Лабораторная работа 1: Параллельные алгоритмы матрично-векторного умножения

Цель лабораторной работы ... 2
Упражнение 1 – Постановка задачи матрично-векторного умножения ... 2
Упражнение 2 – Реализация последовательного алгоритма умножения матрицы на вектор ... 3
Задание 1 – Открытие проекта SerialMatrixVectorMult ... 3
Задание 2 – Ввод размеров объектов ... 4
Задание 3 – Ввод данных ... 5
Задание 4 – Завершение процесса вычислений ... 7
Задание 5 – Реализация умножения матрицы на вектор ... 7
Задание 6 – Проведение вычислительных экспериментов ... 8
Упражнение 3 – Разработка параллельного алгоритма умножения матрицы на вектор ... 10
Принципы распараллеливания ... 10
Определение подзадач ... 11
Выделение информационных зависимостей ... 12
Масштабирование и распределение подзадач по процессорам ... 12
Упражнение 4 – Реализация параллельного алгоритма матрично-векторного умножения ... 13
Понятие параллельной программы ... 13
Понятие коммуникатора и группы процессов ... 13
Задание 1 – Открытие проекта ParallelMatrixVectorMult ... 13
Задание 2 – Инициализация и завершение параллельной программы ... 14
Задание 3 – Определение количества процессов ... 16
Задание 4 – Ввод размера матрицы и вектора ... 17
Задание 5 – Ввод исходных данных ... 19
Задание 6 – Завершение процесса вычислений ... 20
Задание 7 – Распределение данных между процессами ... 20
Задание 8 – Реализация умножения матрицы на вектор ... 22
Задание 9 – Сбор результатов ... 24
Задание 10 – Проверка правильности работы программы ... 24
Задание 11 – Реализация вычислений для любых размеров матрицы ... 25
Задание 12 – Проведение вычислительных экспериментов ... 28
Контрольные вопросы ... 29
Задания для самостоятельной работы ... 30
Приложение 1. Программный код последовательного приложения для умножения матрицы на вектор ... 30
Приложение 2 – Программный код параллельного приложения для умножения матрицы на вектор ... 32

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.

Цель лабораторной работы

Целью данной лабораторной работы 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

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

При выполнении лабораторной работы предполагается знание раздела 4 "Параллельное программирование на основе MPI", раздела 6 "Принципы разработки параллельных методов" и раздела 7 "Параллельные методы умножения матрицы на вектор" учебных материалов курса. Кроме того, предполагается, что выполнена ознакомительная лабораторная работа "Параллельное программирование с использованием MPI".

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):

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

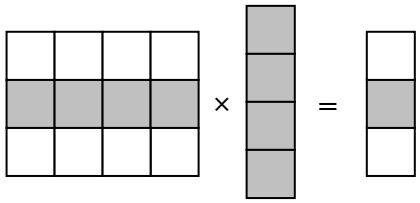

Рис. 1.1. Элемент результирующего вектора – это результат скалярного умножения строки матрицы на вектор

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.

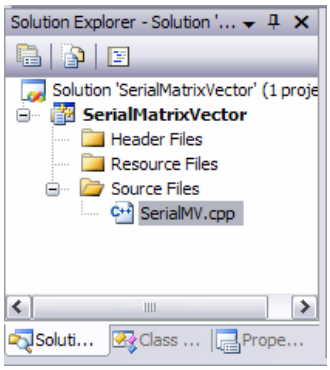

Рис. 1.3. Открытие файла SerialMV.cpp

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 matrix
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 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-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 object'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 object'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 program\n");
  ProcessInitialization(pMatrix, pVector, pResult, Size);
  getch();
}
```

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

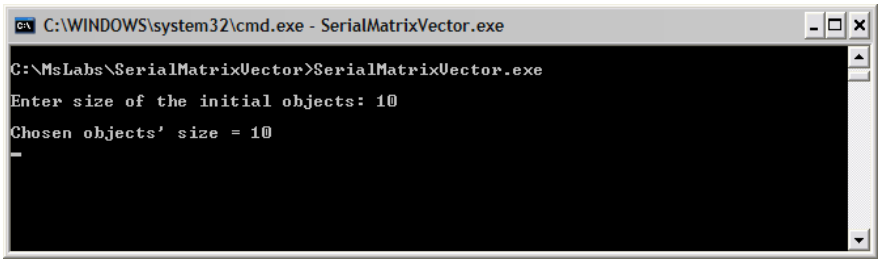

Рис. 1.4. Задание размера объектов

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 введено

положительное 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 objects' 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 object'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' elements
  ProcessInitialization(pMatrix, pVector, pResult, Size);

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

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.

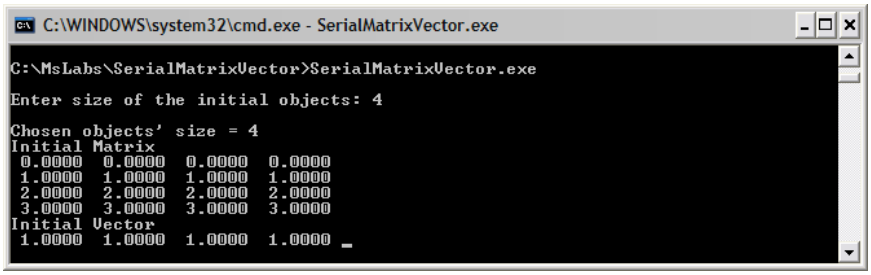

Рис. 1.5. Результат работы программы при завершении задания 3

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* pVector,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' elements
  ProcessInitialization(pMatrix, pVector, pResult, Size);

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

  // 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* pVector, 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' elements
  ProcessInitialization(pMatrix, pVector, pResult, Size);

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

  // 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.

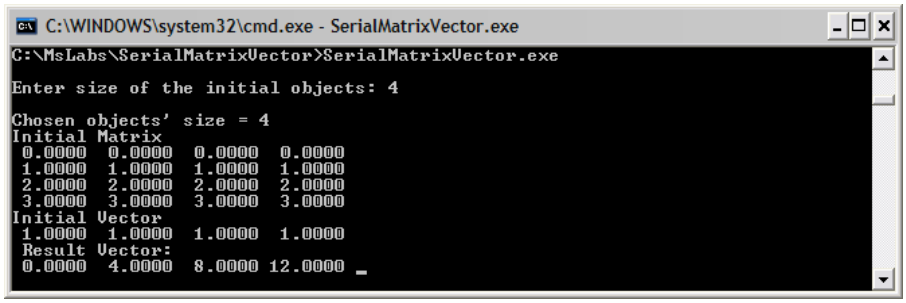

Рис. 1.6. Результат выполнения матрично-векторного умножения

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' 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);
  }
}
```

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 objects' 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

$$N = Size \cdot (2 \cdot Size - 1). \tag{1.2}$$

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:

$$T_1 = N \cdot \tau = Size \cdot (2 \cdot Size - 1) \cdot \tau . \tag{1.3}$$

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 ориентироваться 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), более подробно схема Фостера рассмотрена в разделе 6 учебных материалов курса.

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:

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

где $a_i = (a_{i1}, a_{i2}, ... a_{in}),\ 0 \le i < m,$ 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 разбиении матрица A принимает вид

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

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 $p = s \cdot q$, количество строк матрицы является кратным s, а количество столбцов – кратным q, то есть $m = k \cdot s$ и $n = l \cdot q$. Представим исходную матрицу A в виде набора прямоугольных блоков следующим образом (рис. 1.7в):

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

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

$$A_{ij} = \begin{pmatrix} a_{i_0 j_0} & a_{i_0 j_1} & ...a_{i_0 j_{l-1}} \\ & ... & \\ a_{i_{k-1} j_0} & a_{i_{k-1} j_1} & a_{i_{k-1} j_{l-1}} \end{pmatrix},\ i_v = ik + v,\ 0 \le v < k,\ k = m/s,\ \ j_u = jl + u,\ 0 \le u \le l,\ l = n/q.$$

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.

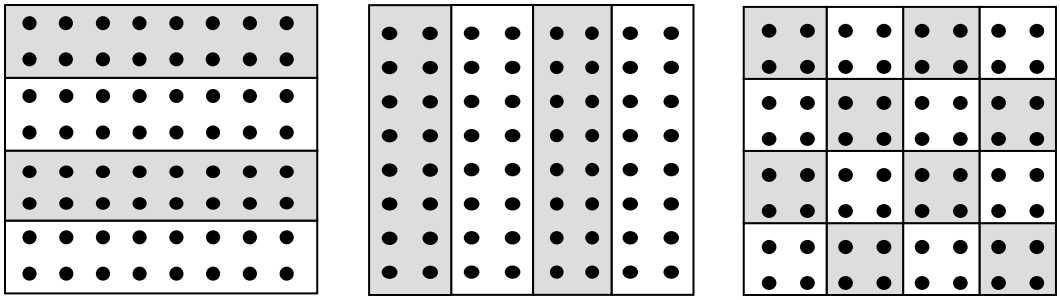

Рис. 1.7. Способы распределения элементов матрицы между процессорами вычислительной системы

Далее в лабораторной работе 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.

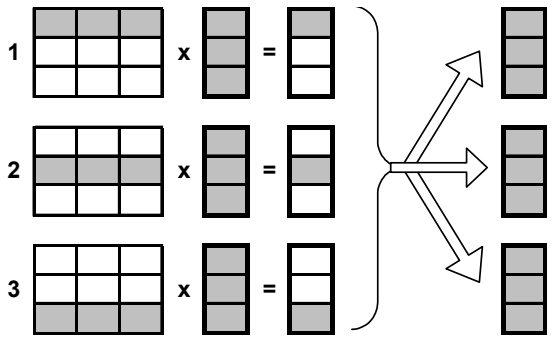

Рис. 1.8. Организация вычислений при выполнении параллельного алгоритма умножения матрицы на вектор, основанного на разделении матрицы по строкам

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.

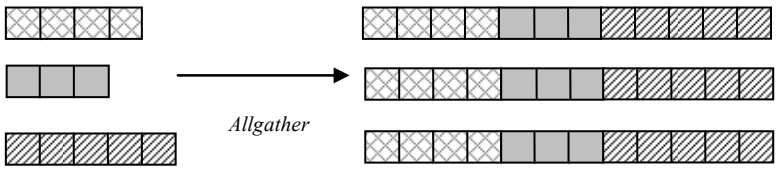

Рис. 1.9. Коллективная коммуникационная операция сбора и обмена данными между всеми процессорами

Упражнение 4 – Реализация параллельного алгоритма матрично-векторного 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, более подробно данная информация рассмотрена в разделе 4 учебных материалов курса.

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.

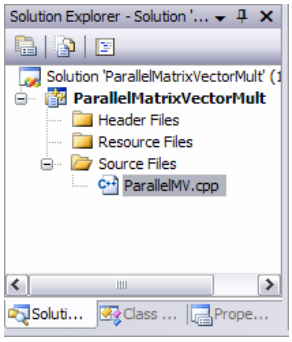

Рис. 1.10. Открытие файла ParallelMV.cpp с использованием Solution Explorer

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 лабораторной работы). 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-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 program\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).

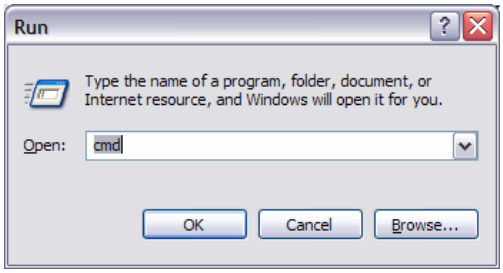

Рис. 1.11. Запуск **Command Prompt**

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

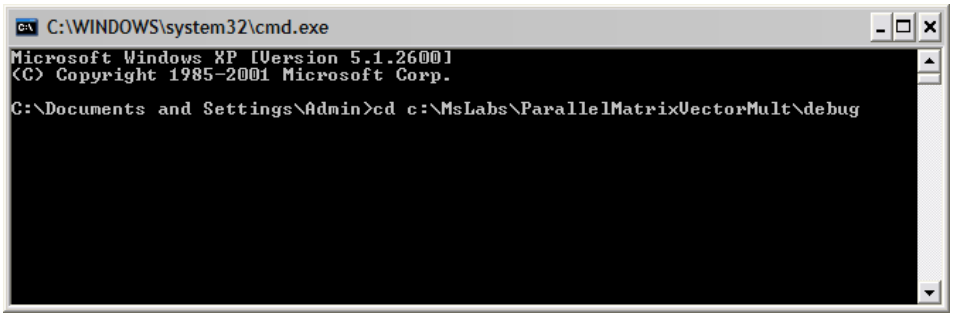

Рис. 1.12. Задание папки, в которой содержится исполняемый модуль параллельной программы

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
```

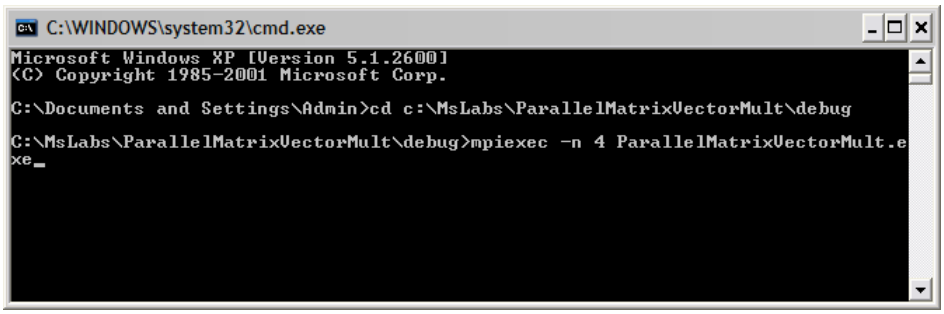

Рис. 1.13. Запуск параллельной программы

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).

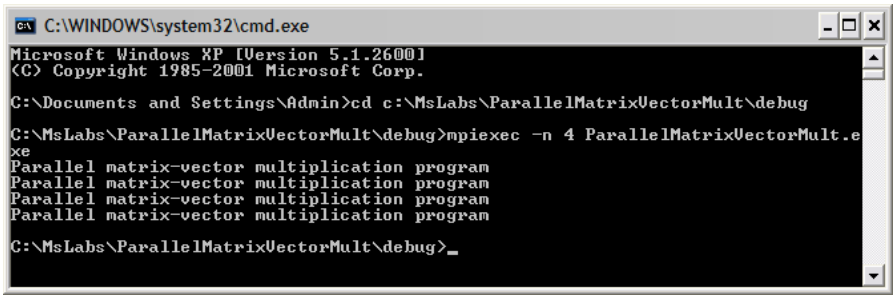

Рис. 1.14. Результат работы первой параллельной программы

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 program\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 program\n")
  printf ("Number of available processes = %d \n", ProcNum);
  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.

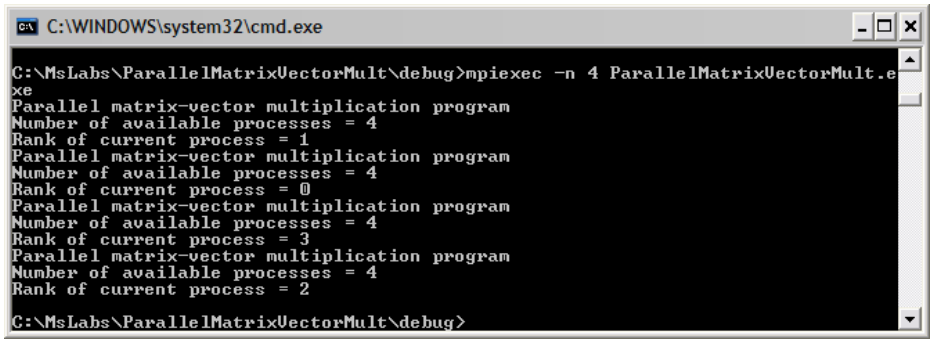

Рис. 1.15. Печать количества и ранга процессов

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 program\n”)
    printf ("Number of abailable processes = %d \n", ProcNum);
  }
  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 поведения - для этого тот фрагмент кода, который производит ввод 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 сообщением (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 program\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 объектов 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 - 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;   // Stripe of the matrix on current process
  double* pProcResult; // Block of result vector on current 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 initialization
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* pVector, 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 between 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 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);
}
```

При нашем подходе матрица хранится в одномерном массиве *pMatrix* 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 рассылку 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 processes
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*.

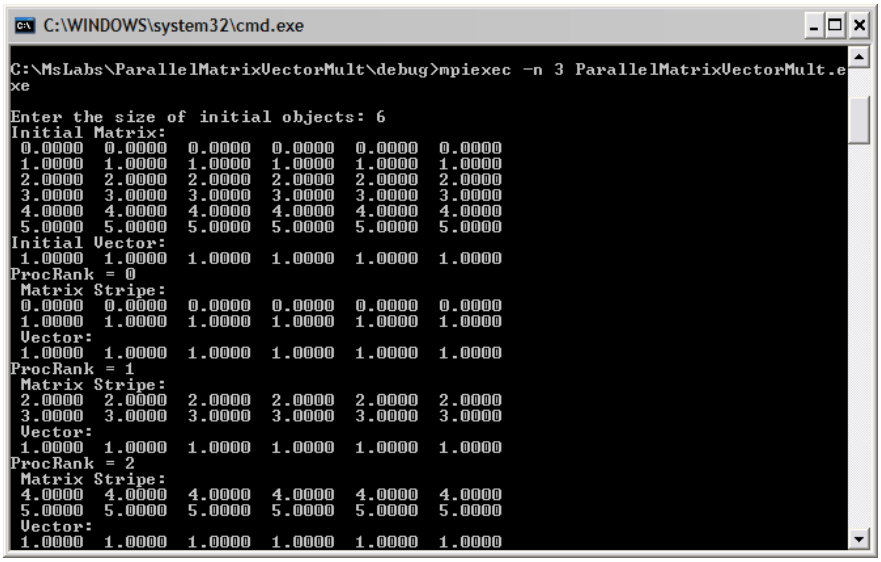

Рис. 1.16. Распределение данных в случае, когда приложение запускается на трех процессах, а порядок матрицы равен шести

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, double* 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 processes
DataDistribution(pMatrix, pProcRows, pVector, Size, RowNum);
TestDistribution(pMatrix, pVector, pProcRows, Size, RowNum);

// Process rows and vector multiplication
ParallelResultCalculation(pProcRows, pVector, pProcResult, 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:

```
// Fuction for testing the results of multiplication of matrix stripe
// by a vector
void TestPartialResults(double* pProcResult, 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
ParallelResultCalculation(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), а на втором процессе – блок (18, 24, 30) (рис. 1.17).

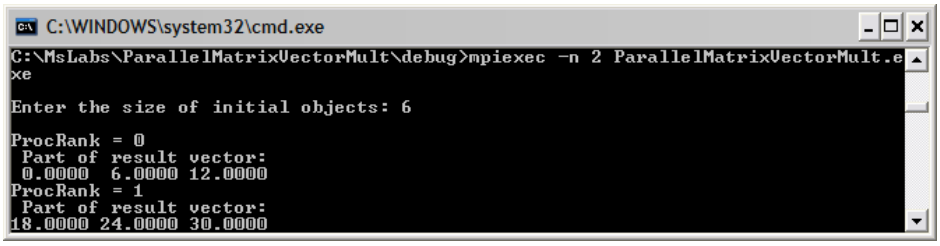

Рис. 1.17. Результат проверки блоков частичных результатов умножения матрицы на вектор в случае, когда для приложения используется два процесса, и размер объектов равен шести

Скомпилируйте и запустите приложение. Проверьте правильность получения частичных результатов по приведенным формулам, задавая разное количество процессов и разные размеры объектов.

Задание 9 – Сбор результатов

На следующем этапе необходимо собрать результирующий вектор из частей, расположенных на разных процессах. Как уже говорилось в упражнении 3, в библиотеке MPI предусмотрена соответствующая функция *MPI_Allgather*, которая собирает из блоков, расположенных на разных процессах коммуникатора, единый массив, и копирует его на все процессы. Функция имеет следующий интерфейс:

```
int MPI_Allgather(void *sbuf,int scount,MPI_Datatype stype,
                  void *rbuf,int rcount,MPI_Datatype rtype, MPI_Comm comm),
где
    - sbuf, scount, stype - параметры передаваемого сообщения,
    - rbuf, rcount, rtype - параметры принимаемого сообщения,
    - comm - коммуникатор, в рамках которого выполняется передача данных.
```

За сбор результатов отвечает функция *ResultReplication*, которая будет состоять только из вызова функции *MPI_Allgather*:

```
// Result vector replication
void ResultReplication(double* pProcResult, double* pResult, int Size,
  int RowNum) {
  MPI_Allgather(pProcResult, RowNum, MPI_DOUBLE, pResult, RowNum,
                MPI_DOUBLE, MPI_COMM_WORLD);
}
```

Вызов функции из основной программы:

```
  ParallelResultCalculation(pProcRows, pVector, pProcResult, Size, RowNum);

  // Result replication
  ResultReplication(pProcResult, pResult, Size, RowNum);
```

После выполнения сбора, добавьте в код основной функции приложения печать результирующего вектора при помощи функции *PrintVector* на всех процессах параллельного приложения. Скомпилируйте и запустите приложение. Оцените правильность его работы.

Задание 10 – 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 векторе *pSerialResult*, а затем поэлементно сравним этот вектор с вектором *pResult*, полученным при помощи параллельного алгоритма. Функция *TestResult* должна иметь доступ к исходным матрице *pMatrix* и вектору *pVector*, а значит может быть выполнена только на ведущем процессе:

```
// Testing the result of parallel matrix-vector multiplication
void TestResult(double* pMatrix, double* pVector, double* pResult,
    int Size) {
  // Buffer for storing the result of serial matrix-vector multiplication
  double* pSerialResult;
  int equal = 0;  // Flag, that shows wheather the vectors are identical
```

```
  int i;          // Loop variable

  if (ProcRank == 0) {
    pSerialResult = new double [Size];
    SerialResultCalculation(pMatrix, pVector, pSerialResult, 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 $\lfloor Size/ProcNum \rfloor$, а остальные - $\lceil Size/ProcNum \rceil$ 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 $\lfloor Size/ProcNum \rfloor$ строк матрицы, а последнему процессу выделяются все оставшиеся строки ($Size - \lfloor Size/ProcNum \rfloor \cdot (ProcNum - 1)$ 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 $\lfloor Size/ProcNum \rfloor$ строк (результат операции $\lfloor\ \rfloor$ совпадает с результатом целочисленного деления переменной *Size* на переменную *ProcNum*). После выполнения этой операции остается распределить $Size - \lfloor Size/ProcNum \rfloor$ 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. $\lfloor RestRows/(ProcNum - i) \rfloor$ строк.

Изменим определение значения переменной *RowNum*:

```
// Function for memory allocation and data initialization
```

```
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 communicator ),
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 current 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 о снования полагать, что время выполнения алгоритма другими процессами несущественно отличается от приведенного.

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

Размер объектов	Последовательный алгоритм	Параллельный алгоритм					
		2 процесса		4 процесса		8 процессов	
		Время	Ускорение	Время	Ускорение	Время	Ускорение
10							

100							
1000							
2000							
3000							
4000							
5000							
6000							
7000							
8000							
9000							
10 000							

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:

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

(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. В качестве значений латентности и пропускной способности следует использовать величины, полученные при выполнении лабораторной работы "Выполнение заданий под управлением Microsoft Compute Cluster Server 2003".

Вычислите теоретическое время выполнения параллельного алгоритма по формуле (1.4). Результаты занесите в таблицу:

Размер матрицы	2 процесса		4 процесса		8 процессов	
	Модель	Эксперимент	Модель	Эксперимент	Модель	Эксперимент
10						
100						
1000						
2000						
3000						
4000						
5000						
6000						
7000						
8000						
9000						
10 000						

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.

Приложение 1. 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 object'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* pVector, 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* pVector,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 program\n");
  // Memory allocation and definition of objects' elements
  ProcessInitialization(pMatrix, pVector, pResult, Size);

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

  // 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\n", duration);

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

```
}
```

Приложение 2 – 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 processes
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 initialization
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 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;
}

// 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 current 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,  double*  pVector,  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 calculating partial matrix-vector multiplication
void ParallelResultCalculation(double* pProcRows, double* 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, double* 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, pSerialResult, 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* pVector, 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, 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();
}
```