}
```

Как видно из представленного фрагмента кода, данная функция осуществляет задание элементов матрицы и вектора достаточно простым образом: значение элемента матрицы совпадает с номером строки, в которой он расположен, а все элементы вектора равны 1. То есть в случае, когда пользователь выбрал размер объектов, равный 4, будут определены следующие матрица и вектор:

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

(задание данных при помощи датчика случайных чисел будет рассмотрено в задании 6).

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

```
// Function for memory allocation and definition of ob-
ject's elements void

ProcessInitialization (double* &pMatrix, double*
&pVector,

 double* &pResult, int Size) {
 // Setting the size of initial matrix and vector
 do {
 <..>
```

```
  }
  while (Size <= 0);

  // Memory allocation
  <...>

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

Реализуем еще две функции, которые помогут контролировать ввод данных. Это функции форматированного вывода объектов: *PrintMatrix* и *PrintVector*. В качестве аргументов в функцию форматированной печати матрицы *PrintMatrix* передается указатель на одномерный массив, где эта матрица хранится построчно, а также размеры матрицы по вертикали (количество строк *RowCount*) и горизонтали (количество столбцов *ColCount*). Для форматированной печати вектора при помощи функции *PrintVector*, необходимо сообщить функции указатель на вектор, а также количество элементов в нем.

```
// 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]);
  printf("\n");
}
```

Добавим вызов этих функций в основную функцию приложения:

```
  // Memory allocation and definition of objects' ele-
ments
  ProcessInitialization(pMatrix,    pVector,    pResult,
```

```
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. 1.5). Выполните несколько запусков приложения, задавайте различные размеры объектов.

Рис. 1.5. Результат работы программы при завершении задания 3

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 результата умножения *pResult*. Следовательно, эти объекты необходимо передать в функцию *ProcessTermination* в качестве аргументов:

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

Вызов функции *ProcessTermination* необходимо выполнить перед завершением той части программы, которая выполняет умножение матрицы на вектор:

```
  // Memory allocation and definition of objects' ele-
ments
  ProcessInitialization(pMatrix, pVector, pResult, Size);

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

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

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

Задание 5 – Реализация умножения матрицы на вектор

Выполним теперь разработку основной вычислительной части программы. Для выполнения умножения матрицы на вектор реализуем функцию *ResultCalculation*, которая принимает на вход исходные матрицу *pMatrix* и вектор *pVector*, размеры этих объектов *Size*, а также указатель на вектор в памяти, где должен быть сохранен результат *pResult*.

В соответствии с алгоритмом, изложенным в упражнении 1, код этой функции должен быть следующий:

```
  // Function for matrix-vector multiplication
  void ResultCalculation(double* pMatrix, double* pVec-
tor, double* pResult,
    int Size) {
    int i, j;   // Loop variables
    for (i=0; i<Size; i++) {
    pResult[i] = 0;
    for (j=0; j<Size; j++)
    pResult[i] += pMatrix[i*Size+j]*pVector[j];
    }
  }
```

Выполним вызов функции вычисления умножения матрицы на вектор из основной программы. Для контроля правильности выполнения умножения распечатаем результирующий вектор:

```
    // Memory allocation and definition of objects' ele-
ments
    ProcessInitialization(pMatrix,   pVector,   pResult,
```

```
Size);

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

    // Matrix-vector multiplication
    ResultCalculation(pMatrix, pVector, pResult, Size);

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

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

Скомпилируйте и запустите приложение. Проанализируйте результат работы алгоритма умножения матрицы на вектор. Если алгоритм реализован правильно, то результирующий вектор должен иметь следующую структуру: *i*-ый элемент результирующего вектора равен произведению размера вектора *Size* на номер элемента *i*. Так, если размер объектов *Size* равен 4, результирующий вектор *pResult* должен быть таким: *pResult* = (0, 4, 8, 12). Проведите несколько вычислительных экспериментов, изменяя размеры объектов.

Рис. 1.6. Результат выполнения матрично-векторного умножения

Задание 6 – 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 объектов. Задавать элементы больших матриц и векторов будем при помощи датчика случайных чисел. Для этого реализуем еще одну функцию задания элементов *RandomDataInitialization* (датчик случайных чисел инициализируется текущим значением времени):

```
   // Function for random initialization of objects' ele-
ments
   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++)
    pMatrix[i*Size+j] = rand()/double(1000);
    }
   }
```

Будем вызывать эту функцию вместо ранее разработанной функции *DummyDataInitialization*, которая генерировала такие данные, что можно было легко проверить правильность работы алгоритма:

```
   // Function for memory allocation and definition of ob-
jects' elements void
   ProcessInitialization   (double*   &pMatrix,   double*
&pVector,
    double* &pResult, int Size) {
    // Size of initial matrix and vector definition
    <...>

    // Memory allocation
    <...>

    // Random definition of objects' elements
    RandomDataInitialization(pMatrix, pVector, Size);
   }
```

Скомпилируйте и запустите приложение. Убедитесь в том, что данные генерируются случайным образом.

Для определения времени добавьте в получившуюся программу вызовы функций, позволяющие узнать время работы программы или её части. Мы будем пользоваться функцией:

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

Эта функция возвращает количество тактов (*тиков*) процессора, прошедших с момента старта системы. Следовательно, вызвав эту функцию два раза – до и после исследуемого фрагмента можно вычислить время его работы. Например, этот фрагмент вычислит время *duration* работы функции f().

```
time_t t1, t2;
t1 = clock();
f();
t2 = clock();
double duration = (t2-t1)/double(CLOCKS_PER_SEC);
```

Добавим в программный код вычисление и вывод времени непосредственного выполнения умножения матрицы на вектор, для этого поставим замеры времени до и после вызова функции *ResultCalculation*:

```
// Matrix-vector multiplication
start = clock();
ResultCalculation(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 time spent by matrix-vector multiplication
printf("\n Time of execution: %f", duration);
```

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

Номер теста	Размер матрицы	Время работы (сек)
Тест №1	10	
Тест №2	100	
Тест №3	1000	
Тест №4	2000	

Тест №5	3000	
Тест №6	4000	
Тест №7	5000	
Тест №8	6000	
Тест №9	7000	
Тест №10	8000	
Тест №11	9000	
Тест №12	10 000	

Согласно алгоритму вычисления произведения матрицы и вектора, изложенному в упражнении 1, получение результирующего вектора предполагает повторение *Size* однотипных операций по умножению строк матрицы *pMatrix* и вектора *pVector*. Каждая такая операция включает умножение элементов строки матрицы и вектора (*Size* операций) и последующее суммирование полученных произведений (*Size-1* операций). Общее количество необходимых скалярных операций есть величина

$$N = Size * (2 * Size - 1) \quad (1.2)$$

Для того, чтобы оценить время выполнения параллельного алгоритма, необходимо знать длительность выполнения одной операции τ. Итак, чтобы вычислить время выполнения алгоритма, нужно умножить число выполняемых операций на время выполнения одной операции:

$$T_1 = N * \tau = Size(2 * Size - 1) * \tau \quad (1.3)$$

Заполним таблицу сравнения реального времени выполнения со временем, которое может быть получено по формуле (1.3). Для вычисления времени выполнения одной операции применим следующую методику: выберем один из экспериментов как образец. Пусть, например, в качестве образца выступает эксперимент по умножению матрицы и вектора размером 5000. Время выполнения этого эксперимента поделим на число выполненных операций (число операций может быть вычислено по формуле (1.2)). Таким образом, вычислим время выполнения одной операции. Далее, используя это значение, вычислим теоретическое время выполнения для всех оставшихся экспериментов. Результаты занесите в таблицу:

Время выполнения одной операции τ (сек):			
Номер теста	Размер матрицы	Время работы (сек)	Теоретическое время (сек)
Тест №1	10		
Тест №2	100		
Тест №3	1000		
Тест №4	2000		
Тест №5	3000		
Тест №6	4000		
Тест №7	5000		
Тест №8	6000		
Тест №9	7000		
Тест №10	8000		
Тест №11	9000		
Тест №12	10 000		

Заметим, что время выполнения одной операции, вообще говоря, зависит от размера объектов, которые участвуют в умножении. Такая зависимость объясняется особенностями архитектуры компьютера. Если объекты очень небольшие, то они полностью могут быть помещены в кэш-память процессора, доступ к этой памяти осуществляется очень быстро. Если алгоритм работает с объектами среднего размера, такими, которые полностью могут быть помещены в оперативную память, но не могут быть помещены в кэш, то время выполнения одной операции в этом случае будет несколько больше, так как для обращения к ячейке оперативной памяти требуется больше времени, чем для обращения к кэш. Если же объекты настолько велики, что не могут быть помещены в оперативную память, то включается механизм работы с файлами подкачки (swap file), объекты сохраняются на жестком диске компьютера, время чтения и записи на жесткий диск существенно превышает время записи в ячейку оперативной памяти. Таким образом, при выборе эксперимента для образца (такого эксперимента, для которого будет вычисляться время выполнения одной операции), следует ориентиро-

ваться на некоторую среднюю ситуацию. Именно поэтому нами в качестве образца был выбран эксперимент по умножению матрицы и вектора размером 5000.

Упражнение 3 – Разработка параллельного алгоритма умножения матрицы на вектор

Принципы распараллеливания

Разработка алгоритмов (а в особенности методов параллельных вычислений) для решения сложных научно-технических задач часто представляет собой значительную проблему. Здесь же мы будем полагать, что вычислительная схема решения нашей задачи умножения матрицы на вектор уже известна. Действия для определения эффективных способов организации параллельных вычислений могут состоять в следующем:

- Выполнить анализ имеющейся вычислительной схемы и осуществить ее разделение (*декомпозицию*) на части (*подзадачи*), которые могут быть реализованы в значительной степени независимо друг от друга,
- Выделить для сформированного набора подзадач *информационные взаимодействия*, которые должны осуществляться в ходе решения исходной поставленной задачи,
- Определить необходимую (или доступную) для решения задачи *вычислительную систему* и выполнить *распределение* имеющегося набора подзадач между процессорами системы.

Такие этапы разработки параллельных алгоритмов впервые были предложены Фостером (I. Foster).

При самом общем рассмотрении понятно, что объем вычислений для каждого используемого процессора должен быть примерно одинаков – это позволит обеспечить равномерную вычислительную загрузку (*балансировку*) процессоров. Кроме того, также понятно, что распределение подзадач между процессорами должно быть выполнено таким образом, чтобы наличие информационных связей (*коммуникационных взаимодействий*) между подзадачами было минимальным.

Определение подзадач

Для многих методов матричных вычислений характерным является повторение одних и тех же вычислительных действий для разных элементов матриц. Данный момент свидетельствует о наличии *параллелизма по данным* при выполнении матричных расчетов и, как результат, распараллеливание матричных операций сводится в большинстве случаев к разделению обрабатываемых матриц между процессорами используемой вычислительной системы. Выбор способа разделения матриц приводит к определению конкретного метода параллельных вычислений; существование разных схем распределения данных порождает целый ряд *параллельных алгоритмов матричных вычислений*.

Дадим кратко общую характеристику распределения данных для матричных алгоритмов – более подробно данный материал содержится в разделе 7 учебного курса. Наиболее общие и широко используемые способы разделения матриц состоят в разбиении данных на *полосы* (по вертикали или горизонтали) или на прямоугольные фрагменты (*блоки*).

1. Ленточное разбиение матрицы. При *ленточном* (*block-striped*) разбиении каждому процессору выделяется то или иное подмножество строк (*rowwise* или *горизонтальное разбиение*) или столбцов (*columnwise* или *вертикальное разбиение*) матрицы (рис. 1.7а и 1.7б). Разделение строк и столбцов на полосы в большинстве случаев происходит на *непрерывной* (*последовательной*) основе. При таком подходе для горизонтального разбиения по строкам, например, матрица A представляется в виде:

$$A=\left(A_0, A_1, \ldots, A_{p-1}\right)^T, A_i=\left(a_{i_0}, a_{i_1}, \ldots, a_{i_{k-1}}\right), i_j = ik + j, 0 \le j < k, k = m / p,$$

где $a_i = (a_{i1}, a_{i2}, \ldots a_{in})$, $0 \le i < m$, есть i-я строка матрицы A (предполагается, что количество строк m кратно числу процессоров p, т.е. $m = k \cdot p$). Во всех алгоритмах матричного умножения и умножения матрицы на вектор, которые будут рассмотрены нами в этом и следующем разделах, используется разделение данных на непрерывной основе.

Другой возможный подход к формированию полос состоит в применении той или иной схемы *чередования* (*цикличности*) строк или столбцов. Как правило, для чередования используется число процессоров p – в этом случае при горизонтальном разби-

нии матрица A принимает вид

$$A=\left(A_0,A_1,\ldots,A_{p-1}\right)^T,\ A_i=\left(a_{i_0},a_{i_1},\ldots,a_{i_{k-1}}\right),i_j=i+jp,0\le j<k,k=m/p.$$

Циклическая схема формирования полос может оказаться полезной для лучшей балансировки вычислительной нагрузки процессоров (например, при решении системы линейных уравнений с использованием метода Гаусса – см. раздел 9 учебного курса).

2. Блочное разбиение матрицы. При *блочном* (*checkerboard block*) разделении матрица делится на прямоугольные наборы элементов – при этом, как правило, используется разделение на непрерывной основе. Пусть количество процессоров составляет p=sq,количество строк матрицы является кратным s,а количество столбцов-кратным q,то есть m=ks и n=lq. Представим исходную матрицу A в виде набора прямоугольных блоков следующим образом (рис. 1.7в):

$$A=\begin{pmatrix} A_{00} & A_{02} & \ldots A_{0q-1} \\ & \ldots & \\ A_{s-11} & A_{s-12} & \ldots A_{s-1q-1} \end{pmatrix},$$

где A_{ij} - блок матрицы, состоящий из элементов:

$$A_{ij}=\begin{pmatrix} a_{i_1j_1} & a_{i_1j_2} & \ldots a_{i_1j_{l-1}} \\ & \ldots & \\ a_{i_{k-1}j_1} & a_{i_{k-1}j_2} & \ldots a_{i_{k-1}j_{l-1}} \end{pmatrix},i_v=ik+v,0\le v<k,k=m/s,j_u=jl+u,0\le u\le l,l=n/q.$$

При таком подходе целесообразно, чтобы вычислительная система имела физическую или, по крайней мере, логическую топологию процессорной решетки из s строк и q столбцов. В этом случае при разделении данных на непрерывной основе процессоры, соседние в структуре решетки, обрабатывают смежные блоки исходной матрицы. Следует отметить, однако, что и для блочной схемы может быть применено циклическое чередование строк и столбцов.

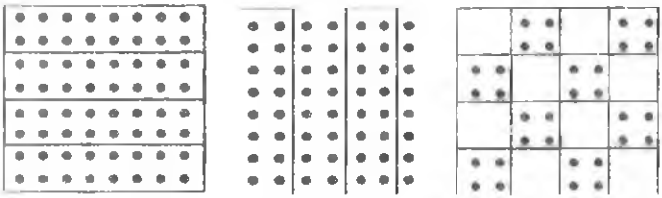

Рис. 1.7. Способы распределения элементов матрицы между процессорами вычислительной системы

Далее в задаче будет рассматриваться алгоритм умножения матрицы на вектор, основанный на представлении матрицы непрерывными наборами (горизонтальными полосами) строк. При таком способе разделения данных в качестве базовой подзадачи может быть выбрана операция скалярного умножения одной строки матрицы на вектор.

Выделение информационных зависимостей

Для выполнения базовой подзадачи скалярного произведения процессор должен содержать соответствующую строку матрицы *pMatrix* и копию вектора *pVector*. После завершения вычислений каждая базовая подзадача определяет один из элементов вектора результата *pResult*.

В общем виде схема информационного взаимодействия подзадач в ходе выполняемых вычислений показана на рис. 1.8.

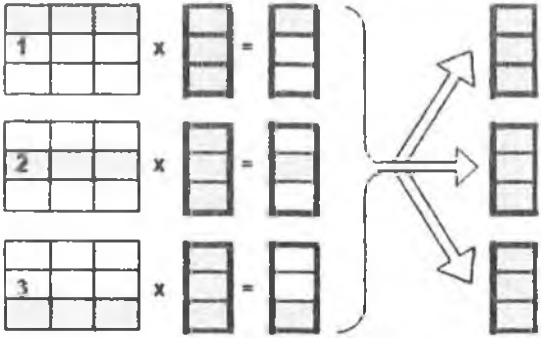

Рис. 1.8. Организация вычислений при выполнении параллельного алгоритма умножения матрицы на вектор, основанного на разделении матрицы по строкам

Для объединения результатов расчета и получения полного вектора *pResult* на каждом из процессоров вычислительной системы необходимо выполнить операцию обобщенного сбора данных, в которой каждый процессор передает свой вычисленный элемент вектора *c* всем остальным процессорам. Этот шаг можно выполнить, например, с использованием функции *MPI_Allgather* из библиотеки MPI (рис. 1.9).

Масштабирование и распределение подзадач по процессорам

В процессе умножения плотной матрицы на вектор количество вычислительных операций для получения скалярного произведения одинаково для всех базовых подзадач. Поэтому в случае, когда число процессоров p меньше числа базовых подзадач m ($p<m$), мы можем объединить базовые подзадачи таким образом, чтобы каждый процессор выполнял несколько таких задач, соответствующих непрерывной последовательности строк матрицы *pMatrix*. В этом случае по окончании вычислений каждая базовая подзадача определяет набор элементов результирующего вектора *pResult*.

Распределение подзадач между процессорами вычислительной системы может быть выполнено произвольным образом.

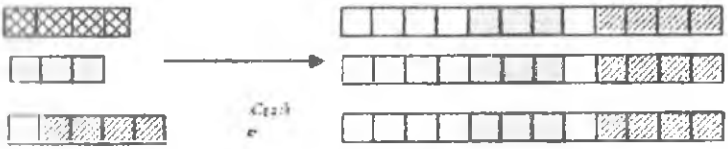

Рис. 1.9. Коллективная коммуникационная операция сбора и обмена данными между всеми процессорами

Упражнение 4 – Реализация параллельного алгоритма матричновекторного умножения

При выполнении этого упражнения Вам будет предложено разработать параллельный алгоритм умножения матрицы на вектор. При работе с этим упражнением Вы

- Познакомитесь с основами MPI, структурой MPI программ и несколькими основными функциями MPI,
- Получите первый опыт разработки параллельных программ.

В параллельных программах, использующих интерфейс передачи сообщений MPI, могут быть выделены следующие основные структурные части:

- Инициализация среды выполнения MPI-программ,
- Основная часть программы, в которой реализуется необходимый алгоритм решения поставленной задачи и в которой осуществляется обмен сообщениями между параллельно выполняемыми частями программы,
- Завершение MPI программы.

Ниже кратко дается характеристика основных понятий MPI.

Понятие параллельной программы

Под *параллельной программой* в рамках MPI понимается множество одновременно выполняемых *процессов*. Процессы могут выполняться на разных процессорах, но на одном процессоре могут располагаться и несколько процессов (в этом случае их исполнение осуществляется в режиме разделения времени). В

предельном случае для выполнения параллельной программы может использоваться один процессор – как правило, такой способ применяется для начальной проверки правильности параллельной программы.

Количество процессов и число используемых процессоров определяется в момент запуска параллельной программы средствами среды исполнения MPI-программ и в ходе вычислений меняться не может (в стандарте MPI-2 предусматривается возможность динамического изменения количества процессов). Все процессы программы последовательно перенумерованы от 0 до $p-1$, где p есть общее количество процессов. Номер процесса именуется *рангом* процесса.

Понятие коммуникатора и группы процессов

Процессы параллельной программы объединяются в *группы*. Под *коммуникатором* в MPI понимается специально создаваемый служебный объект, объединяющий в своем составе группу процессов и ряд дополнительных параметров (*контекст*), используемых при выполнении операций передачи данных.

Как правило, парные операции передачи данных выполняются для процессов, принадлежащих одному и тому же коммуникатору. Коллективные операции применяются одновременно для всех процессов коммуникатора. Как результат, указание используемого коммуникатора является обязательным для операций передачи данных в MPI.

В ходе вычислений могут создаваться новые и удаляться существующие группы процессов и коммуникаторы. Один и тот же процесс может принадлежать разным группам и коммуникаторам. Все имеющиеся в параллельной программе процессы входят в состав создаваемого по умолчанию коммуникатора с идентификатором MPI_COMM_WORLD.

Задание 1 – Открытие проекта ParallelMatrixVectorMult

Откройте проект **ParallelMatrixVectorMult**, 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\ParallelMatrixVectorMult**,
- Дважды щелкните на файле **ParallelMatrixVectorMult.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 **ParallelMV.cpp**, как это показано на рисунке 10. После этих действий код, который вам предстоит модифицировать, будет открыт в рабочей области Visual Studio.

Рис. 1.10. Открытие файла ParallelMV.cpp с использованием Solution Explorer

В файле **ParallelMV.cpp** расположена главная функция (*main*) будущего параллельного приложения, которая содержит объявления необходимых переменных. Также в файле **ParallelMV.cpp** расположены функции, перенесенные сюда из проекта, содержащего последовательный алгоритм умножения матрицы на вектор: *DummyDataInitialization*, *RandomDataInitialization*, *ResultCalculation*, *PrintMatrix* и *PrintVector* (подробно о назначении этих функций рассказывается в упражнении 2 данной задачи). Эти функции можно будет использовать и в параллельной программе. Кроме того, помещены заготовки для функций инициализации процесса вычислений (*ProcessInitialization*) и завершения процесса (*ProcessTermination*).

Скомпилируйте и запустите приложение стандартными сред-

ствами Visual Studio. Убедитесь в том, что в командную консоль выводится приветствие: "Parallel matrix-vector multiplication program".

Задание 2 – Инициализация и завершение параллельной программы

Перед тем, как использовать функции MPI в своем приложении, необходимо добавить заголовочный файл MPI в текст программы. Для приложений, написанных на языке C/C++, заголовочный файл имеет имя *mpi.h*. Этот файл содержит все определения и прототипы функций библиотеки MPI. Добавьте выделенную строку в список подключаемых библиотек в файле исходного кода параллельной программы:

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

В главной функции программы необходимо проинициализировать среду выполнения MPI- программы и завершить ее использование при окончании работы программы. Добавьте выделенный код непосредственно за блоком объявления переменных:

```
void main(int argc, char* argv[]) {
 double* pMatrix;          // The first argument - initial
matrix
 double* pVector;          // The second argument - initial
vector
 double* pResult;          //  Result  vector  for  matrix-
vector multiplication
 int Size;                 // Sizes of initial matrix and
vector
 double Start, Finish, Duration;

 MPI_Init(&argc, &argv);
 printf  ("Parallel  matrix-vector  multiplication  pro-
gram\n");
 MPI_Finalize();
}
```

Функция *MPI_Init* инициализирует среду выполнения MPI-программ. В качестве аргументов этой функции передаются аргументы функции *main*: количество аргументов командной строки *argc* и массив, содержащий эти аргументы, *argv*. Функция

MPI_Init должна вызываться в каждой MPI-программе до вызова любой из функций MPI, в каждой программе функция *MPI_Init* может быть вызвана только один раз.

После выполнения всех необходимых действий, перед завершением выполнения программы, необходимо закрыть среду выполнения MPI-программ. Для завершения среды служит функция *MPI_Finalize*. Добавьте вызов функции *MPI_Finalize* последней строчкой вашей программы.

Теперь обратим внимание на процедуру запуска параллельного приложения. Скомпилируйте параллельное приложение средствами **Visual Studio** (выполните команду **Rebuild Solution** пункта меню **Build**). Для того, чтобы запустить параллельную программу, запустите программу **Command prompt**, выполняя следующие действия:

1. Нажмите кнопку **Пуск**, а затем **Выполнить**,
2. В появившемся диалоговом окне наберите название программы **cmd** (рис. 1.11).

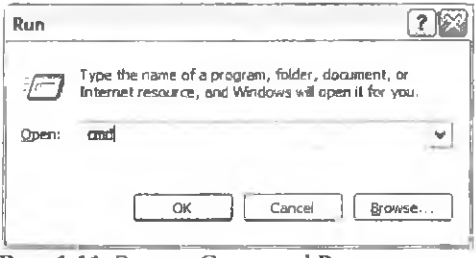

Рис. 1.11. Запуск Command Prompt

В командной строке перейдите в папку, где содержится исполняемый модуль разработанной программы (рис. 1.12):

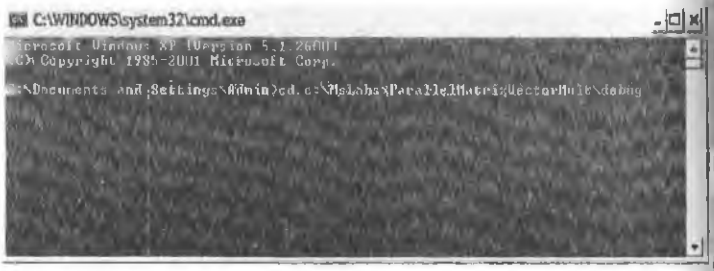

Рис. 1.12. Задание папки, в которой содержится исполняемый модуль параллельной программы

Запуск MPI-программы осуществляется при помощи вызова утилиты **mpiexec**. Формат вызова в общем виде выглядит следующим образом:

`mpiexec -n <кол-во процессов> <имя исполняемого модуля> <аргументы>`.

Для запуска параллельной программы, состоящей из 4 процессов, наберите команду (рис. 1.13):

```
mpiexec -n 4 ParallelMatrixVectorMult.exe
```

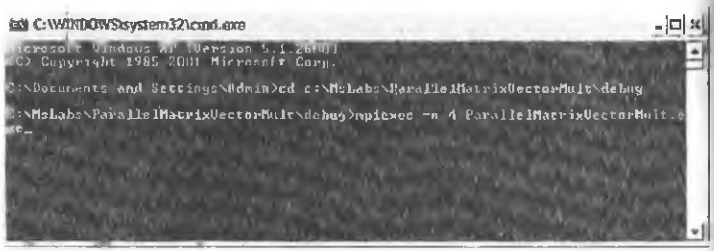

Рис. 1.13. Запуск параллельной программы

Если все было сделано верно, на командную консоль должно быть выведено четыре одинаковых строки приветствия: `"Parallel matrix-vector multiplication program"`, так как печать осуществил каждый процесс параллельной программы (рис. 1.14).

Рис. 1.14. Результат работы первой параллельной программы

Задание 3 – Определение количества процессов

Определение *количества процессов* в выполняемой параллельной программе осуществляется при помощи функции *MPI_Comm_size*. В параметрах функции указывается коммуникатор, для которого определяется количество процессов (тем самым, для определения общего числа процессов, доступных для MPI-программы, необходимо указать коммуникатор *MPI_COMM_WORLD*). Для определения *ранга* процесса в рамках коммуникатора используется функция *MPI_Comm_rank*. (напомним, каждому процессу в рамках коммуникатора соответствует уникальное целое число – ранг). Заведем переменные целого типа для хранения числа доступных процессов *ProcNum* и ранга текущего процесса *ProcRank*. Эти значения обычно используются во всех функциях параллельного приложения. Для того, чтобы эти переменные оказались доступными, объявим *ProcNum* и *ProcRank* как глобальные переменные.

Добавьте выделенные строки в соответствующее место в программном коде:

```
    int ProcNum;        // Number of available processes
    int ProcRank;       // Rank of current process

    void main(int argc, char* argv[]) {
     double* pMatrix;     // The first argument - initial
matrix
     double* pVector;     // The second argument - initial
vector
     double* pResult;     //  Result vector for matrix-
vector multiplication
     int Size;            // Sizes of initial matrix and
vector
     double Start, Finish, Duration;
```

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

    printf ("Parallel matrix-vector multiplication pro-
gram\n")

    MPI_Finalize();
  }
```

Выведем на печать число доступных процессов MPI-программы *ProcNum* и ранг каждого процесса *ProcRank*. После строки, выводящей приветствие, добавьте выделенные строки в тело основной функции приложения:

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

  printf ("Parallel matrix-vector multiplication pro-
gram\n")
    printf ("Number of available processes = %d \n", Proc-
Num);
    printf ("Rank of current process = %d \n", ProcRank);

  MPI_Finalize();
```

Скомпилируйте и запустите приложение на четырех процессах. Если все было сделано верно, то результат работы программы должен выглядеть так, как показано на рис. 1.15. Каждый процесс должен напечатать по три строки: начальное сообщение, значение количества процессов и свой ранг. Значение количества процессов во всех процессах одно и то же, а ранги – разные. Обратите внимание на то, что ранги печатаются не по порядку. Выполните несколько запусков приложения. Убедитесь в том, что порядок печати рангов может меняться от запуска к запуску.

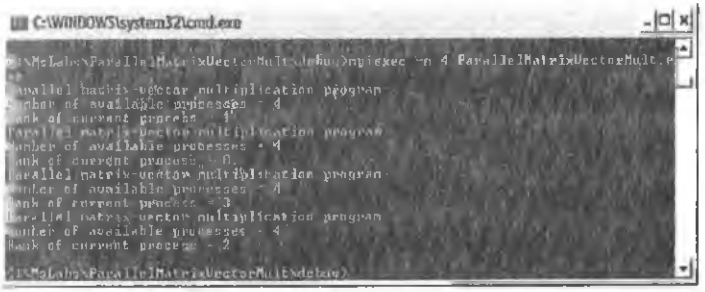

Рис. 1.15. Печать количества и ранга процессов

Разумно внести такие изменения в код, чтобы печать приветствия и числа доступных процессов выполнял только один процесс, например, процесс с рангом 0. Добавьте выделенный код в приложение:

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

if (ProcRank == 0) {
 printf ("Parallel matrix-vector multiplication pro-
gram\n")
 printf ("Number of abailable processes = %d \n", Proc-
Num);
}
printf ("Rank of current process = %d \n", ProcRank);

MPI_Finalize();
```

Повторно скомпилируйте и запустите приложение. Убедитесь в том, что теперь приветствие и число процессов печатается только один раз. Выполните несколько запусков приложения, изменяя количество доступных процессов.

Задание 4 – Ввод размера матрицы и вектора

Теперь перейдем к организации ввода и вывода данных. Как уже известно из материалов упражнения 2, разработка приложения, выполняющего умножение матрицы на вектор, начинается с задания исходных объектов. На самом первом этапе нужно определить размер этих объектов.

Для инициализации вычислительных процессов, как и ранее,

служит функция *ProcessInitialization*:

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

Для определения размеров объектов необходимо реализовать диалог с пользователем. Такой диалог должен проводить только один процесс. Этот процесс назовем *ведущим процессом*. Обычно в качестве ведущего процесса используется процесс с нулевым рангом. Добавьте выделенный фрагмент кода в тело функции *ProcessInitialization*:

```
// Function for memory allocation and initialization of
objects' elements void ProcessInitialization(double*
&pMatrix, double* &pVector,
  double* &pResult, int &Size) {
  if (ProcRank == 0) {
  printf("\nEnter size of the initial objects: ");
  scanf("%d", &Size);
  }
}
```

В ответ на вопрос, пользователь вводит размер объектов, который затем считывается нулевым процессом параллельной программы из стандартного потока ввода *stdin* и сохраняется в переменной *Size*. Итак, после выполнения выделенного фрагмента кода, ведущий процесс параллельной программы хранит в переменной *Size* введенный размер объектов.

При вводе размера объектов возможно возникновение ошибочных ситуаций. Так, например, в качестве размера объектов пользователь может указать число, меньшее, чем число доступных процессов. Кроме того, для более быстрой и простой подготовки первого варианта параллельной программы будем вначале предполагать, что размер объектов нацело делится на число процессов. В этом случае все процессы обрабатывают одно и то же количество строк исходной матрицы, и получают одно и то же число элементов результирующего вектора (вариант программы для общего случая, когда размер объектов не кратен числу процессов, будет рассмотрен в задании 11). В случае ввода пользователем некорректного размера матрицы и вектора, приложение должно либо завершить свое выполнение, либо продолжать запрашивать размер до тех пор, пока пользователь не введет "правильное" число. Как и ранее, реализуем второй вариант

[illegible] - для этого тот фрагмент кода, который производит [illegible] размера объектов, поместим в цикл с постусловием:

```
// Function for memory allocation and initialization of
objects' elements void ProcessInitialization(double*
&pMatrix, double* &pVector,
double* &pResult, int &Size) {
if (ProcRank == 0) {
do {
printf("\nEnter size of the initial objects: ");
scanf("%d", &Size);
if (Size < ProcNum) {
printf("Size of the objects must be greater than "
"number of processes! \n ");
}
if (Size%ProcNum != 0) {
printf("Size of objects must be divisible by "
"number of processes! \n");
}
}
while ((Size < ProcNum) || (Size%ProcNum != 0));
}
}
```

После того, как значение переменной *Size* определено корректно, необходимо передать это значение остальным процессам. Для этого используем функцию широковещательной рассылки от одного процесса остальным. Функция имеет следующий интерфейс:

```
int MPI_Bcast(void *buf, int count, MPI_Datatype type, int
root,
MPI_Comm comm),
где
buf, count, type - буфер памяти с отправляемым сообще-
нием (для
процесса с рангом root), и для приема сообщений для всех
остальных
процессов,
```

- **root** - ранг процесса, выполняющего рассылку данных,
- **comm** - коммуникатор, в рамках которого выполняется передача данных.

В нашем случае необходимо передать значение переменной *Size* с нулевого процесса остальным процессам:

```
if (ProcRank == 0) {
<...>
```

```
}
MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Добавьте вызов функции инициализации вычислительных процессов вместо строк, выполняющих печать количества процессов и их рангов:

```
void main(int argc, char* argv[]) {
 double* pMatrix;        // The first argument - initial
matrix
 double* pVector;        // The second argument - initial
vector
 double* pResult;        //  Result  vector  for  matrix-
vector multiplication
 int Size;                 // Sizes of initial matrix and
vector
 double* pProcRows;
 double* pProcResult;
 int RowNum;
 time_t start, finish;
 double duration;
 MPI_Init(&argc, &argv);
 MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
 MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
 if (ProcRank == 0)
 printf("Parallel   matrix-vector   multiplication   pro-
gram\n");
 // Memory allocation and data initialization
 ProcessInitialization(pMatrix, pVector, pResult, Size);
 MPI_Finalize();
}
```

Скомпилируйте и запустите приложение. Убедитесь в том, что все ошибочные ситуации обрабатываются корректно. Для этого выполните несколько запусков приложения, задавая различное количество параллельных процессов (при помощи параметра запуска утилиты **mpiexec**) и разные размеры объектов.

Задание 5 – Ввод исходных данных

После того, как размер объектов определен, можно перейти к выделению памяти и заданию значений элементов матрицы и вектора. Обычно определение начальных данных осуществляется одним из процессов (пусть, как и ранее, этим процессом будет процесс с рангом 0). Далее, согласно схеме параллельных вычислений, изложенной в упражнении 3, исходная матрица распределяется между всеми процессами таким образом, что

каждый процесс обрабатывает непрерывную последовательность строк (горизонтальную полосу). Отметим, что первая версия разрабатываемой программы ориентирована на случай, когда размер матрицы делится нацело на число процессов, то есть полосы матрицы на всех процессах содержат одно и то же количество строк. Это количество строк будем хранить в переменной *RowNum*. Адреса буферов памяти, где содержатся горизонтальные полосы строк на каждом из процессов, будем хранить в переменной *pProcRows* (*pProcRows* – матрица, которая содержит *RowNum* строк и *Size* столбцов и хранится построчно). Исходный вектор *pVector* копируется с процесса с рангом 0 на все процессы. В результате умножения полосы матрицы на вектор, каждый процесс получает *RowNum* элементов результирующего вектора. Будем хранить эти элементы в массиве *pProcResult*.

В основной функции программы объявим переменные:

```
   void main(int argc, char* argv[]) {
    double* pMatrix;          // The first argument - ini-
tial matrix
    double* pVector;          // The second argument - ini-
tial vector
    double* pResult;          // Result vector for matrix-
vector multiplication
    int Size;                 // Sizes of initial matrix
and vector
    double* pProcRows;        // Stripe of the matrix on
current process
    double* pProcResult; // Block of result vector on cur-
rent process
    int RowNum;               // Number of rows in matrix
stripe
    double Start, Finish, Duration;
```

Изменим список аргументов функции *ProcessInitialization* для того, чтобы эта функция могла определять значение переменной *RowNum* и выделять память для хранения новых объектов:

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

Определим значение переменной *RowNum*, выделим память для хранения объектов и проинициализируем исходные матрицу и вектор на ведущем процессе. Добавьте выделенный код в тело функции *ProcessInitialization*:

```
if (ProcRank == 0) {
 <…>
}
MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);

// Determine the number of matrix rows stored on each
process
RowNum = Size/ProcNum;

// Memory allocation
pVector = new double [Size];
pResult = new double [Size];
pProcRows = new double [RowNum*Size];
pProcResult = new double [RowNum];

// Obtain the values of initial objects' elements
if (ProcRank == 0) {
// Initial matrix exists only on the pivot process
pMatrix = new double [Size*Size];
// Values of elements are defined only on the pivot
process
DummyDataInitialization(pMatrix, pVector, Size);
 } // if
```

Для задания элементов матрицы и вектора на ведущем процессе мы воспользовались функцией генерации данных *DummyDataInitialization*, которая была нами разработана при реализации последовательного приложения для умножения матрицы на вектор. Напомним, что эта функция заполняет вектор *pVector* единицами, а значение элемента матрицы *pMatrix* равно номеру строки, в которой этот элемент расположен.

Для контроля правильности ввода исходных данных можно воспользоваться функциями *PrintMatrix* и *PrintVector*, которые были реализованы при разработке последовательного приложения. В основной функции программы после вызова функции *ProcessInitialization* добавьте вызовы функций *PrintMatrix* и *PrintVector* для объектов *pMatrix* и *pVector* на нулевом процессе. Скомпилируйте и запустите приложение. Убедитесь в том, что данные задаются корректно.

Задание 6 – 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 исходной матрицы *pMatrix*, на всех процессах выделялась память для хранения исходного вектора *pVector* и вектора-результата *pResult*, а также память для хранения полосы матрицы *pProcRows* и блока вектора результата *pProcResult*. Все эти объекты необходимо передать в функцию *ProcessTermination* в качестве аргументов:

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

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

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

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

Задание 7 – Распределение данных между процессами

В соответствии со схемой параллельных вычислений, изложенной в предыдущем упражнении, матрица должна быть разделена между процессами равными горизонтальными полосами, а исходный вектор должен быть скопирован на все процессы.

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*, а также размеры объектов (размер матрицы и вектора *Size* и число полос в горизонтальной полосе *RowNum*):

```
// Function for distribution of the initial objects be-
tween the processes void DataDistribution(double* pMatrix,
double* pProcRows, double* pVector,
    int Size, int RowNum);
```

Для копирования вектора на все процессы параллельной программы используем, как и ранее, функцию широковещательной рассылки:

```
// Function for distribution of the initial objects be-
tween the processes void DataDistribution(double*
pMatrix, double* pProcRows, double* pVector,
int Size, int RowNum) {
MPI_Bcast(pVector, Size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
```

При нашем подходе матрица хранится в одномерном массиве построчно. Следовательно, для того, чтобы разделить матрицу на горизонтальные полосы, необходимо разделить этот массив на блоки одинакового размера и разослать эти блоки процессам. Такая операция носит название обобщенной передачи данных от одного процесса всем процессам MPI программы (*распределение данных*). Данная операция отличается от широковещательной рассылки тем, что процесс передает всем процессам программы различающиеся данные. Выполнение данной операции может быть обеспечено при помощи функции:

```
int MPI_Scatter(void *sbuf,int scount,MPI_Datatype stype,
void *rbuf,int rcount,MPI_Datatype rtype,
int root, MPI_Comm comm),
```

где

- `sbuf, scount, stype` - параметры передаваемого сообщения (`scount` определяет количество элементов, передаваемых на каждый процесс),
- `rbuf, rcount, rtype` - параметры сообщения, принимаемого в процессах,
- `root` - ранг процесса, выполняющего рас-

сылку данных,

- comm - коммуникатор, в рамках которого выполняется передача данных.

Добавьте в тело функции *DataDistribution* вызов функции MPI_Scatter:

```
// Function for distribution of the initial objects between the processes void DataDistribution(double* pMatrix, double* pProcRows, double* pVector,
  int Size, int RowNum) {
  MPI_Bcast(pVector, Size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Scatter(pMatrix, RowNum*Size, MPI_DOUBLE, pProcRows, RowNum*Size,
  MPI_DOUBLE, 0, MPI_COMM_WORLD);
 }
```

Соответственно, вызывать эту функцию из основной программы нужно непосредственно после вызова функции инициализации вычислительного процесса *ProcessInitialization*, перед тем, как приступить непосредственно к выполнению матрично-векторного умножения:

```
ProcessInitialization(pMatrix, pVector, pResult, pProcRows, pProcResult,
  Size, RowNum);

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

Теперь выполним проверку правильности разделения данных между процессами. Для этого после выполнения функции *DataDistribution* распечатаем исходные матрицу и вектор, а затем полосы матрицы, 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*:

```
void TestDistribution(double* pMatrix, double* pVector, double* pProcRows,
 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(pVector, Size);
  }
  MPI_Barrier(MPI_COMM_WORLD);
  }
}
```

Такой способ проверки правильности выполнения этапов параллельной программы называется *отладочной печатью* и часто используется в процессе разработки параллельных приложений в том случае, если объем данных, которые необходимо проверить, невелик.

Поясним реализацию функции *TestDistribution*. В ряде ситуаций независимо выполняемые в процессах вычисления необходимо синхронизировать. *Синхронизация* процессов, т.е. одновременное достижение процессами тех или иных точек процесса вычислений, обеспечивается при помощи функции MPI:

```
Int MPI_Barrier(MPI_Comm comm);
```

Функция *MPI_Barrier* определяет коллективную операции и, тем самым, при использовании должна вызываться всеми процессами используемого коммуникатора. При вызове функции *MPI_Barrier* выполнение процесса блокируется, продолжение вычислений процесса произойдет только после вызова функции *MPI_Barrier* всеми процессами коммуникатора.

В функции *TestDistribution* функция *MPI_Barrier* используется для того, чтобы обеспечить порядок печати. Так, сначала необходимо напечатать исходные объекты на ведущем процессе. Для того, чтобы в то же самое время свою печать не вели другие процессы параллельной программы, вызвана функция *MPI_Barrier*. Выполнение действий на других процессах начнется только после того, как ведущий процесс вызовет *MPI_Barrier* по окончании печати исходных объектов. Далее, та же схема

используется для того, чтобы процессы печатали свои полосы матриц по порядку (сначала свою полосу печатает процесс с рангом 0, далее процесс с рангом 1 и т.д.).

Добавьте вызов функции проверки распределения непосредственно после функции DataDistribution:

```
  // Distributing the initial objects between the pro-
cesses
  DataDistribution(pMatrix,  pProcRows,  pVector,  Size,
RowNum);

  // Distribution test
  TestDistribution(pMatrix,  pVector,  pProcRows,  Size,
RowNum);
```

Напомним, что функция генерации исходных данных *DummyDataInitialization* устроена таким образом, что она назначает элементу матрицы значение, равное номеру строки, в которой он расположен. Значит, после разделения данных на процессе с рангом *i* должны оказаться строки матрицы, в которых хранятся значения в интервале от *i*RowNum* до *(i+1)*RowNum-1*.

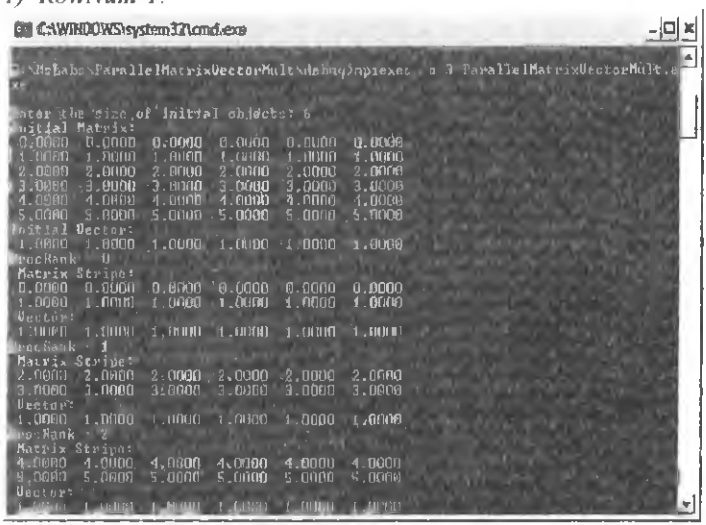

Рис. 1.16. Распределение данных в случае, когда приложение запускается на трех процессах, а порядок матрицы равен шести

Скомпилируйте приложение. Если в приложении обнаружились ошибки, исправьте их, сверяя свой код с кодом, представленным в данном пособии. Запустите приложение на трех процессах и установите размер данных 6. Убедитесь в том, что распределение данных выполняется правильно (рис. 1.16).

Задание 8 – Реализация умножения матрицы на вектор

Выполнение умножения происходит в функции *ParallelResultCalculation*. Для вычисления блока результирующего вектора необходимо иметь доступ к полосе матрицы *pProcRows*, вектору *pVector* и блоку результирующего вектора *pProcResult*. Кроме того, необходимо знать размеры этих объектов. Таким образом, в функцию *ParallelResultCalculation* необходимо передать следующие аргументы:

```
void ParallelResultCalculation(double* pProcRows, double*
pVector,
 double* pProcResult, int Size, int RowNum);
```

Для получения значения каждого конкретного элемента результирующего вектора необходимо, как и в последовательном алгоритме, выполнить скалярное умножение строки матрицы на вектор-аргумент. Отличие от последовательного кода состоит только в том, что процесс работает не с самой матрицей, а ее частью *pProcRows* и обрабатывает не *Size*, а только *RowNum* строк.

```
    // Process rows and vector multiplication
    void ParallelResultCalculation(double* pProcRows, dou-
ble* pVector,
     double* pProcResult, int Size, int RowNum) {
     int i, j;
     for (i=0; i<RowNum; i++) {
     pProcResult[i] = 0;
     for (j=0; j<Size; j++) {
     pProcResult[i] += pProcRows[i*Size+j]*pVector[j];
     }
     }
    }
```

Вызывать функцию *ParallelResultCalculation* в основной программе нужно следующим образом:

```
    // Distributing the initial objects between the pro-
cesses DataDistribution(pMatrix, pProcRows, pVector, Size,
RowNum);  TestDistribution(pMatrix,  pVector,  pProcRows,
Size, RowNum);
```

```
    // Process rows and vector multiplication
    ParallelResultCalculation(pProcRows, pVector, pProcRe-
sult, Size, RowNum);
```

Этот этап, как и все предыдущие, необходимо проверить. Разработаем для этого функцию проверки частичных результатов, которые были получены каждым из процессов, *TestPartialResults*. Снова используем отладочную печать:

```
    // Function for testing the results of multiplication
of matrix stripe
    // by a vector
    Void    TestPartialResults(double*    pRocResult,    int
RowNum) {
     Int i; //Loop variable
     For (i=0; i<ProcNum; i++) {
     If (ProcRank==i) {
     printf("ProcRank = %d \n", ProcRank);
     printf("Part of result vector: \n");
     PrintVector(pProcResult, RowNum);
     }
     MPI_Barrier(MPI_COMM_WORLD);
     }
    }
```

Для уменьшения объема отладочного вывода закомментируйте вызов функции *TestDistribution*. Вызов функции *TestPartialResults* следует поместить непосредственно после выполнения умножения:

```
DataDistribution(pMatrix,    pProcRows,    pVector,    Size,
RowNum);
//  TestDistribution(pMatrix,  pVector,  pProcRows,  Size,
RowNum);

// Process rows and vector mulriplication ParallelResult-
Calculation(pProcRows,    pVector,    pProcResult,    Size,
RowNum); TestPartialResults(pProcResult, RowNum);
```

Для матриц, элементы которых задаются при помощи функции *DummyDataInitialization*, результат умножения на вектор, заполненный единицами, заранее известен. На процессе с рангом *i* получается блок результирующего вектора, содержащий элементы в диапазоне от *Size*(i*RowNum)* до *Size*((i+1)*RowNum-1)*. Так, например, если параллельное приложение запускается на двух процессах, а размер объектов равен шести, то на первом процессе должен получиться блок (0, 6, 12),

```
tors are identical
    int i; // Loop variable

    if (ProcRank == 0) {
    pSerialResult = new double [Size];
    SerialResultCalculation(pMatrix, pVector, pSerialRe-
sult, Size);
    for (i=0; i<Size; i++) {
    if (pResult[i] != pSerialResult[i])
    equal = 1;
    }
    if (equal == 1)
    printf("The results of serial and parallel algorithms
"
    "are NOT identical. Check your code.");
    Else
    printf("The results of serial and parallel algorithms
are "
    "identical.");
    delete [] pSerialResult;
    }
  }
```

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 (функция *TestDistribution*, *TestPartialResult*). Вместо функции *DummyDataInitialization*, которая генерирует матрицы простого вида, вызовите функцию *RandomDataInitialization*, которая генерирует матрицу и вектор при помощи датчика случайных чисел. Скомпилируйте и запустите приложение. Задавайте различные объемы исходных данных. Убедитесь в том, что приложение работает правильно.

Задание 11 – Реализация вычислений для любых размеров матрицы

Параллельное приложение, которое разрабатывалось в ходе выполнения предыдущих заданий, было ориентировано на случай, когда размер исходных объектов *Size* нацело делится на число процессоров *ProcNum*. В этом случае матрица делится между процессами на равные полосы, число *RowNum* строк, которые обрабатывает процесс, для всех процессов было одним и тем же.

Теперь рассмотрим случай, когда размер объектов *Size* не кратен числу процессов *ProcNum*. В этом случае значение *RowNum* числа обрабатываемых строк на каждом процессе будет свое: некоторые процессы получат $\lfloor Size / \mathrm{Pr}ocNum \rfloor$, а остальные - $\lceil Size / \mathrm{Pr}ocNum \rceil$ строк матрицы (операция $\lfloor \ \rfloor$ означает округление значения до ближайшего меньшего целого числа, операция $\lceil \ \rceil$ – округление до ближайшего большего целого числа).

В функции *ProcessInitialization* уберем обработку ошибочной ситуации, которая возникает в случае, когда размер объектов не делится нацело на число процессов. Теперь необходимо определить, сколько строк должен обрабатывать каждый процесс. Один из самых простых способов может состоять в следующем: всем процессам, кроме последнего (процесса с рангом *ProcNum*-1) выделяется $\lfloor Size / \mathrm{Pr}ocNum \rfloor$ строк матрицы, а последнему процессу выделяются все оставшиеся строки ($Size - \lfloor Size / \mathrm{Pr}ocNum \rfloor . (\mathrm{Pr}ocNum - 1)$ штук). Однако, в этом случае, возможно, что нагрузка будет распределена между процессами неравномерно. Так, например, если порядок матрицы равен 5, а параллельное приложение запускается на трех процессах, то первым двум процессам будет выделено по одной строке матрицы, а последнему процессу – три строки.

Чтобы избежать такой неравномерности, будем использовать следующий алгоритм распределения. Будем последовательно выделять строки процессам: в первую очередь определим, сколько строк будет обрабатывать процесс с рангом 0, затем – процесс с рангом 1, и так далее. Процессу с рангом 0 выделим

$\lfloor Size / \mathrm{Pr}ocNum \rfloor$ строк (результат операции $\lfloor\ \rfloor$ совпадает с результатом целочисленного деления переменной *Size* на переменную *ProcNum*). После выполнения этой операции остается распределить $Size - \lfloor Size / \mathrm{Pr}ocNum \rfloor$ строк между *ProcNum-1* процессами и т.д. Как результат, каждому следующему процессу *i* назначим количество строк, равное результату целочисленного деления оставшегося количества строк *RestRows* на оставшееся число процессов, т.е. $\lfloor \mathrm{Re}stRows / (\mathrm{Pr}ocNum - i) \rfloor$ строк.

Изменим определение значения переменной *RowNum*:

```
    // Function for memory allocation and data initiali-
zation
    void ProcessInitialization (double* &pMatrix, double*
&pVector,
     double*  &pResult,  double*  &pProcRows,  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 size of the initial objects: ");
     scanf("%d", &Size);
     if (Size < ProcNum) {
     printf("Size of the objects 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);

    pVector = new double [Size];
    pResult = new double [Size];
```

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

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

В случае, когда матрица распределяется между процессами не поровну, для распределения данных нельзя использовать функцию *MPI_Scatter*. Вместо нее используется более общая функция *MPI_Scatterv*, которая дает возможность одному процессу распределить непрерывный набор элементов всем процессам коммуникатора, включая его самого. Эта функция имеет следующий интерфейс:

```
MPI_Scatterv (void *send_buffer, int* send_cnt, int*
send_disp,
MPI_Datatype send_type, void *receive_buffer, int
recv_cnt,
MPI_Datatype recv_type, int root,MPI_COMM communica-
tor ),
```

где

- **send_buffer** - указатель на буфер, содержащий элементы для распределения.
- **send_cnt** - i-ый элемент – количество последовательных элементов в send_buffer, предназначенных процессу i.
- **send_disp** - i-ый элемент – это смещение первого элемента, предназначенного процессу i, относительно начала send_buffer.
- **send_type** - тип элементов в send_buffer.
- **recv_buffer** - указатель на буфер, содержащий порцию получаемых данным процессом элементов.
- **recv_cnt** - количество элементов, которые получит данный процесс.

- **recv_type** - тип элементов в recv_buffer.

- root - идентификатор процесса, содержащего данные для раскидывания.

- **communicator** - коммуникатор, в котором происходит раскидывание.

Итак, для того, чтобы вызвать функцию *MPI_Scatterv*, необходимо определить два вспомогательных массива, размер этих массивов совпадает с числом доступных процессов. Внесем необходимые изменения в код функции *DataDistribution*:

```
// Data distribution among the processes
void DataDistribution(double* pMatrix, double*
pProcRows, double* pVector,
 int Size, int RowNum) {
 int *pSendNum; // the number of elements sent to the
process
 int *pSendInd; // the index of the first data element
sent to the process
 int RestRows=Size; // Number of rows, that haven't
been distributed yet
 MPI_Bcast(pVector, Size, MPI_DOUBLE, 0,
MPI_COMM_WORLD);
 // Alloc memory for temporary objects
 pSendInd = new int [ProcNum];
 pSendNum = new int [ProcNum];

 // Determine the disposition of the matrix rows for
current process
 RowNum = (Size/ProcNum);
 pSendNum[0] = RowNum*Size;
 pSendInd[0] = 0;
 for (int 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;
  }
```

Аналогично для сбора данных, вместо функции *MPI_Allgather*, ориентированной на сбор данных одинакового объема со всех процессов коммуникатора, будем использовать более общую функцию *MPI_Allgatherv*. Функция имеет следующий интерфейс:

```
MPI_Allgatherv(void*     send_buffer,     int     send_cnt,
MPI_Datatype send_type,
    void*  recv_buffer,  int*  recv_cnt,  int*  recv_disp,
MPI_Datatype recv_type,
    MPI_Comm communicator),
```

где

- **send_buffer** – адрес буфера, из которого данный процесс отсылает данные.
- **send_cnt** – количество элементов в send_buffer.
- **send_type** – тип элементов в send_buffer.
- **recv_buffer** – адрес буфера, куда помещается результат сбора.
- **recv_cnt** – i-ый элемент равен объему данных, которые передает процесс с рангом i.
- **recv_disp** – i-ый элемент – это смещение первого элемента, принятого от процесса i, относительно начала recv_buffer.
- **recv_type** – тип элементов в recv_buffer.
- **communicator** – коммуникатор, в котором происходит сбор.

Как и при использовании *MPI_Scatterv*, использование *MPI_Allgatherv* требует использования двух дополнительных массивов:

```
  // Function for gathering the result vector
  void  ResultReplication(double*  pProcResult,  double*
pResult, int Size,
    int RowNum) {
    int i;                // Loop variable
    int *pReceiveNum;     // Number of elements, that cur-
rent process sends
```

```
    int *pReceiveInd;     /* Index of the first element
from current process in
    result vector */
    int RestRows=Size; // Number of rows, that haven't
been distributed yet

    // Alloc memory for temporary objects
    pReceiveNum = new int [ProcNum];
    pReceiveInd = new int [ProcNum];

  // Detrmine the disposition of the result vector block
of current
  // processor
  pReceiveInd[0] = 0;

  pReceiveNum[0] = Size/ProcNum;
  for (i=1; i<ProcNum; i++) {
  RestRows -= pReceiveNum[i-1];
  pReceiveNum[i] = RestRows/(ProcNum-i);
  pReceiveInd[i] = pReceiveInd[i-1]+pReceiveNum[i-1];
  }
  // Gather the whole result vector on every processor
  MPI_Allgatherv(pProcResult,        pReceiveNum[ProcRank],
MPI_DOUBLE, pResult,
  pReceiveNum, pReceiveInd, MPI_DOUBLE, MPI_COMM_WORLD);

  // Free the memory
  delete [] pReceiveNum;
  delete [] pReceiveInd;
}
```

Скомпилируйте и запустите приложение. Проверьте правильность выполнения умножения при помощи функции *CheckResult*.

Задание 12 – Проведение вычислительных экспериментов

Основная задача при реализации параллельных алгоритмов решения сложных вычислительных задач – обеспечить ускорение вычислений (по сравнению с последовательным алгоритмом) за счет использования нескольких процессоров. Время выполнения параллельного алгоритма должно быть меньше, чем при выполнении последовательного алгоритма.

Определим время выполнения параллельного алгоритма. Для этого добавим в программный код замеры времени. Следует отметить, что в MPI для замеров времени имеется специальная

функция:

```
MPI_Wtime();
```

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 *ResultReplication*:

```
    ProcessInitialization(pMatrix, pVector, pResult,
pProcRows, pProcResult,
    Size, RowNum);

    Start = MPI_Wtime();
    DataDistribution(pMatrix, pProcRows, pVector, Size,
RowNum);
    ParallelResultCalculation(pProcRows, pVector,
pProcResult, Size, RowNum);
    ResultReplication(pProcResult, pResult, Size,
RowNum);
    Finish = MPI_Wtime();
    Duration = Finish-Start;

    TestResult(pMatrix, pVector, pResult, Size);
    if (ProcRank == 0) {
    printf("Time of execution = %f\n", Duration);
    }

    ProcessTermination(pMatrix, pVector, pResult,
pProcRows, pProcResult);

    MPI_Finalize();
```

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 основания 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:

Размер матрицы	процесса		4 процесса		процессов	
	Модель	Экспе-римент	Модель	Экспери-мент	Модель	Экспери-мент
10						
100						
1000						
2000						
3000						
4000						
5000						
6000						
7000						
8000						
9000						
10000						

В графу "Последовательный алгоритм" внесите время выполнения последовательного алгоритма, замеренное при проведении тестирования последовательного приложения в упражнении 2. 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 время выполнения параллельного алгоритма, реализованного согласно вычислительной схеме, приведенной в упражнении 3, можно воспользоваться следующим соотношением:

$$T_p = \lceil n/p \rceil (2n-1)\tau + \alpha \lceil log_2 p \rceil + w \lceil n/p \rceil \left(2^{\lceil log_2 p \rceil} - 1\right)/\beta \quad (1.4)$$

(подробный вывод этой формулы приведен в разделе 7 учебного курса). Здесь n – размер объектов, 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.

Вычислите теоретическое время выполнения параллельного

алгоритма по формуле (1.4).

Результаты занесите в таблицу:

Размер матрицы	процесса		4 процесса		процессов	
	Модель	Экспе-римент	Модель	Экспери-мент	Модель	Экспери-мент
10						
100						
1000						
2000						
3000						
4000						
5000						
6000						
7000						
8000						
9000						
10000						

Контрольные вопросы

•В качестве времени выполнения параллельного алгоритма было выбрано время, затраченное первым процессом. Как нужно модифицировать код для того, чтобы выбрать максимальное среди времен, полученных на всех процессах?

•Насколько сильно отличаются время, затраченное на выполнение последовательного и параллельного алгоритма? Почему?

•Получилось ли ускорение при матрице размером 10 на 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 параллельный алгоритм умножения матрицы на вектор, основанный на ленточном вертикальном разделении матрицы. Напишите программу, реализующую этот алгоритм.

2. Изучите параллельный алгоритм умножения матрицы на вектор, основанный на блочном разделении матрицы. Напишите программу, реализующую этот алгоритм.

Программный код последовательного приложения для умножения матрицы на вектор

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

// 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] = 1;
 for (j=0; j<Size; j++)
 pMatrix[i*Size+j] = i;
 }
}

// 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++)
 pMatrix[i*Size+j] = rand()/double(1000);
 }
 }

// Function for memory allocation and definition of ob-
ject's elements
```

```
   void ProcessInitialization (double* &pMatrix, double*
&pVector,
    double* &pResult, int &Size) {
    // Size of initial matrix and vector definition
    do {
    printf("\nEnter size of the initial objects: ");
    scanf("%d", &Size);
    printf("\nChosen objects 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];
    // Definition of matrix and vector elements
    DummyDataInitialization(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]);
   }

   // Function for matrix-vector multiplication
   void ResultCalculation(double* pMatrix, double* pVec-
tor, double* pResult,
    int Size) {
    int i, j;   // Loop variables
    for (i=0; i<Size; i++) {
    pResult[i] = 0;
    for (j=0; j<Size; j++)
    pResult[i] += pMatrix[i*Size+j]*pVector[j];
    }
```

```
}

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


void main() {
 double* pMatrix;     // The first argument - initial
matrix
 double* pVector;     // The second argument - initial
vector
 double* pResult;     // Result vector for matrix-
vector multiplication
 int Size;              // Sizes of initial matrix and
vector
 time_t start, finish;
 double duration;

 printf("Serial matrix-vector multiplication pro-
gram\n");
 // Memory allocation and definition of objects' ele-
ments
 ProcessInitialization(pMatrix, pVector, pResult,
Size);

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

 // Matrix-vector multiplication
 start = clock();
 ResultCalculation(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 time spent by matrix-vector multipli-
```

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

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

Программный код параллельного приложения для умножения матрицы на вектор

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

  int ProcNum = 0;          // Number of available pro-
cesses
  int ProcRank = 0;         // Rank of current process

  // 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] = 1;
    for (j=0; j<Size; j++)
    pMatrix[i*Size+j] = i;
    }
  }

  // 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++)
    pMatrix[i*Size+j] = rand()/double(1000);
    }
  }

  // Function for memory allocation and data initializa-
tion
```

```
   void ProcessInitialization (double* &pMatrix, double*
&pVector,
    double*   &pResult,   double*   &pProcRows,   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 size of the initial objects: ");
    scanf("%d", &Size);
    if (Size < ProcNum) {
    printf("Size of the objects 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);

    pVector = new double [Size];
    pResult = new double [Size];
    pProcRows = new double [RowNum*Size];
    pProcResult = new double [RowNum];

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

   // Data distribution among the processes
   void    DataDistribution(double*    pMatrix,    double*
pProcRows, double* pVector,
    int Size, int RowNum) {
    int *pSendNum; // the number of elements sent to the
process
    int *pSendInd; // the index of the first data element
sent to the process
    int RestRows=Size; // Number of rows, that haven't
been distributed yet
```

```
    MPI_Bcast(pVector,      Size,      MPI_DOUBLE,      0,
MPI_COMM_WORLD);

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

    // Define the disposition of the matrix rows for cur-
rent process
    RowNum = (Size/ProcNum);
    pSendNum[0] = RowNum*Size;
    pSendInd[0] = 0;
 for (int 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;
   }

   // Function for gathering the result vector
   void ResultReplication(double* pProcResult, double*
pResult, int Size,
    int RowNum) {
    int i;                // Loop variable
    int *pReceiveNum;     // Number of elements, that cur-
rent process sends
    int *pReceiveInd;     /* Index of the first element
from current process
    in result vector */
    int RestRows=Size; // Number of rows, that haven't
been distributed yet

    //Alloc memory for temporary objects
    pReceiveNum = new int [ProcNum];
    pReceiveInd = new int [ProcNum];
```

```
    //Define the disposition of the result vector block of
current processor
    pReceiveInd[0] = 0;
    pReceiveNum[0] = Size/ProcNum;
  for (i=1; i<ProcNum; i++) {
  RestRows -= pReceiveNum[i-1];
  pReceiveNum[i] = RestRows/(ProcNum-i);
    pReceiveInd[i] = pReceiveInd[i-1]+pReceiveNum[i-1];
    }

    //Gather the whole result vector on every processor
    MPI_Allgatherv(pProcResult,      pReceiveNum[ProcRank],
MPI_DOUBLE, pResult,
    pReceiveNum, pReceiveInd, MPI_DOUBLE, MPI_COMM_WORLD);

    //Free the memory
    delete [] pReceiveNum;
    delete [] pReceiveInd;
   }

   // Function for sequential matrix-vector multiplication
   void SerialResultCalculation(double*           pMatrix,
   pResult, int Size) {
    int i, j;    // Loop variables
    for (i=0; i<Size; i++) {
    pResult[i] = 0;
    for (j=0; j<Size; j++)
    pResult[i] += pMatrix[i*Size+j]*pVector[j];
    }
   }

   // Function for calculating partial matrix-vector mul-
tiplication
   void ParallelResultCalculation(double* pProcRows, dou-
ble* pVector, double*
   pProcResult, int Size, int RowNum) {
    int i, j;    // Loop variables
    for (i=0; i<RowNum; i++) {
    pProcResult[i] = 0;
    for (j=0; j<Size; j++)
    pProcResult[i] += pProcRows[i*Size+j]*pVector[j];
    }
   }

   // 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]);
  }

  void TestDistribution(double* pMatrix, double* pVector,
double* pProcRows,
    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(pVector, Size);
    }
    MPI_Barrier(MPI_COMM_WORLD);
    }
  }

  void  TestPartialResults(double*  pProcResult,  int
RowNum) {
    int i;     // Loop variables
    for (i=0; i<ProcNum; i++) {
    if (ProcRank == i) {
    printf("\nProcRank = %d \n Part of result vector: \n",
ProcRank);
    PrintVector(pProcResult, RowNum);
    }
    MPI_Barrier(MPI_COMM_WORLD);
    }
  }
```

```
   void TestResult(double* pMatrix, double* pVector, dou-
ble* pResult,
     int Size) {
     // Buffer for storing the result of serial matrix-
vector multiplication
     double* pSerialResult;
     // Flag, that shows wheather the vectors are identical
or not
     int equal = 0;
     int i;                        // Loop variable

     if (ProcRank == 0) {
     pSerialResult = new double [Size];
     SerialResultCalculation(pMatrix, pVector, pSerialRe-
sult, Size);
     for (i=0; i<Size; i++) {
     if (pResult[i] != pSerialResult[i])
     equal = 1;
     }
     if (equal == 1)
     printf("The results of serial and parallel algorithms
"
     "are NOT identical. Check your code.");
     else
     printf("The results of serial and parallel algorithms
"
     "are identical.");
     }
   }

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

   void main(int argc, char* argv[]) {
     double* pMatrix;      // The first argument - initial
matrix
     double* pVector;      // The second argument - initial
vector
     double* pResult;      // Result vector for matrix-
vector multiplication
```

```
    int Size;               // Sizes of initial matrix and
vector
    double* pProcRows;
    double* pProcResult;
    int RowNum;
    double Start, Finish, Duration;

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

    ProcessInitialization(pMatrix,    pVector,    pResult,
pProcRows, pProcResult,
    Size, RowNum);

    Start = MPI_Wtime();
    DataDistribution(pMatrix,  pProcRows,  pVector,  Size,
RowNum);
    ParallelResultCalculation(pProcRows, pVector, pProcRe-
sult, Size, RowNum);
    ResultReplication(pProcResult, pResult, Size, RowNum);
    Finish = MPI_Wtime();
    Duration = Finish-Start;

    TestResult(pMatrix, pVector, pResult, Size);
    if (ProcRank == 0) {
    printf("Time of execution = %f\n", Duration);
    }

    ProcessTermination(pMatrix,     pVector,     pResult,
pProcRows, pProcResult);

    MPI_Finalize();
  }
```

ЗАДАЧА 2: ПАРАЛЛЕЛЬНЫЕ АЛГОРИТМЫ МАТРИЧНОГО УМНОЖЕНИЯ

Операция умножения матриц является одной из основных задач матричных вычислений. В данной задаче рассматриваются последовательный алгоритм матричного умножения и параллельный алгоритм Фокса (*the Fox algorithm*), основанный на блочной схеме разделения данных.

Обзор задачи

Целью данной задачи является разработка параллельной программы, которая выполняет умножение двух квадратных матриц. Выполнение задачи включает:

- Упражнение 1 – Определение задачи матричного умножения
- Упражнение 2 – Реализация последовательного алгоритма матричного умножения
- Упражнение 3 – Разработка параллельного алгоритма матричного умножения
- Упражнение 4 - Реализация параллельного алгоритма умножения матриц

При выполнении задачи предполагается знание разделов "Параллельное программирование на основе MPI", "Принципы разработки параллельных методов" и раздела "Параллельные алгоритмы матричного умножения".

Упражнение 1 – Определение задачи матричного умножения

Умножение матрицы A размера $m \times n$ и матрицы B размера $n \times l$ приводит к получению матрицы C размера $m \times l$, каждый элемент которой определяется в соответствии с выражением:

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} \cdot b_{ik}, 0 \le i < m, 0 \le j < l \qquad (2.1)$$

Как следует из (2.1) , каждый элемент результирующей матрицы C есть скалярное произведение соответствующих строки матрицы A и столбца матрицы B (рис. 2.1):

$$c_{ij} = \left(a_i, b_j^T\right), a_i = \left(a_{i0}, a_{i1}, ..., a_{in-1}\right), b_j^T = \left(b_{0j}, b_{1j}, ..., b_{n-1j}\right)^T \qquad (2.2)$$

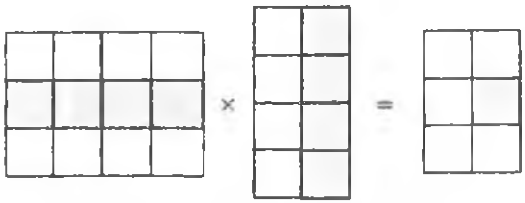

Рис. 2.2. Элемент результирующей матрицы C – результат скалярного умножения соответствующих строки матрицы A и столбца матрицы B

Так, например, при умножении матрицы A, состоящей из 3 строк и 4 столбцов на матрицу B из 4 строк и 2 столбцов, получается матрица C из 3 строк и 2 столбцов:

$$\begin{pmatrix} 3 & 2 & 0 & -1 \\ 5 & -2 & 1 & 1 \\ 1 & 0 & -1 & -1 \end{pmatrix} \times \begin{pmatrix} 1 & -1 \\ 2 & 5 \\ -3 & 2 \\ 7 & 4 \end{pmatrix} = \begin{pmatrix} 0 & 3 \\ 5 & -9 \\ -3 & -7 \end{pmatrix}$$

Рис. 2.2. Пример умножения матриц

Тем самым, получение результирующей матрицы C предполагает повторение $m \times l$ однотипных операций по умножению строк матрицы A и столбцов матрицы B. Каждая такая операция включает умножение элементов строки и столбца матриц и последующее суммирование полученных произведений.

Псевдокод для представленного алгоритма умножения матрицы на вектор может выглядеть следующим образом (здесь и далее предполагается, что матрицы, участвующие в умножении, квадратные, то есть имеют размерность *Size*×*Size*):

```
// Serial algorithm of matrix multiplication
for (i=0; i<Size; i++) {
 for (j=0; j<Size; j++) {
 MatrixC[i][j] = 0;
 for (k=0; k<Size; k++) {
 MatrixC[i][j]       =       MatrixC[i][j]       +       Ma-
trixA[i][k]*MatrixB[k][j];
 }
 }
}
```

Упражнение 2 – Реализация последовательного алгоритма матричного умножения

При выполнении этого упражнения необходимо реализовать последовательный алгоритм матричного умножения. Начальный вариант будущей программы представлен в проекте *SerailMatrixMult*, 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, умножения матриц и вывода результатов.

Задание 1 – Открытие проекта SerialMatrixMult

Откройте проект **SerialMatrixMult**, 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\SerialMatrixMult,**
- Дважды щелкните на файле **SerialMatrixMult.sln** или выбрав файл выполните команду **Open.**

После открытия проекта в окне Solution Explorer (Ctrl+Alt+L) дважды щелкните на файле исходного кода **SerialMM.cpp**, как это показано на рис. 2.3. После этих действий код, который предстоит в дальнейшем расширить будет открыт в рабочей области Visual Studio.

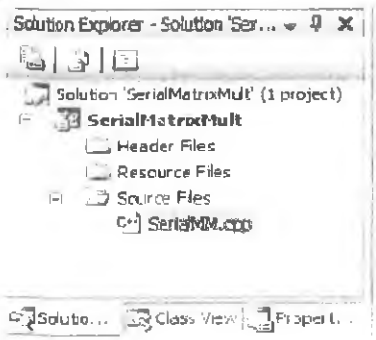

Рис. 2.2. Открытие проекта SerialMatrixMult

В файле SerialMV.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 (*pAMatrix* и *pBMatrix*) – это, соответственно, матрицы которые участвуют в матричном умножении в качестве аргументов. Третья переменная *pCMatrix* – матрица, которая должна быть получена в результате умножения. Переменная *Size* определяет размер матриц (предполагаем, что все матрицы квадратные, имеют размерность *Size*×*Size*).

```
double* pAMatrix; // The first argument of matrix multiplication
double* pBMatrix; // The second argument of matrix multiplication
double* pCMatrix; // The result matrix
int Size; // Size of matricies
```

Как и при разработке алгоритмов умножения матрицы на вектор, для хранения матриц используются одномерные массивы, в которых матрицы 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 matrix multiplication program\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 matrix multiplication program`". Для того, чтобы завершить выполнение программы, нажмите любую клавишу.

Задание 2 – Ввод размеров матриц

Для задания исходных данных последовательного алгоритма матричного умножения реализуем функцию *ProcessInitialization*. Эта функция предназначена для определения размера матриц, выделения памяти для исходных матриц *pAMatrix* и *pBMatrix*, и матрицы-результата умножения *pCMatrix*, а также для задания значений элементов исходных матриц. Значит, функция должна иметь следующий интерфейс:

```
// Function for memory allocation and initialization of
matrix elements
void ProcessInitialization (double* &pAMatrix, double*
&pBMatrix,
    double* &pCMatrix, 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 initialization of
matrices' elements
void ProcessInitialization (double* &pAMatrix, double*
&pBMatrix,
 double* &pCMatrix, int &Size) {
 // Setting the size of matricies
 printf("\nEnter size of matricies: ");
 scanf("%d", &Size);
 printf("\nChosen matricies' 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. 2.4).

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

```
  void main() {
 double* pAMatrix; // The first argument of matrix multi-
plication
 double* pBMatrix; // The second argument of matrix multi-
plication
 double* pCMatrix; // The result matrix
 int Size; // Size of matricies
 time_t start, finish;
 double duration;
 printf ("Serial matrix multiplication program\n");
```

```
ProcessInitialization(pAMatrix, pBMatrix, pCMatrix, Size);
getch();
```

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

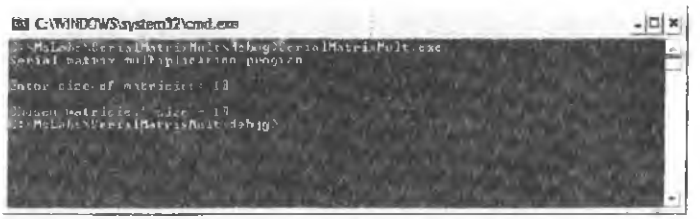

Рис. 2.4. Задание размера объекта

Как и в задаче 1, выполним контроль правильности ввода. Организуем проверку размера матриц 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 matricies
do {
 printf("\nEnter size of matricies: ");
 scanf("%d", &Size);
 printf("\nChosen matricies 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 initialization of
matrices' elements
void ProcessInitialization (double* &pAMatrix, double*
&pBMatrix,
 double* &pCMatrix, int &Size) {
```

```
 // Setting the size of matricies
 do {
 <…>
}
while (Size <= 0);

// Memory allocation
pAMatrix = new double [Size*Size];
pBMatrix = new double [Size*Size];
pCMatrix = new double [Size*Size];
}
```

Далее необходимо задать значения всех элементов исходных объектов: матриц *pAMatrix*, *pBMatrix* и *pCMatrix*. Значения элементов результирующей матрицы до выполнения матричного умножения равны 0. Для задания значений элементов матриц *A* и *B* реализуем еще одну функцию *DummyDataInitialization*. Интерфейс и реализация этой функции представлены ниже:

```
// Function for simple initialization of matrix elements
void  DummyDataInitialization(double*  pAMatrix,  double*
pBMatrix, int Size){
 int i, j; // Loop variables
 for (i=0; i<Size; i++) {
 for (j=0; j<Size; j++) {
 pAMatrix[i*Size+j] = 1;
 pBMatrix[i*Size+j] = 1;
 }
 }
}
```

Как видно из представленного фрагмента кода, данная функция осуществляет задание элементов матриц простым образом: значения всех элементов матриц равны 1. То есть в случае, когда пользователь выбрал размер объектов, равный 4, будут определены следующие матрица и вектор:

$$pAMatrix = \begin{pmatrix} 1111 \\ 1111 \\ 1111 \\ 1111 \end{pmatrix}, pBMatrix = \begin{pmatrix} 1111 \\ 1111 \\ 1111 \\ 1111 \end{pmatrix}$$

(задание данных при помощи датчика случайных чисел будет рассмотрено в задании 6).

Вызов функции *DummyDataInitialization* и процедуру заполнения результирующей матрицы нулями необходимо выполнить после выделения памяти внутри функции *ProcessInitialization*:

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

// Memory allocation
<…>

// Initialization of matrix elements
DummyDataInitialization(pAMatrix, pBMatrix, Size);
for (int i=0; i<Size*Size; i++) {
pCMatrix[i] = 0;
}
}
```

Для контроля ввода данных воспользуемся функцией форматированного вывода объектов *PrintMatrix*, которая была разработана при выполнении задачи 1 и текст которой уже имеется в проекте (подробнее о функции *PrintMatrix* см. задание 3 упражнение 2 задачи 1). Добавим вызов этой функций для печати объектов *pAMatrix* и *pBMatrix* в основную функцию приложения:

```
// Memory allocation and initialization of matrix elements
ProcessInitialization(pAMatrix, pBMatrix, pCMatrix, Size);

// Matrix output
printf ("Initial A Matrix \n");
PrintMatrix(pAMatrix, Size, Size);
printf("Initial B Matrix \n");
PrintMatrix(pBMatrix, Size, Size);
```

Скомпилируйте и запустите приложение. Убедитесь в том, что ввод данных происходит по описанным правилам (рис. 2.5). Выполните несколько запусков приложения, задавайте различные размеры матриц.

Рис. 2.5. Результат работы программы при завершении задания 3

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 матриц *pAMatrix* и *pBMatrix*, а также для хранения матрицы - результата умножения *pCMatrix*. Следовательно, эти объекты необходимо передать в функцию *ProcessTermination* в качестве аргументов:

```
// Function for computational process termination
void ProcessTermination (double* pAMatrix, double* pBMa-
trix,
 double* pCMatrix) {
 delete [] pAMatrix;
 delete [] pBMatrix;
 delete [] pCMatrix;
}
```

Вызов функции *ProcessTermination* необходимо выполнить перед завершением той части программы, которая выполняет умножение матрицы на вектор:

```
// Memory allocation and initialization of matrix elements
ProcessInitialization(pAMatrix, pBMatrix, pCMatrix, Size);

// Matrix output
printf ("Initial A Matrix \n");
PrintMatrix(pAMatrix, Size, Size);
printf("Initial B Matrix \n");
PrintMatrix(pBMatrix, Size, Size);

// Computational process termination
```

```
ProcessTermination(pAMatrix, pBMatrix, pCMatrix);
```

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

Задание 5 – Реализация матричного умножения

Выполним теперь разработку основной вычислительной части программы. Для выполнения умножения матриц *SerialResultCalculation*, которая принимает на вход исходные матрицы *pAMatrix* и *pBMatrix*, размер этих матриц *Size*, а также указатель на результатирующую матрицу *pCMatrix*.

В соответствии с алгоритмом, изложенным в упражнении 1, код этой функции должен иметь следующий вид:

```
// Function for matrix multiplication
void SerialResultCalculation(double* pAMatrix, double*
pBMatrix,
double* pCMatrix, int Size) { int i, j, k; // Loop varia-
bles
for (i=0; i<Size; i++) {
for (j=0; j<Size; j++) {
for (k=0; k<Size; k++) {
pCMatrix[i*Size+j]+= pAMa-
trix[i*Size+k]*pBMatrix[k*Size+j];
}
}
}
}
```

Выполним вызов функции вычисления матричного произведения из основной программы. Для контроля правильности выполнения умножения распечатаем результирующую матрицу:

```
// Memory allocation and initialization of matrix elements
ProcessInitialization(pAMatrix, pBMatrix, pCMatrix, Size);

// Matrix output
printf ("Initial A Matrix \n");
PrintMatrix(pAMatrix, Size, Size);
printf("Initial B Matrix \n");
PrintMatrix(pBMatrix, Size, Size);

// Matrix multiplication
SerialResultCalculation(pAMatrix, pBMatrix, pCMatrix,
Size);

// Printing the result matrix
printf ("\n Result Matrix: \n");
PrintMatrix(pCMatrix, Size, Size);
// Computational process termination
```

```
ProcessTermination(pAMatrix, pBMatrix, pCMatrix);
```

Скомпилируйте и запустите приложение. Проанализируйте результат работы алгоритма умножения матриц. Если алгоритм реализован правильно, то в результате должна быть получена матрица, значения всех элементов которой равны порядку этой матрицы (рис. 2.6).

$$\begin{pmatrix} 111 \\ 111 \\ 111 \end{pmatrix} \times \begin{pmatrix} 111 \\ 111 \\ 111 \end{pmatrix} = \begin{pmatrix} 333 \\ 333 \\ 333 \end{pmatrix}$$

Рис. 2.6. Результат матричного умножения

Проведите несколько вычислительных экспериментов, изменяя размеры объектов.

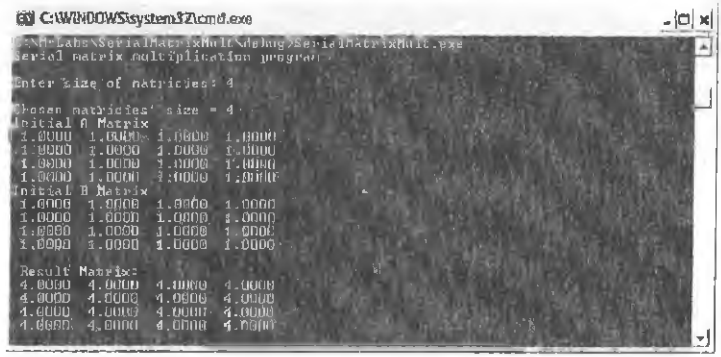

Рис. 2.7. Результат выполнения матрично-векторного умножения

Задание 6 – 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 объектов. Задавать элементы больших матриц и векторов будем при помощи датчика случайных чисел. Для этого реализуем еще одну функцию задания элементов *RandomDataInitialization* (датчик случайных чисел инициализируется текущим значением времени):

```
// Function for random initialization of matrix elements
```

```
void RandomDataInitialization (double* pAMatrix, double*
pBMatrix,
 int Size) {
 int i, j; // Loop variables
 srand(unsigned(clock()));
 for (i=0; i<Size; i++)
 for (j=0; j<Size; j++) {
 pAMatrix[i*Size+j] = rand()/double(1000);
 pBMatrix[i*Size+j] = rand()/double(1000);
 }
}
```

Будем вызывать эту функцию вместо ранее разработанной функции *DummyDataInitialization*, которая генерировала такие данные, что можно было легко проверить правильность работы алгоритма:

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

 // Random initialization of matrix elements
 RandomDataInitialization(pAMatrix, pBMatrix, Size);
 for (int i=0; i<Size*Size; i++) {
 pCMatrix[i] = 0;
 }
}
```

Скомпилируйте и запустите приложение. Убедитесь в том, что данные генерируются случайным образом.

Для определения времени добавьте в получившуюся программу вызовы функций, позволяющие узнать время работы программы или её части. Мы, как и ранее, будем пользоваться функцией:

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

Добавим в программный код вычисление и вывод времени непосредственного выполнения умножения матрицы на вектор, для этого поставим замеры времени до и после вызова функции *SerialResultCalculation*:

```
// Matrix multiplication
start = clock();
SerialResultCalculation(pAMatrix,    pBMatrix,    pCMatrix,
Size);

finish = clock();
duration = (finish-start)/double(CLOCKS_PER_SEC);
// Printing the result matrix
printf ("\n Result Matrix: \n");
PrintMatrix(pCMatrix, Size, Size);

// Printing the time spent by matrix multiplication
printf("\n Time of execution: %f\n", duration);
```

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

Таблица 2.1. Время выполнения последовательного алгоритма умножения матриц

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

Согласно алгоритму вычисления произведения матриц, изложенному в упражнении 1, получение результирующей матрицы предполагает повторение *Size*×*Size* однотипных операций по умножению строк матрицы *pAMatrix* и столбцов матрицы *pBMatrix*. Каждая такая операция включает умножение элементов строки и столбца (*Size* операций) и последующее суммирование полученных произведений (*Size-1* операций). Как результат, общее время матричного умножения можно определить при помощи соотношения:

$$T_1 = Size^2(2Size - 1)\tau \qquad (2.3)$$

где τ есть время выполнения одной базовой вычислительной операции.

Выполним таблицу сравнения реального времени выполнения со временем, которое может быть получено по формуле (2.3). Для вычисления времени выполнения одной операции τ, как и при выполнении задачи 1, выберем один из экспериментов в качестве образца, время выполнения этого эксперимента поделим на число выполненных операций (число операций может быть вычислено по формуле (2.3)). 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 матриц, участвующих в умножении (см. задачу 1), поэтому при выборе эксперимента для образца следует ориентироваться на некоторый средний случай.

Вычислите теоретическое время выполнения матричного умножения. Результаты занесите в таблицу:

Таблица 2.2. Сравнение реального времени выполнения последовательного алгоритма умножения матриц 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 3 – Разработка параллельного алгоритма матричного умножения

При построении параллельных способов выполнения матричного умножения наряду с рассмотрением матриц в виде наборов строк и столбцов широко используется блочное представление

матриц. Выполним более подробное рассмотрение данного способа организации вычислений.

Определение подзадач

Блочная схема разбиения матриц подробно рассмотрена в подразделе 7.2 лекционного материала и в упражнении 3 задачи 1. При таком способе разделения данных исходные матрицы A, B и результирующая матрица C представляются в виде наборов блоков. Для более простого изложения следующего материала будем предполагать далее, что все матрицы являются квадратными размера $n \times n$, количество блоков по горизонтали и вертикали являются одинаковым и равным q (т.е. размер всех блоков равен $k \times k$, $k=n/q$). При таком представлении данных операция матричного умножения матриц A и B в блочном виде может быть представлена в виде:

$$\begin{pmatrix} A_{00} A_{01} \ldots A_{0q-1} \\ \ldots \\ A_{q-10} A_{q-11} \ldots A_{q-1q-1} \end{pmatrix} \times \begin{pmatrix} B_{00} B_{01} \ldots B_{0q-1} \\ \ldots \\ B_{q-10} B_{q-11} \ldots B_{q-1q-1} \end{pmatrix} = \begin{pmatrix} C_{00} C_{01} \ldots C_{0q-1} \\ \ldots \\ C_{q-10} C_{q-11} \ldots C_{q-1q-1} \end{pmatrix}$$

где каждый блок C_{ij} матрицы C определяется в соответствии с выражением

$$C_{ij} = \sum_{s=0}^{q-1} A_{is} B_{sj}.$$

При блочном разбиении данных для определения базовых подзадач естественным представляется взять за основу вычисления, выполняемые над матричными блоками. С учетом сказанного определим базовую подзадачу как процедуру вычисления всех элементов одного из блоков матрицы C.

Для выполнения всех необходимых вычислений базовым подзадачам должны быть доступны соответствующие наборы строк матрицы A и столбцов матрицы B. Размещение всех требуемых данных в каждой подзадаче неизбежно приведет к дублированию и к значительному росту объема используемой памяти. Как результат, вычисления должны быть организованы таким образом, чтобы в каждый текущий момент времени подзадачи содержали лишь часть необходимых для проведения расчетов данных, а доступ к остальной части данных обеспечивался бы

при помощи передачи сообщений. Один из возможных подходов – *алгоритм Фокса* (*the Fox algorithm*) – подробно рассмотрим в данном упражнении.

Выделение информационных зависимостей

Итак, за основу параллельных вычислений для матричного умножения при блочном разделении данных принят подход, при котором базовые подзадачи отвечают за вычисления отдельных блоков матрицы C и при этом в подзадачах на каждой итерации расчетов располагаются только по одному блоку исходных матриц A и B. Для нумерации подзадач будем использовать индексы размещаемых в подзадачах блоков матрицы C, т.е. подзадача (i,j) отвечает за вычисление блока C_{ij} – тем самым, набор подзадач образует квадратную решетку, соответствующую структуре блочного представления матрицы C.

В соответствии с алгоритмом Фокса в ходе вычислений на каждой базовой подзадаче (i,j) располагается четыре матричных блока:

– блок C_{ij} матрицы C, вычисляемый подзадачей;

– блок A_{ij} матрицы A, размещаемый в подзадаче перед началом вычислений;

– блоки A'_{ij}, B'_{ij} матриц A и B, получаемые подзадачей в ходе выполнения вычислений.

Выполнение параллельного метода включает:

• этап инициализации, на котором каждой подзадаче (i,j) передаются блоки A_{ij}, B_{ij} и обнуляются блоки C_{ij} на всех подзадачах;

• этап вычислений, в рамках которого на каждой итерации l, $0 \le l < q$, осуществляются следующие операции:

– для каждой строки i, $0 \le i < q$, блок A_{ij} подзадачи (i,j) пересылается на все подзадачи той же строки i решетки; индекс j, определяющий положение подзадачи в строке, вычисляется в соответствии с выражением

$$j = (i + l) \bmod q, \qquad (2.4)$$

где *mod* есть операция получения остатка от целочисленного деления;

– полученные в результаты пересылок блоки A'_{ij}, B'_{ij} каждой подзадачи (i,j) перемножаются и прибавляются к блоку C_{ij}

$$C_{ij} = C_{ij} + A'_{ij} \times B'_{ij}$$

– блоки B'_{ij} каждой подзадачи *(i,j)* пересылаются подзадачам, являющимися соседями сверху в столбцах решетки подзадач (блоки подзадач из первой строки решетки пересылаются подзадачам последней строки решетки).

Для пояснения приведенных правил параллельного метода на рис. 2.8 приведено состояние блоков в каждой подзадаче в ходе выполнения итераций этапа вычислений (для решетки подзадач *2×2*).

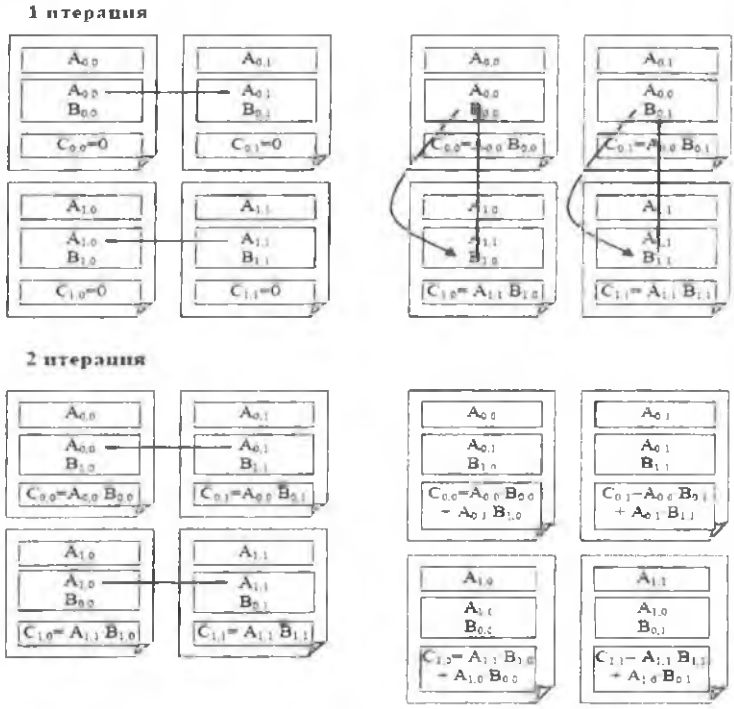

Рис. 2.8. Состояние блоков в каждой подзадаче в ходе выполнения итераций алгоритма Фокса

Масштабирование и распределение подзадач по процессорам

В рассмотренной схеме параллельных вычислений количество блоков может варьироваться в зависимости от выбора размера блоков – эти размеры могут быть подобраны таким образом, чтобы общее количество базовых подзадач совпадало с числом процессоров p. Так, например, в наиболее простом случае, когда число процессоров представимо в виде $p=\delta^2$ (т.е. является полным квадратом) можно выбрать количество блоков в матрицах по вертикали и горизонтали равным δ (т.е. $q=\delta$). Такой способ определения количества блоков приводит к тому, что объем вычислений в каждой подзадаче является одинаковым и, тем самым, достигается полная балансировка вычислительной нагрузки между процессорами. В более общем случае при произвольных количестве процессоров и размеров матриц балансировка вычислений может отличаться от абсолютно одинаковой, но, тем не менее, при надлежащем выборе параметров может быть распределена между процессорами равномерно в рамках требуемой точности.

Для эффективного выполнения алгоритма Фокса, в котором базовые подзадачи представлены в виде квадратной решетки и в ходе вычислений выполняются операции передачи блоков по строкам и столбцам решетки подзадач, наиболее адекватным решением является организация множества имеющихся процессоров также в виде квадратной решетки. В этом случае можно осуществить непосредственное отображение набора подзадач на множество процессоров – базовую подзадачу (i,j) следует располагать на процессоре Pi,j. Необходимая структура сети передачи данных может быть обеспечена на физическом уровне, если топология вычислительной системы имеет вид решетки или полного графа.

Упражнение 4 – Реализация параллельного алгоритма умножения матриц

При выполнении этого упражнения Вам будет предложено разработать параллельный алгоритм Фокса для матричного умножения. При работе с этим упражнением Вы

- На практике познакомитесь со схемой организации матричных вычислений на основе блочного разделения данных,
- Получите опыт разработки более сложных параллельных

программ,

• Познакомитесь с процедурой создания в MPI коммуникаторов и виртуальных топологий.

Задание 1 – Открытие проекта ParallelMatrixMult

Откройте проект **ParallelMatrixMult**, 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\ParallelMatrixMult**,

• Дважды щелкните на файле **ParallelMatrixMult.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 **ParallelMM.cpp**, как это показано на рисунке 2.9. После этих действий код, который вам предстоит модифицировать, будет открыт в рабочей области Visual Studio.

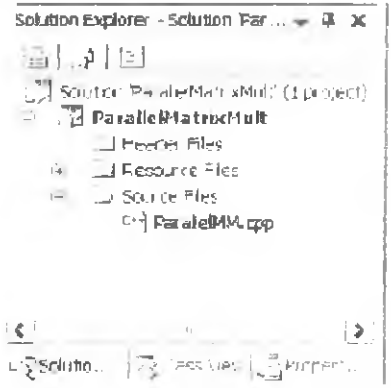

Рис. 2.9. Открытие проекта ParallelMatrixMult

В файле **ParallelMM.cpp** расположена главная функция (*main*) будущего параллельного приложения, которая содержит строки подключения библиотек, объявления необходимых переменных, вызовы функций инициализации и остановки среды вы-

ношения MPI-программ, функции для определения числа доступных процессов и рангов процессов:

```
int ProcNum = 0; // Number of available processes
int ProcRank = 0; // Rank of current process

void main(int argc, char* argv[]) {
double* pAMatrix; // The first argument of matrix multi-
plication
double* pBMatrix; // The second argument of matrix multi-
plication
double* pCMatrix; // The result matrix
int Size; // Size of matricies
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 matrix multiplication program\n");

MPI_Finalize();
}
```

Заметим, что переменные *ProcNum* и *ProcRank*, как и при разработке параллельного алгоритма умножения матрицы на вектор (см. описание задачи 1) были объявлены глобальными.

Также в файле **ParallelMM.cpp** расположены функции, перенесенные сюда из проекта, содержащего последовательный алгоритм умножения матриц: *DummyDataInitialization*, *RandomDataInitialization*, *SerialResultCalculation*, *PrintMatrix* (подробно о назначении этих функций рассказывается в упражнении 2 данной задачи). Эти функции можно будет использовать и в параллельной программе. 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 matrix multiplication program".

Задание 2 – Создание виртуальной декартовой топологии

Согласно схеме параллельных вычислений, описанной в упражнении 3, для эффективного выполнения алгоритма Фокса необходимо организовать доступные процессы MPI-программы в

виртуальную топологию в виде двумерной квадратной решетки. Это возможно только в том случае, когда число доступных процессов является полным квадратом.

Перед тем, как приступить к выполнению параллельного алгоритма проверим, является ли число доступных процессоров полным квадратом: $ProcNum = GridSize \times GridSize$. В случае, когда это условие не выполняется, выведем диагностическое сообщение. Продолжим выполнение приложения только в том случае, когда условие выполняется.

Назовем величину *GridSize* размером решетки. Эта величина будет использоваться при разделении и сборе данных, а также при выполнении итераций алгоритма Фокса. Объявим соответствующую глобальную переменную и определим ее значение.

```
int ProcNum = 0; // Number of available processes
int ProcRank = 0; // Rank of current process
int GridSize; // Size of virtual processor grid

void main(int argc, char* argv[]) {
<...>
 MPI_Init(&argc, &argv);
 MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
 MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);

 GridSize = sqrt((double)ProcNum);
 if (ProcNum != GridSize*GridSize) {
 if (ProcRank == 0) {

printf ("Number of processes must be a perfect square \n");
 }
 }
 else {
 if (ProcRank == 0)
 printf("Parallel matrix multiplication program\n");
 // Place the code of the parallel Fox algorithm here
 }
 MPI_Finalize();
}
```

Реализуем функцию *CreateGridCommunicators*, которая создаст коммуникатор в виде двумерной квадратной решетки, определит координаты каждого процесса в этой решетке, а также создаст коммуникаторы отдельно для каждой строки и каждого столбца.

```
// Creation of two-dimensional grid communicator and
// communicators for each row and each column of the grid
void CreateGridCommunicators();
```

Для непосредственного создания декартовой топологии (решетки) в MPI предназначена функция:

int **MPI_Cart_create**(MPI_Comm **oldcomm**, int **ndims**, int *dims,

int *periods, int reorder, MPI_Comm *cartcomm),

где:

- **oldcomm** - исходный коммуникатор,
- **ndims** - размерность декартовой решетки,
- **dims** - массив длины ndims, задает количество процессов в каждом измерении решетки,
- **periods** - массив длины ndims, определяет, является ли решетка периодической вдоль каждого измерения,
- **reorder** - параметр допустимости изменения нумерации процессов,
- **cartcomm** – создаваемый коммуникатор с декартовой топологией процессов.

Итак, для создания декартовой топологии нужно определить два массива: первый *DimSize* определяет размерности решетки, а второй *Periodic* определяет, является ли решетка периодической вдоль каждого измерения. Поскольку нам необходимо создать двумерную квадратную решетку, то оба элемента *DimSize* должны быть определены следующим образом: $DimSize[0] = DimSize[1] = \sqrt{\Pr ocNum}$. Согласно схеме параллельных вычислений (упражнение 3) нам предстоит осуществлять циклический сдвиг вдоль столбцов процессорной решетки, следовательно, второе измерение декартовой топологии должно обязательно быть периодическим. В результате выполнения функции *MPI_Cart_create* новый коммуникатор сохраняется в переменной *cartcomm*. Значит, нужно объявить переменную для хранения нового коммуникатора и передать ее в качестве аргумента функции *MPI_Cart_create*. Поскольку коммуникатор в виде решетки будет широко использоваться во всех функциях параллельного приложения, объявим соответствующую переменную как глобальную. В библиотеке MPI все коммуникаторы имеют тип *MPI_Comm*.

Добавим в тело функции *CreateGridCommunicators* вызов функции создания решетки:

```
int ProcNum = 0; // Number of available processes
int ProcRank = 0; // Rank of current process
int GridSize; // Size of virtual processor grid
```

```
MPI_Comm GridComm; // Grid communicator
<…>
// Creation of two-dimensional grid communicator and
// communicators for each row and each column of the grid
void CreateGridCommunicators() {
 int DimSize[2]; // Number of processes in each dimension
of the grid
 int Periodic[2]; // =1, if the grid dimension should be
periodic

 DimSize[0] = GridSize;
 DimSize[1] = GridSize;

 Periodic[0] = 1;
 Periodic[1] = 1;

 // Determination of the size of the virtual grid
 MPI_Dims_create(ProcNum, 2, DimSize);
 // Creation of the Cartesian communicator
 MPI_Cart_create(MPI_COMM_WORLD, 2, DimSize, Periodic, 1,
&GridComm);
}
```

Для определения декартовых координат процесса по его рангу можно воспользоваться функцией:

```
int MPI_Card_coords(MPI_Comm comm,int rank,int ndims,int
*coords),
```

где:

- comm - коммуникатор с топологией решетки,
- rank - ранг процесса, для которого определяются декартовы координаты,
- ndims - размерность решетки,
- coords - возвращаемые функцией декартовы координаты процесса.

Поскольку нами была создана двумерная решетка, каждый процесс в этой решетке имеет две координаты, они соответствуют номеру строки и столбца, на пересечении которых расположен данный процесс. Объявим глобальную переменную – массив для хранения координат каждого процесса и определим эти координаты при помощи функции *MPI_Cart_coords*:

```
int GridSize; // Size of virtual processor grid
MPI_Comm GridComm; // Grid communicator
int GridCoords[2]; // Coordinates of current processor in
grid
<…>
// Creation of two-dimensional grid communicator and
// communicators for each row and each column of the grid
void CreateGridCommunicators() {
 <…>
```

```
 // Creation of the Cartesian communicator
 MPI_Cart_create(MPI_COMM_WORLD, 2, DimSize, Periodic, 1,
&GridComm);

 // Determination of the cartesian coordinates for every
process
 MPI_Cart_coords(GridComm, ProcRank, 2, GridCoords);
}
```

Теперь создадим коммуникаторы для каждой строки и каждого столбца процессорной решетки. Для этого в библиотеке MPI реализованы функции, позволяющие разделить решетку на подрешетки (подробнее об использовании функции *MPI_Cart_sub* см. раздел 4 "Параллельное программирование на основе MPI" лекционных материалов):

```
int MPI_Card_sub(MPI_Comm comm, int *subdims, MPI_Comm
*newcomm),
```

где:

- **comm** - исходный коммуникатор с топологией решетки,
- **subdims** – массив для указания, какие измерения должны остаться в создаваемой подрешетке,
- **newcomm** - создаваемый коммуникатор с подрешеткой.

Объявим коммуникаторы для строки и столбца как глобальные переменные и разделим уже созданный коммуникатор *GridComm*:

```
MPI_Comm GridComm; // Grid communicator
MPI_Comm ColComm; // Column communicator
MPI_Comm RowComm; // Row communicator

// Creation of two-dimensional grid communicator and
// communicators for each row and each column of the grid
void CreateGridCommunicators() {
 int DimSize[2]; // Number of processes in each dimension
of the grid
 int Periodic[2]; // =1, if the grid dimension should be
periodic
 int Subdims[2]; // =1, if the grid dimension should be
fixed


 // Determination of the cartesian coordinates for every
process
 MPI_Cart_coords(GridComm, ProcRank, 2, GridCoords);

 // Creating communicators for rows
 Subdims[0] = 0; // Dimension is fixed
```

```
 Subdims[1] = 1; // Dimension belong to the subgrid
 MPI_Cart_sub(GridComm, Subdims, &RowComm);
// Creating communicators for columns
 Subdims[0] = 1; // Dimension belong to the subgrid
 Subdims[1] = 0; // Dimension is fixed
 MPI_Cart_sub(GridComm, Subdims, &ColComm);
}
```

Вызовем функцию *CreateGridCommunicators* из основной функции параллельного приложения:

```
void main(int argc, char* argv[]) {
 double* pAMatrix; // The first argument of matrix multi-
plication
 double* pBMatrix; // The second argument of matrix multi-
plication
 double* pCMatrix; // The result matrix
 int Size; // Size of matricies
 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);

 GridSize = sqrt((double)ProcNum);
 if (ProcNum != GridSize*GridSize) {
 if (ProcRank == 0) {
 printf ("Number of processes must be a perfect square
\n");
 }
 }
 else {
 if (ProcRank == 0)
 printf("Parallel matrix multiplication program\n");

 // Grid communicator creating
 CreateGridCommunicators();
 }

 MPI_Finalize();
}
```

Скомпилируйте приложение. Если в процессе компиляции были обнаружены ошибки, исправьте их, сверяя свой программный код с кодом, представленным в данном пособии. Запустите приложение несколько раз, изменяя количество доступных процессов. Убедитесь в том, что в случае, когда доступное число процессов не является полным квадратом, выдается диагностическое сообщение и приложение завершает работу.

Задание 3 – Определение размеров объектов и ввод исходных данных

На следующем этапе разработки параллельного приложения необходимо задать размеры матриц и выделить память для хранения исходных матриц и их блоков. Согласно схеме параллельных вычислений, на каждом процессе в каждый момент времени располагается четыре матричных блока: два блока матрицы A, блок матрицы B и блок результирующей матрицы C (см. упражнение 3). Определим переменные для хранения матричных блоков и размера этих блоков:

```
void main(int argc, char* argv[]) {
 double* pAMatrix; // The first argument of matrix multi-
plication
 double* pBMatrix; // The second argument of matrix multi-
plication
 double* pCMatrix; // The result matrix
 int Size; // Size of matricies
 int BlockSize; // Sizes of matrix blocks on current pro-
cess
 double *pMatrixAblock; // Initial block of matrix A on
current process
 double *pAblock; // Current block of matrix A on current
process
 double *pBblock; // Current block of matrix B on current
process
 double *pCblock; // Block of result matrix C on current
process
```

Для определения размеров матриц и матричных блоков, выделения памяти для их хранения и определения элементов исходных матриц реализуем функцию *ProcessInitialization*.

```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pAMatrix, double*
&pBMatrix,
 double* &pCMatrix, double* &pAblock, double* &pBblock,
double* &pCblock,
 double* &pMatrixAblock, int &Size, int &BlockSize )
```

Начнем с определения размеров. Для простоты, как и ранее, будем предполагаем, что все матрицы, участвующие в умножении, квадратные порядка $Size \times Size$. Размер *Size* должен быть таким, чтобы матрицы можно было разделить между процессами равными квадратными блоками, то есть размер *Size* должен быть кратен размеру процессорной решетки *GridSize*.

Для определения размера, как и при выполнении задачи 1, ор-

ганизуем диалог с пользователем. Если пользователь вводит некорректное число, ему предлагается повторить ввод. Диалог осуществляется только на *ведущем процессе*. Напомним, что ведущим процессом обычно является процесс, который имеет нулевой ранг в рамках коммуникатора *MPI_COMM_WORLD*. Когда размеры матриц корректно определены, значение переменной *Size* рассылается на все процессы:

```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pAMatrix, double*
&pBMatrix,
 double* &pCMatrix, double* &pAblock, double* &pBblock,
double* pCblock,
 double* &pTemporaryAblock, int &Size, int &BlockSize ) {
 if (ProcRank == 0) {
 do {
 printf("\nEnter size of the initial objects: ");
 scanf("%d", &Size);
 if (Size%GridSize != 0) {
 printf ("Size of matrices must be divisible by the grid
size! \n");
 }
 } while (Size%GridSize != 0);
 }
 MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);
}
```

После того, как размеры матриц определены, появляется возможность определить размер матричных блоков, а также выделить память для хранения исходных матриц, матрицы результата, матричных блоков (исходные матрицы существуют только на ведущем процессе):

```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pAMatrix, double*
&pBMatrix,
 double* &pCMatrix, double* &pAblock, double* &pBblock,
double* &pCblock,
 double* &pMatrixAblock, int &Size, int &BlockSize ) {
 <…>
 MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);

 BlockSize = Size/GridSize;

 pAblock = new double [BlockSize*BlockSize];
 pBblock = new double [BlockSize*BlockSize];
 pCblock = new double [BlockSize*BlockSize];
 pMatrixAblock = new double [BlockSize*BlockSize];

 if (ProcRank == 0) {
```

```
pAMatrix = new double [Size*Size];
pBMatrix = new double [Size*Size];
pCMatrix = new double [Size*Size];
}
```

Для определения элементов исходных матриц будем использовать функцию *DummyDataInitialization*, которая была разработана нами при реализации последовательного алгоритма матричного умножения:

```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pAMatrix, double*
&pBMatrix,
double* &pCMatrix, double* &pAblock, double* &pBblock,
double* &pCblock,
double* &pMatrixAblock, int &Size, int &BlockSize ) {

if (ProcRank == 0) {
pAMatrix = new double [Size*Size];
pBMatrix = new double [Size*Size];
pCMatrix = new double [Size*Size];
DummyDataInitialization(pAMatrix, pBMatrix, Size);
}
}
```

Блок результирующей матрицы pCblock служит для суммирования результатов умножения блоков матриц A и B. Для того, чтобы суммы накапливались правильно, необходимо первоначально обнулить все его элементы:

```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pAMatrix, double*
&pBMatrix,
double* &pCMatrix, double* &pAblock, double* &pBblock,
double* pCblock,
double* &pMatrixAblock, int &Size, int &BlockSize ) {

if (ProcRank == 0) {

}
for (int i=0; i<BlockSize*BlockSize; i++) {
pCblock[i] = 0;
}
}
```

Вызовем функцию *ProcessInitialization* из основной функции параллельного приложения. Для контроля правильности ввода исходных данных воспользуемся функцией форматированного вывода матриц *PrintMatrix*: распечатаем исходные матрицы A и B на ведущем процессе.

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

```
 <...>
// Memory allocation and initialization of matrix elements
 ProcessInitialization ( pAMatrix, pBMatrix, pCMatrix,
pAblock, pBblock,
 pCblock, pMatrixAblock, Size, BlockSize );
 if (ProcRank == 0) {
 printf("Initial matrix A \n");
 PrintMatrix(pAMatrix, Size, Size);
 printf("Initial matrix B \n");
 PrintMatrix(pBMatrix, Size, 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 исходных матриц. Если данные задаются верно, то все элементы исходных матриц должны быть приравнены 1 (рис. 2.10).

Рис. 2.10. Задание исходных данных

Задание 4 – 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 матриц *pAMatrix* и *pBMatrix* и память для хранения результирующей матрицы *pCMatrix*, на всех процессах выделялась память для хранения четырех матричных блоков *pMatrixAblock*, *pAblock*, *pBblock*, *pCblock*. Все эти объекты необходимо передать в функцию *ProcessTermination* в качестве аргументов:

```
// Function for computational process termination
void ProcessTermination (double* pAMatrix, double*
pBMatrix,
 double* pCMatrix, double* pAblock, double* pBblock,
double* pCblock,
 double* pMatrixAblock) {
 if (ProcRank == 0) {
 delete [] pAMatrix;
 delete [] pBMatrix;
 delete [] pCMatrix;
 }
 delete [] pAblock;
 delete [] pBblock;
 delete [] pCblock;
 delete [] pMatrixAblock;
}
```

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

```
 // Process termination
 ProcessTermination(pAMatrix, pBMatrix, pCMatrix,
pAblock, pBblock,
 pCblock, pMatrixAblock);
 }
 MPI_Finalize();
}
```

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

Задание 5 – Распределение данных между процессами

Согласно схеме параллельных вычислений, исходные матрицы, которые нужно перемножить, расположены на ведущем процессе. Ведущий процесс – процесс с рангом 0, расположен в левом верхнем углу процессорной решетки.

Нужно распределить матрицы поблочно между процессами так, чтобы блоки *Aij* и *Bij* были помещены на процессе, расположенном на пересечении *i*-ой строки и *j*-ого столбца процессорной решетки. Матрицы и матричные блоки хранятся в одномерных массивах построчно. Блок матрицы не хранится непрерывной последовательностью элементов в массиве хранения матрицы, следовательно для распределения по блокам невозможно выполнить с использованием стандартных типов данных библиотеки MPI.

Для организации передачи блоков в рамках одной и той же

коммуникационной операции можно сформировать средствами MPI производный тип данных. Оставив такой подход для самостоятельной проработки, применим в данной задаче следующую двухэтапную схему распределения данных. На первом этапе матрица разделяется на горизонтальные полосы, каждая из которых содержит *BlockSize* строк. Эти полосы распределяются на процессы, составляющие нулевой столбец процессорной решетки (рис. 2.11).

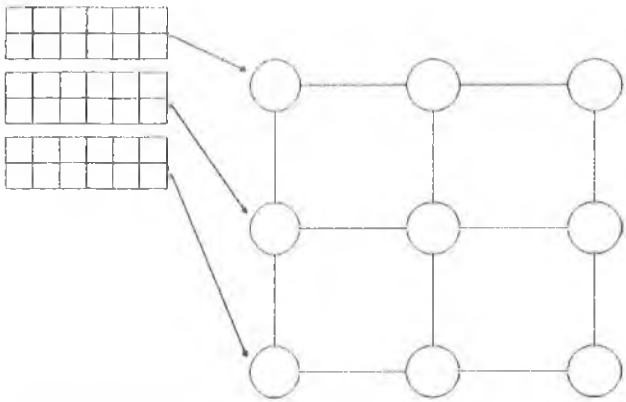

Рис. 2.11. Первый этап распределения данных

Далее каждая полоса разделяется на блоки между процессами, составляющими строки процессорной решетки. Заметим, что распределение полосы на блоки будет осуществляться последовательно через распределение строк полосы при помощи функции *MPI_Scatter* (рис. 2.12).

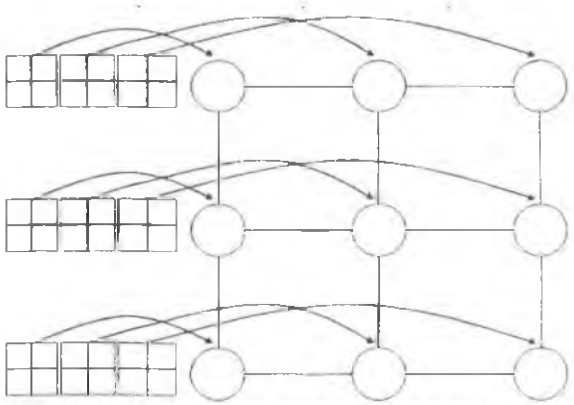

Рис. 2.12. Второй этап разделения данных

Для блочного разделения матрицы между процессами процессорной решетки реализуем функцию *CheckerboardMatrixScatter*.

```
// Function for checkerboard matrix decomposition
void CheckerboardMatrixScatter(double* pMatrix, dou-
ble* pMatrixBlock,
  int Size, int BlockSize);
```

Эта функция принимает в качестве аргументов матрицу, которая хранится на ведущем процессе *pMatrix*, указатель на буфер хранения матричного блока на каждом из процессов параллельного приложения *pMatrixBlock*, размер матрицы *Size* и размер матричного блока *BlockSize*.

На первом этапе необходимо разделить матрицу горизонтальными полосами между процессами, составляющими нулевой столбец решетки процессов. Для этого воспользуемся функцией *MPI_Scatter* в рамках коммуникатора *ColComm*. Заметим, что в параллельном приложении ранее было создано *GridSize* коммуникаторов *ColComm*. Для того, чтобы определить именно тот коммуникатор, который соответствует нулевому столбцу процессорной решетки, воспользуемся значениями, записанными в массиве *GridCoords*. Функцию *MPI_Scatter* будем вызывать только в тех процессах, у которых значение второй координаты равно 0 (то есть процесс расположен в нулевом столбце).

```
// Function for checkerboard matrix decomposition
void CheckerboardMatrixScatter(double* pMatrix, dou-
ble* pMatrixBlock,
```

```
        int Size, int BlockSize) {
        double * pMatrixRow = new double [BlockSize*Size];
        if (GridCoords[1] == 0) {
        MPI_Scatter(pMatrix, BlockSize*Size, MPI_DOUBLE,
pMatrixRow,
        BlockSize*Size, MPI_DOUBLE, 0, ColComm);
        }
       }
```

Отметим, что для временного хранения горизонтальной полосы матрицы используется буфер *pMatrixRow*.

На втором этапе необходимо распределить каждую строчку горизонтальной полосы матрицы вдоль строк процессорной решетки. Снова воспользуемся функцией *MPI_Scatter* в рамках коммуникатора *RowComm*. После выполнения этих действий освободим выделенную память:

```
      // Function for checkerboard matrix decomposition
      void CheckerboardMatrixScatter(double* pMatrix, dou-
ble* pMatrixBlock,
        int Size, int BlockSize) {
        double * pMatrixRow = new double [BlockSize*Size];
        if (GridCoords[1] == 0) {
        MPI_Scatter(pMatrix, BlockSize*Size, MPI_DOUBLE, Ma-
trixRow,
        BlockSize*Size, MPI_DOUBLE, 0, ColComm);
        }

        for (int i=0; i<BlockSize; i++) {
        MPI_Scatter(&pMatrixRow[i*Size], BlockSize,
MPI_DOUBLE,
        &(pMatrixBlock[i*BlockSize]), BlockSize, MPI_DOUBLE,
0, RowComm);
        }
        delete [] pMatrixRow;

      }
```

Для выполнения алгоритма Фокса необходимо поблочно разделить матрицу *A* (блоки матрицы сохраняются в переменной *pMatrixABlock*) и матрицу *B* (блоки сохраняются в переменной pBblock).

Реализуем функцию *DataDistribution*, которая осуществляет разделение указанных матриц:

```
      // Function for data distribution among the processes
      void DataDistribution(double* pAMatrix, double* pBMa-
trix,
       double* pMatrixAblock, double* pBblock, int Size,
int BlockSize) {
```

```
    CheckerboardMatrixScatter(pAMatrix, pMatrixAblock,
Size, BlockSize);
    CheckerboardMatrixScatter(pBMatrix, pBblock, Size,
BlockSize);
  }
```

Вызовем функцию распределения данных из главной функции параллельного приложения.

```
  void main(int argc, char* argv[]) {
    <…>
    // Memory allocation and initialization of matrix
elements
    ProcessInitialization ( pAMatrix, pBMatrix, pCMa-
trix, pAblock, pBblock,
    pCblock, pMatrixAblock, Size, BlockSize );
    if (ProcRank == 0) {
    printf("Initial matrix A \n");
    PrintMatrix(pAMatrix, Size, Size);
    printf("Initial matrix B \n");
    PrintMatrix(pBMatrix, Size, Size);
    }

    // Data distribution among the processes
    DataDistribution(pAMatrix, pBMatrix, pMatrixAblock,
pBblock, Size,
    BlockSize);

    MPI_Finalize();
  }
```

Для контроля правильности распределения исходных данных снова воспользуемся отладочной печатью. Реализуем функцию, которая будет последовательно распечатывать содержимое матричного блока на всех процессах, назовем эту функцию *TestBlocks*.

```
  // Test printing of the matrix block
  void TestBlocks (double* pBlock, int BlockSize, char
*str[]) {
    MPI_Barrier(MPI_COMM_WORLD);
    if (ProcRank == 0) {
    printf("%s \n", str);
    }
    for (int i=0; i<ProcNum; i++) {
    if (ProcRank == i) {
    printf ("ProcRank = %d \n", ProcRank);
    PrintMatrix(pBlock, BlockSize, BlockSize);
    }
    MPI_Barrier(MPI_COMM_WORLD);
    }
  }
```

Вызовем функцию проверки распределения исходных данных из главной функции параллельной программы:

```
        // Data distribution among the processes
        DataDistribution(pAMatrix, pBMatrix, pMatrixABlock,
pBblock, Size,
        BlockSize);
        TestBlocks(pMatrixAblock, BlockSize, "Initial blocks
of matrix A");
        TestBlocks(pBblock, BlockSize, "Initial blocks of
matrix B");
        }
```

Скомпилируйте приложение. Если в процессе компиляции были обнаружены ошибки, исправьте их, сверяя свой код с программным кодом, представленным в данном пособии. Запустите приложение.

Убедитесь в том, что данные распределяются верно (рис. 2.13):

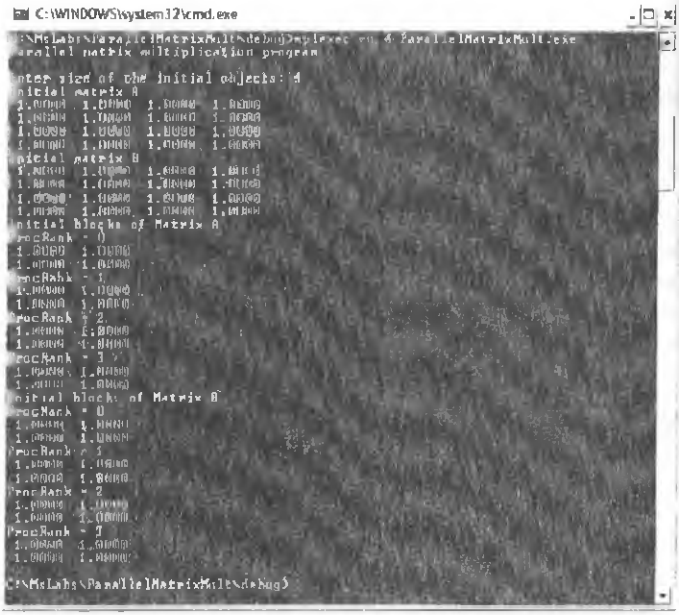

Рис. 2.13. Распределение исходных данных в случае, когда приложение запускается на 4 процессах и размер матриц равен 4

Измените задание исходных данных. Для определения эле-

вместо исходных матриц вместо функции *DummyDataInitialization* примените функцию *RandomDataInitialization*. Скомпилируйте и запустите приложение. Убедитесь в том, что матрицы корректно распределяются между процессами.

Задание 6 – Начало реализации параллельного алгоритма матричного умножения

За выполнение параллельного алгоритма Фокса матричного умножения отвечает функция *ParallelResultCalculation*. В качестве аргументов ей необходимо передать все матричные блоки и размер этих блоков:

```
void ParallelResultCalculation(double* pAblock, dou-
ble* pMatrixAblock,
  double* pBblock, double* pCblock, int BlockSize);
```

Согласно схеме параллельных вычислений, описанной в упражнении 3, для выполнения матричного умножения с помощью алгоритма Фокса необходимо выполнить *GridSize* итераций, каждая из которых состоит из выполнения трех действий:

- рассылка блока матрицы A по строке процессорной решетки (для выполнения этого шага реализуем функцию ABlockCommunication),
- выполнение умножения матричных блоков (для выполнения умножения матричных блоков

можно воспользоваться функцией SerialResultCalculation, которая была реализована в ходе разработки последовательного алгоритма умножения матриц),

- циклический сдвиг блоков матрицы B вдоль столбца процессорной решетки (функция BBlockCommunication).

Значит, код, выполняющий алгоритм Фокса матричного умножения, имеет следующий вид:

```
// Execution of the Fox method
void ParallelResultCalculation(double* pAblock, dou-
ble* pMatrixAblock,
  double* pBblock, double* pCblock, int BlockSize) {
  for (int iter = 0; iter < GridSize; iter ++) {
  // Sending blocks of matrix A to the process grid
rows
  ABlockCommunication(iter, pAblock, pMatrixAblock,
BlockSize);

  // Block multiplication
```

```
      BlockMultiplication( pAblock, pBblock, pCblock,
BlockSize );

      // Cyclic shift of blocks of matrix B in process
grid columns
     BblockCommunication ( pBblock, BlockSize, ColComm );
      }
     }
```

Рассмотрим этапы более подробно в ходе отдельных упражнений задачи.

Задание 7 – Рассылка блоков матрицы A

Итак, в начале каждой итерации iter алгоритма для каждой строки процессной решетки выбирается процесс, который будет рассылать свой блок матрицы A по процессам соответствующей строки решетки. Номер этого процесса *Pivot* в строке определяется в соответствии с выражением:

$$Pivot = (i + iter) \bmod GridSize$$

где i – номер строки процессорной решетки, для которой определяется номер рассылающего процесса (для каждого процесса номер строки, в которой он расположен, можно определить, обратившись к первому значению в массиве *GridCoords*), а операция mod есть операция вычисления остатка от деления. Таким образом, на каждой итерации рассылающим назначается процесс, у которого значение второй координаты *GridCoords* совпадает с *Pivot*. После того, как номер рассылающего процесса определен, необходимо выполнить широковещательную рассылку блока матрицы *A* по строке. Сделаем это при помощи функции *MPI_Bcast* в рамках коммуникатора *RowComm*. Здесь нам потребуется использование дополнительного блока матрицы *A*: первый блок *pMatrixAblock* хранит тот блок матрицы, который был помещен на данный процесс перед началом вычислений, а блок *pAblock* хранит тот блок матрицы, который принимает участие в умножении на данной итерации алгоритма. Перед выполнением широковещательной рассылки содержимое блока *pMatrixAblock* копируется в буфер *pAblock*, а затем буфер *pAblock* рассылается на все процессы строки.

```
     // Broadcasting matrix A blocks to process grid rows
     void ABlockCommunication (int iter, double *pAblock,
double* pMatrixAblock,
      int BlockSize) {
```

```
    // Defining the leading process of the process grid
    int Pivot = (GridCoords[0] + iter) % GridSize;

    // Copying the transmitted block in a separate
y buffer
    if (GridCoords[1] == Pivot) {
    for (int i=0; i<BlockSize*BlockSize; i++)
    pAblock[i] = pMatrixAblock[i];
    }

    // Block broadcasting
    MPI_Bcast(pAblock, BlockSize*BlockSize, MPI_DOUBLE,
, RowComm);
  }
```

Оценим правильность выполнения этапа рассылки блоков матрицы *A*. Для этого добавим вызов функции *ParallelResultCalculation* в главную функцию параллельного приложения. Внутри функции *ParallelResultCalculation* закомментируем вызовы не реализованных пока функций перемножения матричных блоков (*SerialResultCalculation*) и циклического сдвига блоков матрицы *B* (*BblockCommunication*).

Напомним, что сейчас для генерации значений исходных матриц используется функция *RandomDataInitialization* (мы использовали такое задание для проверки правильности выполнения этапа распределения данных). Внутри функции *ParallelResultCalculation* после выполнения рассылки блоков матрицы *A* распечатаем значения, которые хранятся в блоках *pAblock* на всех процессорах:

```
  // Execution of the Fox method
  void ParallelResultCalculation(double* pAblock, dou-
ble* pMatrixAblock,
    double* pBblock, double* pCblock, int BlockSize) {
    for (int iter = 0; iter < GridSize; iter ++) {
    // Sending blocks of matrix A to the process grid
rows
    ABlockCommunication(iter, pAblock, pMatrixAblock,
BlockSize);
    if (ProcRank == 0)
    printf("Iteration number %d \n", iter);
    TestBlocks(pAblock, BlockSize, "Block of A matrix");

    // Block multiplication
    // BlockMultiplication ( pAblock, pBblock, pCblock,
BlockSize );
```

```
        // Cyclic shift of blocks of matrix B in process
grid columns
        // BblockCommunication ( pBblock, BlockSize, ColComm
);
        }
      }
```

Скомпилируйте и запустите приложение на 9 процессорах. Проверьте правильность выполнения рассылки блоков матрицы *A*. Для этого сравните блоки, расположенные на процессах на каждой итерации алгоритма Фокса с выводом, выполненным после выполнения функции *DataDistribution*. Номер блока, который расположен на всех процессорах строки *i*, должен вычисляться по формуле (2.4).

Задание 8 – Циклический сдвиг блоков матрицы B вдоль столбцов процессорной решетки

После выполнения умножения матричных блоков нужно осуществить циклический сдвиг блоков матрицы *B* вдоль столбцов процессорной решетки (рис. 2.14).

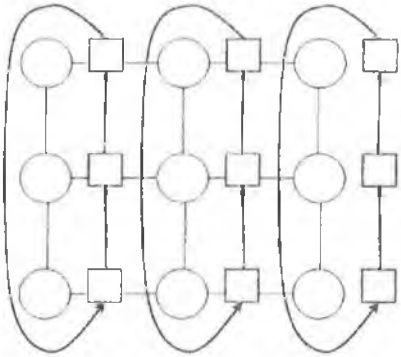

Рис. 2.14. Циклический сдвиг блоков матрицы B вдоль столбцов процессорной решетки

Такой сдвиг можно выполнить несколькими разными способами. Наиболее очевидный подход состоит в организации последовательностей передачи и приема матричных блоков при помощи функций *MPI_Send* и *MPI_Receive* (подробнее об этих функциях можно узнать из раздела 4 лекционного материала).

Сложность здесь состоит в том, чтобы организовать эту последовательность таким образом, чтобы не возникло тупиковых ситуаций, то есть таких ситуаций, когда один процесс ждет приема сообщения от другого процесса, тот, в свою очередь, от третьего, и так далее.

Достижение эффективного и гарантированного одновременного выполнения операций передачи и приема данных может быть обеспечено при помощи функции MPI:

```
int MPI_Sendrecv(void *sbuf,int scount,MPI_Datatype
stype,int dest,
 int stag, void *rbuf,int rcount,MPI_Datatype
rtype,int source,int rtag,
 MPI_Comm comm, MPI_Status *status),
```

где

- **sbuf, scount, stype, dest, stag** - параметры передаваемого сообщения,
- **rbuf, rcount, rtype, source, rtag** - параметры принимаемого сообщения,
- **comm** - коммуникатор, в рамках которого выполняется передача данных,
- **status** – структура данных с информацией о результате выполнения операции.

Как следует из описания, функция *MPI_Sendrecv* передает сообщение, описываемое параметрами (*sbuf, scount, stype, dest, stag*), процессу с рангом *dest* и принимает сообщение в буфер, определяемый параметрами (*rbuf, rcount, rtype, source, rtag*), от процесса с рангом *source*.

В функции *MPI_Sendrecv* для передачи и приема сообщений применяются разные буфера. В случае же, когда сообщения имеют одинаковый тип, в MPI имеется возможность использования единого буфера:

```
int MPI_Sendrecv_replace (void *buf, int count,
MPI_Datatype type,
 int dest,int stag,int source,int rtag,MPI_Comm
comm,MPI_Status* status);
```

Используем эту функцию для организации циклического сдвига блоков *pBblock* матрицы *B*. Каждый процесс посылает сообщение предыдущему процессу того же столбца процессорной решетки и принимает сообщение от следующего процесса. Процесс, расположенный в нулевой строке процессорной решетки посылает свой блок процессу, расположенному в последней

строке (строке с номером *GridSize-1*).

```
   // Cyclic shift of matrix B blocks in the process grid
columns
   void   BblockCommunication   (double   *pBblock,   int
BlockSize,
    MPI_Comm ColumnComm){
    MPI_Status Status;
    int NextProc = GridCoords[0] + 1;
    if ( GridCoords[0] == GridSize-1 ) NextProc = 0;
    int PrevProc = GridCoords[0] - 1;
    if ( GridCoords[0] == 0 ) PrevProc = GridSize-1;
    MPI_Sendrecv_replace(   pBblock,   BlockSize*BlockSize,
MPI_DOUBLE,
    NextProc, 0, PrevProc, 0, ColumnComm, &Status);
   }
```

Оценим правильность выполнения этого этапа. Внутри функции *ParalletResultCalculation* раскомментируем вызов функции циклического сдвига блоков матрицы *B* (*BblockCommunication*). Удалим отладочную печать блоков матрицы *A*. После выполнения рассылки блоков матрицы *B* распечатаем значения, которые хранятся в блоках *pBblock* на всех процессорах:

```
   // Execution of the Fox method
   void ParallelResultCalculation(double* pAblock, double*
pMatrixAblock,
    double* pBblock, double* pCblock, int BlockSize) {
    for (int iter = 0; iter < GridSize; iter ++) {
    // Sending blocks of matrix A to the process grid rows
    ABlockCommunication(iter,    pAblock,    pMatrixAblock,
BlockSize);
    // Block multiplication
    //  BlockMultiplication ( pAblock, pBblock, pCblock,
BlockSize );
    // Cyclic shift of blocks of matrix B in process grid
columns
    BblockCommunication ( pBblock, BlockSize, ColComm );
    if (ProcRank == 0)
    printf("Iteration number %d \n", iter);
    TestBlocks(pAblock, BlockSize, "Block of B matrix");
    }
   }
```

Скомпилируйте и запустите приложение на 9 процессорах. Проверьте правильность выполнения рассылки блоков матрицы *B* (на каждой следующей итерации блоки матрицы *B* должны смещаться на 1 вверх вдоль столбца процессорной решетки). Для этого сравните блоки, расположенные на процессах на каждой итерации алгоритма Фокса с выводом, выполненным после вы-

...ления функции *DataDistribution*.

...ание 10 – Сбор результатов

Процедура сбора результатов повторяет процедуру распределения исходных данных, разница состоит в том, что этапы необходимо выполнять в обратном порядке. Сначала необходимо осуществить сбор полосы результирующей матрицы из блоков, расположенных на процессорах одной строки процессорной решетки. Далее нужно собрать матрицу из полос, расположенных на процессорах, составляющих столбец процессорной решетки.

Для сбора результирующей матрицы будем использовать функцию MPI_Gather библиотеки MPI. Эта функция собирает данные со всех процессов в коммуникаторе на одном процессе. Действия, выполняемые этой функцией, противоположны действиям функции MPI_Scatter. Функция MPI_Gather имеет следующий интерфейс:

```
int MPI_Gather(void *sbuf,int scount,MPI_Datatype stype,
  void *rbuf,int rcount,MPI_Datatype rtype, int root,
MPI_Comm comm),
```

где

- sbuf, scount, stype - параметры передаваемого сообщения,
- rbuf, rcount, rtype - параметры принимаемого сообщения,
- root – ранг процесса, выполняющего сбор данных,
- comm - коммуникатор, в рамках которого выполняется передача данных.

Процедуру сбора результирующей матрицы C реализуем непосредственно в функции ResultCollection:

```
// Function for gathering the result matrix
void ResultCollection (double* pCMatrix, double*
pCblock, int Size,
  int BlockSize) { double * pResultRow = new double
[Size*BlockSize];
  for (int i=0; i<BlockSize; i++) {
  MPI_Gather(      &pCblock[i*BlockSize],      BlockSize,
MPI_DOUBLE,
  &pResultRow[i*Size], BlockSize, MPI_DOUBLE, 0, Row-
Comm);
  }

  if (GridCoords[1] == 0) {
  MPI_Gather(pResultRow,    BlockSize*Size,    MPI_DOUBLE,
pCMatrix,
  BlockSize*Size, MPI_DOUBLE, 0, ColComm);
```

```
    }
    delete [] pResultRow;
  }
```

Добавим вызов функции ResultCollection вместо вызова функции тестирования частичных результатов при помощи отладочной печати (TestBlocks). Для контроля правильности сбора данных и работы алгоритма в целом, распечатаем результирующую матрицу pCMatrix на ведущем процессе с использованием функции PrintMatrix.

```
  void main(int argc, char* argv[]) {
    <…>
    // Execution of Fox method
    ParallelResultCalculation(pAblock,        pMatrixAblock,
pBblock, pCblock,
    BlockSize);
    // TestBlocks(pCblock, BlockSize, "Result blocks");

    ResultCollection(pCMatrix, pCblock, Size, BlockSize);
    if (ProcRank == 0) {
    printf("Result matrix \n");
    PrintMatrix(pCMatrix, Size, Size);
    }

    // Process Termination
    ProcessTermination    (pAMatrix,    pBMatrix,    pCMatrix,
pAblock, pBblock,
    pCblock, pMatrixAblock);
    MPI_Finalize();
  }
```

Скомпилируйте и запустите приложение. Оцените правильность работы приложения. Напомним, что если исходные данные генерируются при помощи функции DummyDataInitialization, то все элементы результирующей матрицы должны быть равны ее порядку Size (рис. 2.16).

Рис. 2.16. Результат работы алгоритма Фокса

Задание 11 – Проверка правильности работы программы

Теперь, после выполнения функции сбора, необходимо проверить правильность выполнения алгоритма. Для этого разработаем функцию TestResult, которая сравнит результаты последовательного и параллельного алгоритмов. Для выполнения последовательного алгоритма можно использовать функцию SerialResultCalculation, разработанную в упражнении 2. Результат работы этой функции сохраним в матрице pSerialResult, а затем поэлементно сравним эту матрицу с матрицей pCMatrix, полученной при помощи параллельного алгоритма Фокса. Получение каждого элемента результирующей матрицы 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 матриц на равенство. Введем допустимую величину расхождения результатов последовательного и параллельного алгоритма Accuracy. Матрицы будем считать равными в том случае, когда соответствующие элементы отличаются не более чем на величину допустимой погрешности Accuracy.

Функция TestResult должна иметь доступ к исходным матрицам pAMatrix и pCMatrix, а значит может быть выполнена только на ведущем процессе:

```
   void TestResult(double* pAMatrix, double* pBMatrix, dou-
ble* pCMatrix,
    int Size) {
    double* pSerialResult; // Result matrix of serial mul-
tiplication
    double Accuracy = 1.e-6; // Comparison accuracy
    int equal = 0; // =1, if the matrices are not equal
    int i; // Loop variable

    if (ProcRank == 0) {
    pSerialResult = new double [Size*Size];
    for (i=0; i<Size*Size; i++) {
    pSerialResult[i] = 0;
    }
    SerialResultCalculation(pAMatrix, pBMatrix, pSerialRe-
sult, Size);
    for (i=0; i<Size*Size; i++) {
    if (fabs(pSerialResult[i]-pCMatrix[i]) >= Accuracy)
```

```
    equal = 1;
    }
    if (equal == 1)
    printf("The results of serial and parallel algorithms "
    "are NOT identical. Check your code.");
    else
    printf("The results of serial and parallel algorithms "
    "are identical.");
    delete [] pSerialResult;
    }
}
```

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 матрицы простого вида, вызовите функцию RandomDataInitialization, которая генерирует исходные матрицы при помощи датчика случайных чисел. Скомпилируйте и запустите приложение. Задавайте различные объемы исходных данных. Убедитесь в том, что приложение работает правильно.

Задание 12 – 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();
    DataDistribution(pAMatrix,   pBMatrix,   pMatrixAblock,
pBblock,
    Size, BlockSize);

    // Execution of the Fox method
    ParallelResultCalculation(pAblock,       pMatrixAblock,
pBblock, pCblock,
    BlockSize);
```

```
ResultCollection(pCMatrix, pCblock, Size, BlockSize);
Finish = MPI_Wtime();
Duration = Finish-Start;

TestResult(pAMatrix, pBMatrix, pCMatrix, Size);
if (ProcRank == 0) {
printf("Time of execution = %f\n", Duration);
}

ProcessTermination (pAMatrix, pBMatrix, pCMatrix,
pAblock, pBblock,
pCblock, pMatrixAblock);

MPI_Finalize();
```

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 основания 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 2.3. Время выполнения параллельного алгоритма Фокса матричного умножения и достигнутое ускорение

Номер теста	Размер матриц	Последовательный алгоритм	Параллельный алгоритм			
			4 процесса		9 процессов	
			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 2. 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 время выполнения параллельного алгоритма, реализованного согласно вычислительной схеме, приведенной в упражнении 3, можно воспользоваться следующим соотношением:

$$T_p = q[(n^2/p)\cdot(2n/q-1)+(n^2/p)]\cdot\tau+(q\log_2 q+(q-1))(\alpha+w(n^2/p)/\beta) \quad (2.5)$$

(подробный вывод этой формулы приведен в "Параллельные алгоритмы матричного умножения"). Здесь n – размер объектов, p – количество процессов, q – размер процессорной решетки, τ – время выполнения одной скалярной операции (значение было нами вычислено при тестировании последовательного алгоритма), α – латентность а β – пропускная способность сети передачи данных.

Вычислите теоретическое время выполнения параллельного алгоритма по формуле (2.5). Результаты занесите в таблицу 2.4:

Таблица 2.4. Сравнение реального времени выполнения параллельного алгоритма со временем, вычисленным теоретически

Номер текста	Размер матриц	4 процесса		9 процессов	
		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 при матрице размером 10 на 10? Почему?
- Насколько хорошо совпадают время, полученное теоретически, и реальное время выполнения алгоритма? В чем может состоять причина несовпадений?

Задания для самостоятельной работы

1. Измените разработанную реализацию алгоритма Фокса, использовав для рассылки и сборки блоков матриц производный тип данных MPI (см. раздел 4 "Параллельное программирование на основе MPI").
2. Изучите параллельный алгоритм умножения матриц, основанный на ленточном разделении матрицы. Разработайте программу, реализующую этот алгоритм.
3. Изучите параллельный алгоритм Кэннона умножения матриц, основанный на блочном разделении матрицы. Разработайте программу, реализующую этот алгоритм.

Программный код последовательного приложения для матричного умножения

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <time.h>
// Function for simple initialization of matrix elements
void DummyDataInitialization (double* pAMatrix,double*
pBMatrix,int Size) {
  int i, j; // Loop variables
  for (i=0; i<Size; i++)
  for (j=0; j<Size; j++) {
  pAMatrix[i*Size+j] = 1;
  pBMatrix[i*Size+j] = 1;
  }
}
// Function for random initialization of matrix elements
void RandomDataInitialization (double* pAMatrix, double*
pBMatrix,
  int Size) {
```

```
    int i, j; // Loop variables
    srand(unsigned(clock()));
    for (i=0; i<Size; i++)
    for (j=0; j<Size; j++) {
    pAMatrix[i*Size+j] = rand()/double(1000);
    pBMatrix[i*Size+j] = rand()/double(1000);
    }
  }
  // Function for memory allocation and initialization of
matrix elements
  void ProcessInitialization (double* &pAMatrix, double*
&pBMatrix,
    double* &pCMatrix, int &Size) {
    // Setting the size of matricies
    do {
    printf("\nEnter size of matricies: ");
    scanf("%d", &Size);
    printf("\nChosen matricies' size = %d\n", Size);
    if (Size <= 0)
    printf("\nSize of objects must be greater than 0!\n");
    }
    while (Size <= 0);

    // Memory allocation
    pAMatrix = new double [Size*Size];
    pBMatrix = new double [Size*Size];
    pCMatrix = new double [Size*Size];

    // Initialization of matrix elements
    DummyDataInitialization(pAMatrix, pBMatrix, Size);
    for (int i=0; i<Size*Size; i++) {
    pCMatrix[i] = 0;
    }
  }

  // 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 matrix multiplication
  void SerialResultCalculation(double* pAMatrix, double*
pBMatrix,
    double* pCMatrix, int Size) {
```

```
  int i, j, k; // Loop variables
  for (i=0; i<Size; i++) {
  for (j=0; j<Size; j++)
  for (k=0; k<Size; k++)
  pCMatrix[i*Size+j]                    +=                    pAMa-
trix[i*Size+k]*pBMatrix[k*Size+j];
  }
}


// Function for computational process termination
void ProcessTermination (double* pAMatrix, double* pBMa-
trix,
  double* pCMatrix) {
  delete [] pAMatrix;
  delete [] pBMatrix;
  delete [] pCMatrix;
}


void main() {
  double* pAMatrix; // The first argument of matrix mul-
tiplication
  double* pBMatrix; // The second argument of matrix mul-
tiplication
  double* pCMatrix; // The result matrix
  int Size; // Size of matricies
  time_t start, finish;
  double duration;

  printf("Serial matrix multiplication program\n");
  // Memory allocation and initialization of matrix ele-
ments
  ProcessInitialization(pAMatrix,   pBMatrix,   pCMatrix,
Size);

  // Matrix output
  printf ("Initial A Matrix \n");
  PrintMatrix(pAMatrix, Size, Size);
  printf("Initial B Matrix \n");
  PrintMatrix(pBMatrix, Size, Size);

  // Matrix multiplication
  start = clock();
  SerialResultCalculation(pAMatrix,   pBMatrix,   pCMatrix,
Size);
  finish = clock();
  duration = (finish-start)/double(CLOCKS_PER_SEC);

  // Printing the result matrix
  printf ("\n Result Matrix: \n");
  PrintMatrix(pCMatrix, Size, Size);
```

```
    // Printing the time spent by matrix multiplication
    printf("\n Time of execution: %f\n", duration);

    // Computational process termination
    ProcessTermination(pAMatrix, pBMatrix, pCMatrix);
  }
```

Программный код параллельного приложения для матричного умножения

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

  int ProcNum = 0; // Number of available processes
  int ProcRank = 0; // Rank of current process
  int GridSize; // Size of virtual processor grid
  int GridCoords[2]; // Coordinates of current processor
in grid
  MPI_Comm GridComm; // Grid communicator
  MPI_Comm ColComm; // Column communicator
  MPI_Comm RowComm; // Row communicator

  /// Function for simple initialization of matrix ele-
ments
  void DummyDataInitialization (double* pAMatrix, double*
pBMatrix,int
    Size){
    int i, j; // Loop variables
    for (i=0; i<Size; i++)
    for (j=0; j<Size; j++) {
    pAMatrix[i*Size+j] = 1;
    pBMatrix[i*Size+j] = 1;
    }
  }

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

```
    }
  }

  // Function for formatted matrix output
void PrintMatrix (double* pMatrix, int RowCount, int Col-
Count) {

  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 matrix multiplication
  void SerialResultCalculation(double* pAMatrix, double*
pBMatrix,
    double* pCMatrix, int Size) {
    int i, j, k; // Loop variables
    for (i=0; i<Size; i++) {
    for (j=0; j<Size; j++)
    for (k=0; k<Size; k++)
    pCMatrix[i*Size+j]                +=                pAMa-
trix[i*Size+k]*pBMatrix[k*Size+j];
    }
  }

  // Function for block multiplication
  void   BlockMultiplication(double*   pAblock,   double*
pBblock,
    double* pCblock, int Size) {
    SerialResultCalculation(pAblock,   pBblock,   pCblock,
Size);
  }

  // Creation of two-dimensional grid communicator
  // and communicators for each row and each column of the
grid
  void CreateGridCommunicators() {
    int DimSize[2]; // Number of processes in each dimen-
sion of the grid
    int Periodic[2]; // =1, if the grid dimension should be
periodic
    int Subdims[2]; // =1, if the grid dimension should be
fixed
    DimSize[0] = GridSize;
    DimSize[1] = GridSize;
    Periodic[0] = 0;
    Periodic[1] = 0;
```

```
    // Creation of the Cartesian communicator
    MPI_Cart_create(MPI_COMM_WORLD, 2, DimSize, Periodic,
1, &GridComm);

    // Determination of the cartesian coordinates for every
process
    MPI_Cart_coords(GridComm, ProcRank, 2, GridCoords);

    // Creating communicators for rows
    Subdims[0] = 0; // Dimensionality fixing
    Subdims[1] = 1; // The presence of the given dimension
in the subgrid
    MPI_Cart_sub(GridComm, Subdims, &RowComm);

    // Creating communicators for columns
    Subdims[0] = 1;
    Subdims[1] = 0;
    MPI_Cart_sub(GridComm, Subdims, &ColComm);
  }

  // Function for memory allocation and data initializa-
tion
  void ProcessInitialization (double* &pAMatrix, double*
&pBMatrix,
    double* &pCMatrix, double* &pAblock, double* &pBblock,
double*
    &pCblock,
    double* &pTemporaryAblock, int &Size, int &BlockSize )
{
    if (ProcRank == 0) {
    do {
    printf("\nEnter size of the initial objects: ");
    scanf("%d",&Size);
    if (Size%GridSize != 0) {
    printf ("Size of matricies must be divisible by the
grid
    size!\n");
    }
    }
    while (Size%GridSize != 0);
    }
    MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);
    BlockSize = Size/GridSize;
    pAblock = new double [BlockSize*BlockSize];
    pBblock = new double [BlockSize*BlockSize];
    pCblock = new double [BlockSize*BlockSize];
    pTemporaryAblock = new double [BlockSize*BlockSize];
    for (int i=0; i<BlockSize*BlockSize; i++) {
    pCblock[i] = 0;
    }
```

```
    if (ProcRank == 0) {
    pAMatrix = new double [Size*Size];
    pBMatrix = new double [Size*Size];
    pCMatrix = new double [Size*Size];
    DummyDataInitialization(pAMatrix, pBMatrix, Size);
    //RandomDataInitialization(pAMatrix, pBMatrix, Size);
    }
  }

  // Function for checkerboard matrix decomposition
  void CheckerboardMatrixScatter(double* pMatrix, double*
pMatrixBlock,
    int Size, int BlockSize) {
    double * MatrixRow = new double [BlockSize*Size];
    if (GridCoords[1] == 0) {
    MPI_Scatter(pMatrix, BlockSize*Size, MPI_DOUBLE, Ma-
trixRow,
    BlockSize*Size, MPI_DOUBLE, 0, ColComm);
    }
    for (int i=0; i<BlockSize; i++) {
    MPI_Scatter(&MatrixRow[i*Size], BlockSize, MPI_DOUBLE,
    &(pMatrixBlock[i*BlockSize]), BlockSize, MPI_DOUBLE, 0,
RowComm);
    }
    delete [] MatrixRow;
  }

  // Data distribution among the processes
  void DataDistribution(double* pAMatrix, double* pBMa-
trix, double*
    pMatrixAblock, double* pBblock, int Size, int
BlockSize) {
    // Scatter the matrix among the processes of the first
grid column
    CheckerboardMatrixScatter(pAMatrix, pMatrixAblock,
Size, BlockSize);
    CheckerboardMatrixScatter(pBMatrix, pBblock, Size,
BlockSize);
  }

  // Function for gathering the result matrix
  void ResultCollection (double* pCMatrix, double*
pCblock, int Size,
    int BlockSize) {
    double * pResultRow = new double [Size*BlockSize];
    for (int i=0; i<BlockSize; i++) {
    MPI_Gather( &pCblock[i*BlockSize], BlockSize,
MPI_DOUBLE,
    &pResultRow[i*Size], BlockSize, MPI_DOUBLE, 0, Row-
Comm);
```

```
    }
    if (GridCoords[1] == 0) {
    MPI_Gather(pResultRow,    BlockSize*Size,    MPI_DOUBLE,
pCMatrix,
    BlockSize*Size, MPI_DOUBLE, 0, ColComm);
    }
    delete [] pResultRow;
   }

   // Broadcasting matrix A blocks to process grid rows
   void ABlockCommunication (int iter, double *pAblock,
double*
    pMatrixAblock, int BlockSize) {

    // Defining the leading process of the process grid row
    int Pivot = (GridCoords[0] + iter) % GridSize;

    // Copying the transmitted block in a separate memory
buffer
    if (GridCoords[1] == Pivot) {
    for (int i=0; i<BlockSize*BlockSize; i++)
    pAblock[i] = pMatrixAblock[i];
    }

    // Block broadcasting
    MPI_Bcast(pAblock,   BlockSize*BlockSize,   MPI_DOUBLE,
Pivot, RowComm);
   }

   // Cyclic shift of matrix B blocks in the process grid
columns
   void   BblockCommunication   (double   *pBblock,   int
BlockSize) {
    MPI_Status Status;
    int NextProc = GridCoords[0] + 1;
    if ( GridCoords[0] == GridSize-1 ) NextProc = 0;
    int PrevProc = GridCoords[0] - 1;
    if ( GridCoords[0] == 0 ) PrevProc = GridSize-1;

    MPI_Sendrecv_replace(   pBblock,   BlockSize*BlockSize,
MPI_DOUBLE,
    NextProc, 0, PrevProc, 0, ColComm, &Status);
   }

   void ParallelResultCalculation(double* pAblock, double*
pMatrixAblock,
    double* pBblock, double* pCblock, int BlockSize) {
    for (int iter = 0; iter < GridSize; iter ++) {
    // Sending blocks of matrix A to the process grid rows
```

```
    ABlockCommunication (iter, pAblock, pMatrixAblock,
BlockSize);
    // Block multiplication
    BlockMultiplication(pAblock, pBblock, pCblock,
BlockSize);
    // Cyclic shift of blocks of matrix B in process grid
columns
    BblockCommunication(pBblock, BlockSize);
    }
  }

  // Test printing of the matrix block
  void TestBlocks (double* pBlock, int BlockSize, char
str[]) {
    MPI_Barrier(MPI_COMM_WORLD);
    if (ProcRank == 0) {
    printf("%s \n", str);
    }
    for (int i=0; i<ProcNum; i++) {
    if (ProcRank == i) {
    printf ("ProcRank = %d \n", ProcRank);
    PrintMatrix(pBlock, BlockSize, BlockSize);
    }
    MPI_Barrier(MPI_COMM_WORLD);
    }
  }

  void TestResult(double* pAMatrix, double* pBMatrix, dou-
ble* pCMatrix,
    int Size) {
    double* pSerialResult; // Result matrix of serial mul-
tiplication
    double Accuracy = 1.e-6; // Comparison accuracy
    int equal = 0; // =1, if the matrices are not equal
    int i; // Loop variable
    if (ProcRank == 0) {
    pSerialResult = new double [Size*Size];
    for (i=0; i<Size*Size; i++) {
    pSerialResult[i] = 0;
    }
    BlockMultiplication(pAMatrix, pBMatrix, pSerialResult,
Size);
    for (i=0; i<Size*Size; i++) {
    if (fabs(pSerialResult[i]-pCMatrix[i]) >= Accuracy)
    equal = 1;
    }
    if (equal == 1)
    printf("The results of serial and parallel algorithms
are NOT "
    "identical. Check your code.");
```

```
    else
    printf("The results of serial and parallel algorithms
are "
    "identical.");
    }
  }

  // Function for computational process termination
  void ProcessTermination (double* pAMatrix, double* pBMa-
trix,
    double* pCMatrix, double* pAblock, double* pBblock,
double* pCblock,
    double* pMatrixAblock) {
    if (ProcRank == 0) {
    delete [] pAMatrix;
    delete [] pBMatrix;
    delete [] pCMatrix;
    }
    delete [] pAblock;
    delete [] pBblock;
    delete [] pCblock;
    delete [] pMatrixAblock;
  }

  void main(int argc, char* argv[]) {
    double* pAMatrix; // The first argument of matrix mul-
tiplication
    double* pBMatrix; // The second argument of matrix mul-
tiplication
    double* pCMatrix; // The result matrix
    int Size; // Size of matricies
    int BlockSize; // Sizes of matrix blocks on current
process
    double *pAblock; // Initial block of matrix A on cur-
rent process
    double *pBblock; // Initial block of matrix B on cur-
rent process
    double *pCblock; // Block of result matrix C on current
process
    double *pMatrixAblock;
    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);

    GridSize = sqrt((double)ProcNum);
    if (ProcNum != GridSize*GridSize) {
```

```
    if (ProcRank == 0) {
    printf ("Number of processes must be a perfect square
\n");
    }
    }
    else {
    if (ProcRank == 0)
    printf("Parallel matrix multiplication program\n");

    // Creating the cartesian grid, row and column commun-
icators
    CreateGridCommunicators();
    // Memory allocation and initialization of matrix ele-
ments
    ProcessInitialization ( pAMatrix, pBMatrix, pCMatrix,
pAblock,
    pBblock, pCblock, pMatrixAblock, Size, BlockSize );
    DataDistribution(pAMatrix,   pBMatrix,   pMatrixAblock,
pBblock, Size,
    BlockSize);
    // Execution of Fox method
    ParallelResultCalculation(pAblock,       pMatrixAblock,
pBblock,
    pCblock, BlockSize);
    ResultCollection(pCMatrix, pCblock, Size, BlockSize);
    TestResult(pAMatrix, pBMatrix, pCMatrix, Size);
    // Process Termination
    ProcessTermination   (pAMatrix,   pBMatrix,   pCMatrix,
pAblock, pBblock,
    pCblock, pMatrixAblock);
    }

    MPI_Finalize();
```

ЗАДАЧА 3: ПАРАЛЛЕЛЬНЫЕ МЕТОДЫ РЕШЕНИЯ СИСТЕМ ЛИНЕЙНЫХ УРАВНЕНИЙ

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 задаче рассматривается один из прямых методов решения систем линейных уравнений - метод Гаусса и его параллельное обобщение.

Обзор задачи

Целью данной задачи является разработка параллельной программы, которая выполняет решение системы линейных уравнений методом Гаусса. Выполнение задачи 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

При выполнении задачи предполагается знание раздела "Параллельное программирование на основе MPI", раздела "Принципы разработки параллельных методов" и раздела "Параллельные методы решения систем линейных уравнений".

Упражнение 1 - Определение задачи решения системы линейных уравнений

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

$$a_0x_0 + a_1x_1 + ... + a_{n-1}x_{n-1} = b \qquad (3.1)$$

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

Множество n линейных уравнений

$$a_{0,0}x_0 + a_{0,1}x_1 + \ldots + a_{0,n-1}x_{n-1} = b_0$$
$$a_{1,0}x_0 + a_{1,1}x_1 + \ldots + a_{1,n-1}x_{n-1} = b_1 \qquad (3.2)$$
$$\ldots$$
$$a_{n-1,0}x_0 + a_{n-1,1}x_1 + \ldots + a_{n-1,n-1}x_{n-1} = b_{n-1}$$

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 $n * n$, 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:

$$U = \begin{pmatrix} u_{0,0} & \cdots & u_{0,n-1} \\ \vdots & \ddots & \vdots \\ 0 & \cdots & u_{n-1,n-1} \end{pmatrix}$$

На обратном ходе метода Гаусса (второй этап алгоритма) осуществляется определение значений неизвестных. Из последнего уравнения преобразованной системы может быть вычислено значение переменной xn-1, после этого из предпоследнего уравнения становится возможным определение переменной xn-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≤ i<n-1, метода производится исключение неизвестной i для всех уравнений с номерами k, больших i (т.е. i< k< 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:

$$a_{kj} = a_{kj} - (a_{ki}/a_{ii})a_{ij}$$
$$b_k = b_k - (a_{ki}/a_{ii})b \qquad (3.3)$$
$$i \le j \le n\text{-}1,\ i < k \le n\text{-}1, 0 \le i < n\text{-}1$$

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

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

$$2x_0 + 7x_1 + 5x_2 = 18.$$
$$x_0 + 3x_1 + 2x_2 = 1$$
$$x_0 + 4x_1 + 6x_2 = 26$$

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:

$$x_0 + 3x_1 + 2x_2 = 1$$
$$x_1 + x_2 = 16.$$
$$x_1 + 4x_2 = 25$$

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:

$$x_0 + 3x_1 + 2x_2 = 1$$
$$x_1 + x_2 = 16\ .$$
$$3x_2 = 9$$

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 ниже 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 $(n\text{-}1)$ подобной итерации матрица,

определяющая систему линейных уравнений, становится приведенной к верхнему треугольному виду.

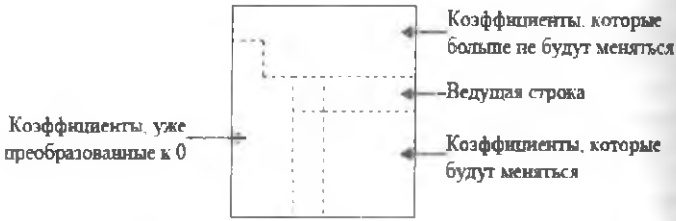

Рис. 3.1. Итерация прямого хода алгоритма Гаусса

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 имеет малое значение, то деление и умножение строк на этот элемент может приводить к накоплению вычислительной погрешности и вычислительной неустойчивости алгоритма.

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

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, 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(n3).

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:

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

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

$$x_0 + 3x_1 + 2x_2 = 1$$
$$x_1 + 2x_2 = 16$$
$$3x_2 = 9$$

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.

$$x_0 + 3x_1 + 2x_2 = 1$$
$$x_1 = 13$$
$$x_2 = 3$$

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 выполниться 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

$$x_0 + 3x_1 = -5$$
$$x_1 = 13$$
$$x_2 = 3$$

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.

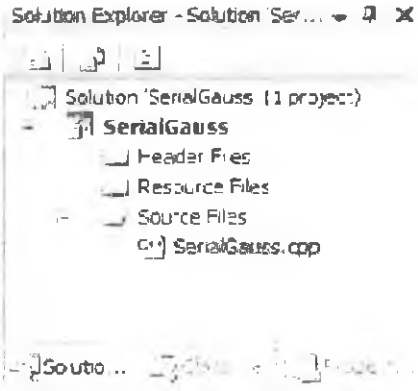

Рис. 3.2. Открытие файла SerialGauss.cpp

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 задачах, 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 sys
tems\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 sys-
tem
  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 sys-
tems\n");
  ProcessInitialization(pMatrix, pVector, pResult, Size);
  getch();
}
```

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

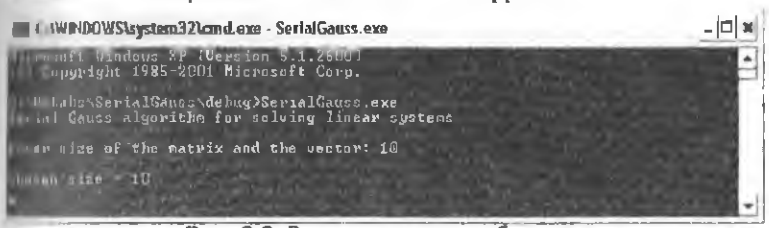

Рис. 3.3. Задание размера объектов

Как и при выполнении предыдущих задач, 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 vec-
tor: ");
     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 initializa-
tion
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 нецелесооб-

ровка (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 initializa-
tion
    void ProcessInitialization (double* &pAMatrix, double*
&pBMatrix,
     double* &pCMatrix, int &Size) {
     <...>
     // Memory allocation
     <...>
     // Initialization of the matrix and the vector ele-
ments
     DummyDataInitialization(pMatrix, pVector, Size);
    }
```

Для контроля ввода данных воспользуемся функциями форматированного вывода объектов *PrintMatrix* и *PrintVector*, которые были разработаны при выполнении задачи 1 и текст которых уже имеется в проекте (подробнее о функциях *PrintMatrix* и *PrintVector* см. задание 3 упражнение 2 задачи 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 урав-

ений 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* pVec-
tor,double* pResult){
     delete [] pMatrix;
     delete [] pVector;
     delete [] pResult;
    }
```

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

```
    // Memory allocation and definition of the objects ele-
ments
    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 *SerialResultCalcula-*

main 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 задачи.

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 select-
ed 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 ele-
ment
    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* закомментируйте 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();
  }
```

В функции *SerialResultCalculation* закомментируйте 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] = pVec-
tor[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).

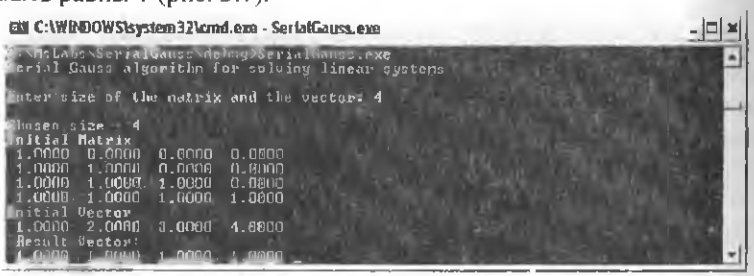

Рис. 3.7. Результат выполнения последовательного алгоритма Гаусса

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. Заполнять 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:

```
clock_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, pRe-
sult, 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"):

$$T_1 = (2size^3 / 3 + size^2)\tau \qquad (3.5),$$

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 m, как и при выполнении предыдущих задач, 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 экслериментов. 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 задачах, 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 задачи). 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:

```
  }
}
```

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 зна-

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 положительные числа от 1 до *Size*. (рис. 3.9).

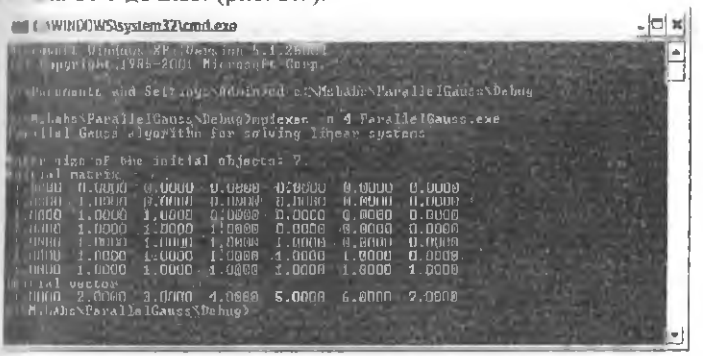

Рис. 3.9. Задание исходных данных

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* pProcRe-
sult) {
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*. В нее 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 задаче 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
  int *pSendInd; // The index of the first data element sent
  // to the process
  int RestRows=Size; // The number of rows, that have not
  // 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 cur-
  rent 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 задаче. 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, pProcRe-
sult, 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 algo-
rithm
 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, pProcRe-
sult, 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 задачи.

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, dou-
ble* 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, dou-
ble* 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, dou-
ble* pProcVector,
 int Size, int RowNum) {
 double MaxValue; // The value of the pivot element of th-
process
 int PivotPos; // The position of the pivot row in the
stripe
 // of the process
 struct { double MaxValue; int ProcRank; } ProcPivot, Piv-
ot;
 // 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, dou-
ble* 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, Piv-
ot;
 // 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,    Piv-
ot.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, dou-
ble* 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, Piv-
ot;
 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,    Piv-
ot.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, dou-
ble* 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).

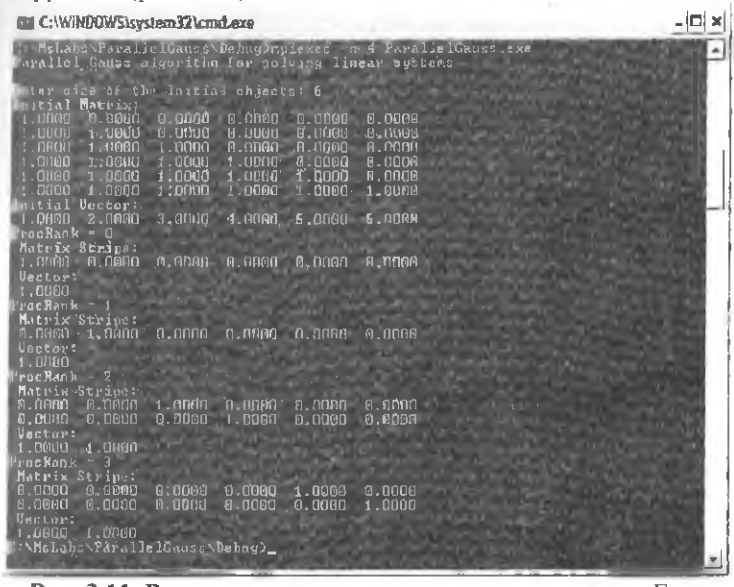

Рис. 3.11. Результат выполнения прямого хода метода Гаусса

Задание 6 – 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 sub
stitution
 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 cur
rent
 // 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
 FindBack-
PivotRow(pParallelPivotPos[i],Size,IterProcRank,IterPivotPo
s);
 // 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 7 – 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 реа-

лизацию:

```
// 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, pProcRe
sult,
 Size, RowNum);
 // Gathering the result vector
 ResultCollection(pProcResult, pResult);
```

Собранный вектор состоит из элементов вектора неизвест ных, 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).

Шаг 8 – 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 mul-
tiplication
 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 9 - 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, pProcVec-
tor, 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 основания 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 выпол-

нения 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:

$$T_p = \frac{1}{p}\sum_{i=1}^{n}(3i+2i^2)\tau + (n-1)\log_2 p(3\alpha + \omega(n+2)\ \beta)$$

Здесь n - размер системы линейных уравнений, p - количество процессов, m - время выполнения одной скалярной операции (значение было нами вычислено при тестировании последовательного алгоритма), α- латентность а β - пропускная способность сети передачи данных.

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

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

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* SpMatrix, double*
SpVector,
 double* SpResult, int SSize) {
 // Setting the size of the matrix and the vector
 do {
 printf("\nEnter size of the matrix and the vector: ");
 scanf("%d", SSize);
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 Col-
Count) {
 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* pVec-
tor,
  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* pVec-
tor,
 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 vec-
tor
 time_t start, finish;
 double duration;
 printf("Serial Gauss algorithm for solving linear sys-
tems\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();
```

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;
 }
 }
```

```
 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 Col
Count) {
 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, dou-
ble* pProcVector,
int Size, int RowNum) {
double MaxValue; // Value of the pivot element of the pro-
cess
int PivotPos; // Position of the pivot row in the process
stripe
// Structure for the pivot row selection
struct { double MaxValue; int ProcRank; } ProcPivot, Piv-
ot;
// 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, Piv-
ot.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 substitu-
tion
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 pro-
cess
 double IterResult; // Calculated value of the current un-
known
 double val;
 // Iterations of the back subst itution 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 algo-
rithm
void ParallelResultCalculation(double* pProcRows, double*
pProcVector,
 double* pProcResult, int Size, int RowNum) {
 ParallelGaussianElimination (pProcRows, pProcVector, Size,
RowNum);
 ParallelBackSubstitution (pProcRows, pProcVector, pProcRe-
sult, Size,
 RowNum);
}
// Function for computational process termination
 void ProcessTermination (double* pMatrix, double* pVector,
double* pResult,
 double* pProcRows, double* pProcVector, double* pProcRe-
sult) {
 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 mul
tiplication
 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 cor-
rect.");
 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 sys-
tems\n");
 // Memory allocation and data initialization
 ProcessInitialization(pMatrix, p Vector, 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, pProcRe-
sult, 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( );
}
```

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Гергель В.П. Основы параллельных вычислений для многопроцессорных вычислительных систем [Текст]/ Гергель В.П., Стронгин, Р.Г. - Н.Новгород, ННГУ, 2003.
2. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления [Текст]/ Воеводин В.В., Воеводин Вл.В. – СПб.: БХВ-Петербург, 2002.
3. Немнюгин С., Стесик О. Параллельное программирование для многопроцессорных вычислительных систем [Текст]/ Немнюгин С., Стесик О. – СПб.: БХВ-Петербург, 2002.
4. Таненбаум Э. Архитектура компьютера [Текст]/ Таненбаум Э. – СПб.: Питер, 2002.
5. Quinn, M. J. Parallel Programming in C with MPI and OpenMP [Текст]/ Quinn, M. J. – New York, NY: McGraw-Hill, 2004.
6. Grama, A. Introduction to Parallel Computing [Текст]/ Grama, A., Gupta, A., Kumar V. – Harlow, England: Addison-Wesley, 2003
7. Pacheco, P. Parallel Programming with MPI. [Текст]/ Pacheco, P. - Morgan Kaufmann, 1996.
8. Chandra, R. Parallel Programming in OpenMP. [Текст]/ Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., and Melon, R. - Morgan Kaufmann Publishers, 2000.
9. Culler, D., Singh, J.P., Gupta, A. Parallel Computer Architecture: A Hardware/Software Approach. [Текст]/ Culler, D., Singh, J.P., Gupta, A. - Morgan Kaufmann, 1998.
10. Таненбаум Э. Современные операционные системы. [Текст]/ Таненбаум Э. – СПб.: Питер, 2002.
11. Кнут, Д. Искусство программирования, том 1. Основные алгоритмы [Текст]/ Кнут Д. — М.:«Вильямс», 2006.

Подписано в печать: 20.08.2013 г.
Формат: 60x84 1/16. Бумага офсетная.
Печать оперативная. Усл. печ. л. 11,68
Тираж: 100 экз. Заказ № 442

Отпечатано в типографии АНО «Издательство СНЦ РАН»
443001, Самара, Студенческий пер., 3а
тел.: (846) 242-37-07