Нижегородский государственный университет им. Н.И.Лобачевского

Факультет Вычислительной математики и кибернетики

Образовательный комплекс

Введение в методы параллельного программирования

Лабораторная работа 1.

Параллельные алгоритмы матрично-векторного умножения

Гергель В.П., профессор, д.т.н.
Кафедра математического обеспечения ЭВМ

Содержание

Упражнения:

- Постановка задачи
- Реализация последовательного алгоритма умножения матрицы на вектор
- Разработка параллельного алгоритма умножения матрицы на вектор
- Реализация параллельного алгоритма матрично-векторного умножения

Упражнение 1: Постановка задачи…

Умножение матрицы на вектор

$$c = A \cdot b$$

или

$$\begin{pmatrix} c_0 \\ c_1 \\ c_{m-1} \end{pmatrix} = \begin{pmatrix} a_{0,0}, & a_{0,1}, & \ldots, & a_{0,n-1} \\ & & \ldots & \\ a_{m-1,0}, & a_{m-1,1}, & \ldots, & a_{m-1,n-1} \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_{n-1} \end{pmatrix}$$

↳ Задача умножения матрицы на вектор может быть сведена к выполнению ***m*** независимых операций умножения строк матрицы ***A*** на вектор ***b***

$$c_i = (a_i, b) = \sum_{j=1}^{n} a_{ij} b_j, \quad 0 \le i < m$$

Упражнение 1: Постановка задачи

```
// 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]
  }
}
```

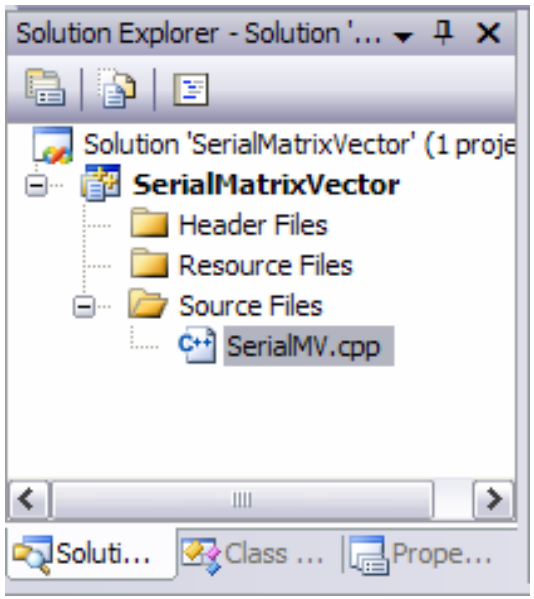
- Для выполнения матрично-векторного умножения необходимо выполнить ***m*** операций вычисления скалярного произведения
- Трудоемкость вычислений имеет порядок *O(mn)*

Упражнение 2: Реализация последовательного алгоритма умножения матрицы на вектор…

- **Поэтапная разработка последовательного алгоритма:**
  - Задание 1 – Открытие проекта **SerialMatrixVectorMult**
  - Задание 2 – Ввод размеров объектов
  - Задание 3 – Ввод данных
  - Задание 4 – Завершение процесса вычислений
  - Задание 5 – Реализация умножения матрицы на вектор
  - Задание 6 – Проведение вычислительных экспериментов

Упражнение 2: Реализация последовательного алгоритма умножения матрицы на вектор…

- **Задание 1** – Открытие проекта **SerialMatrixVectorMult**:…
  - Запустите приложение Microsoft Visual Studio 2005,
  - Откройте проект **SerialMatrixVector.sln**, расположенный в папке **C:\MsLabs\SerialMatrixVector**,
  - При помощи окна **Solution Explorer** откройте файл **SerialMV.cpp**



Упражнение 2: Реализация последовательного алгоритма умножения матрицы на вектор…

- **Задание 1** – Открытие проекта **SerialMatrixVectorMult**:
  - Переменные, которые будут использоваться в программе:

```
double* pMatrix;  // Initial matrix
double* pVector;  // Initial vector
double* pResult;  // Result vector
int Size;         // Sizes of initial matrix and vector
```

  - Вывод начального сообщения и ожидание нажатия любой клавиши перед завершением выполнения приложения:

```
printf ("Serial matrix-vector multiplication program\n");
getch();
```

Упражнение 2: Реализация последовательного алгоритма умножения матрицы на вектор…

- **Задание 2** – Ввод размеров объектов:…
  - Для задания исходных данных реализуем функцию *ProcessInitialization*:
    - определяет размеры матрицы и вектора,
    - выделяет память для всех объектов, участвующих в умножении (*pMatrix, pVector* и *pResult*),
    - задает значения элементов исходных матрицы и вектора

```
// Function for process intialization
void ProcessInitialization (double* &pMatrix,
  double* &pVector, double* &pResult, int &Size);
```

Упражнение 2: Реализация последовательного алгоритма умножения матрицы на вектор…

- **Задание 2** – Ввод размеров объектов:…
  - Определение размеров объектов (задание значения переменной *Size*):

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

Упражнение 2: Реализация последовательного алгоритма умножения матрицы на вектор…

- **Задание 2** – Ввод размеров объектов:…
  - Определение размеров объектов (задание значения переменной *Size*) с контролем ошибочных ситуаций:

```
// Function for process initialization
void ProcessInitialization (double* &pMatrix,
  double* &pVector, double* &pResult, int &Size) {
 // Setting the size of the initial matrix and the 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);
}
```

Упражнение 2: Реализация последовательного алгоритма умножения матрицы на вектор…

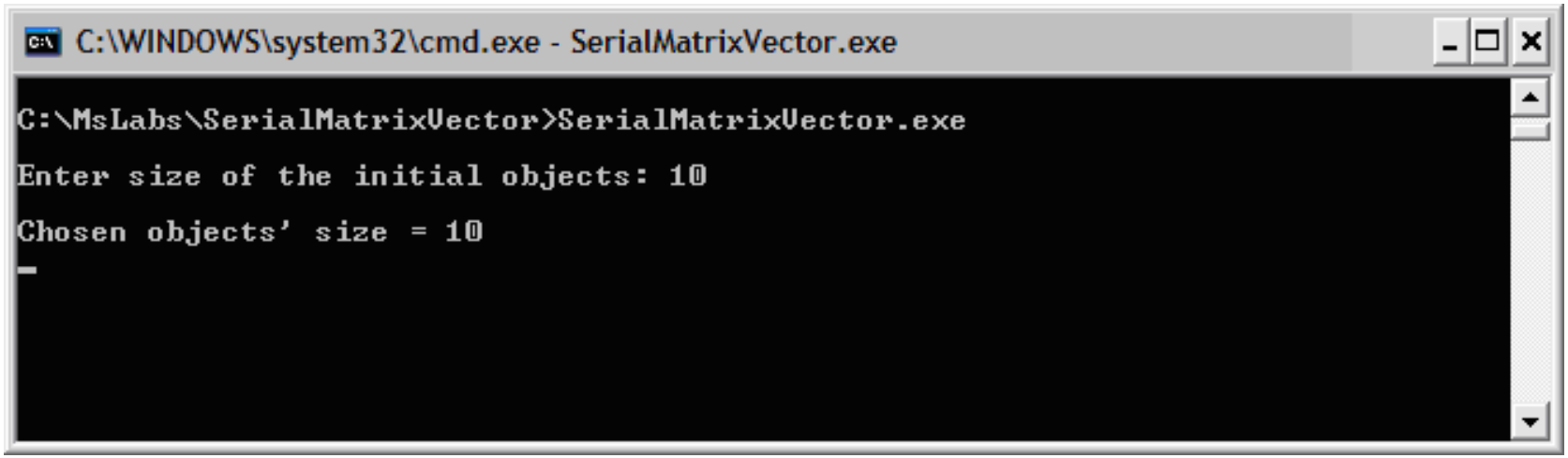
- **Задание 2** – Ввод размеров объектов:…
  - Добавьте вызов функции *ProcessInitialization* в основную функцию приложения:

```
void main() {
  double* pMatrix;  // Initial matrix
  double* pVector;  // Initial vector
  double* pResult;  // Result vector
  int Size;         // Sizes of initial matrix and vector

  printf ("Serial matrix-vector multiplication program\n");
  // Process initializattion
  ProcessInitialization(pMatrix, pVector, pResult, Size);
  getch();
}
```

Упражнение 2: Реализация последовательного алгоритма умножения матрицы на вектор…

- **Задание 2** – Ввод размеров объектов:
  - Скомпилируйте и запустите приложение:



Упражнение 2: Реализация последовательного алгоритма умножения матрицы на вектор…

- **Задание 3** – Ввод данных:…
  - Выделение памяти:

```
// Function for process initialization
void ProcessInitialization (double* &pMatrix,
  double* &pVector, double* &pResult, int &Size) {
  // Setting the size of the initial matrix and the vector
  <…>

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

Упражнение 2: Реализация последовательного алгоритма умножения матрицы на вектор…

- **Задание 3** – Ввод данных:…
  - Определение значений элементов исходных матриц и вектора по шаблону:

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

Упражнение 2: Реализация последовательного алгоритма умножения матрицы на вектор…

- **Задание 3** – Ввод данных:…
  - Функция для простого определения значений элементов исходных матрицы и вектора:

```
// Function for simple data initialization
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;
  }
}
```

Упражнение 2: Реализация последовательного алгоритма умножения матрицы на вектор…

- **Задание 3** – Ввод данных:…
  - Вызов функции задания элементов матрицы и вектора из функции *ProcessInitialization*:

```
// Function for process initialization
void ProcessInitialization (double* &pMatrix,
  double* &pVector, double* &pResult, int &Size) {
  // Setting the size of the initial matrix and the vector
  <…>

 // Memory allocation
  <…>

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

Упражнение 2: Реализация последовательного алгоритма умножения матрицы на вектор…

- **Задание 3** – Ввод данных:…
  - Для контроля правильности задания исходных данных необходимо разработать:
    - Функцию для форматированной печати матрицы *PrintMatrix* (входные параметры - матрица *pMatrix*, количество строк *RowCount* и количество столбцов *ColCount*),
    - Функцию для форматированной печати вектора *PrintVector* (входные параметры - вектор *pVector* и количество элементов в векторе *Size*)

Упражнение 2: Реализация последовательного алгоритма умножения матрицы на вектор…

- **Задание 3** – Ввод данных:…
  - Форматированная печать матрицы:

```
// 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");
  }
}
```

Упражнение 2: Реализация последовательного алгоритма умножения матрицы на вектор…

- **Задание 3** – Ввод данных:…
  - Форматированная печать вектора:

```
// 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");
}
```

Упражнение 2: Реализация последовательного алгоритма умножения матрицы на вектор…

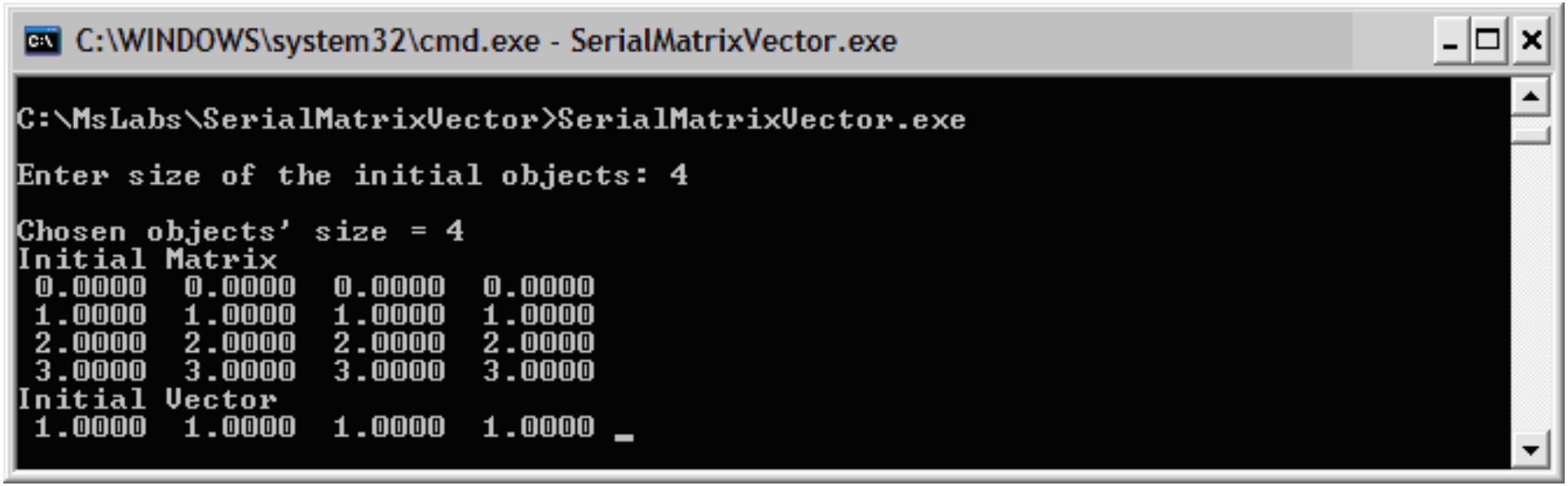
- **Задание 3** – Ввод данных:…
  - Контроль правильности выполнения этапа задания исходных данных:

```
// Process initialization
ProcessInitialization(pMatrix, pVector, pResult, Size);

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

Упражнение 2: Реализация последовательного алгоритма умножения матрицы на вектор…

- **Задание 3** – Ввод данных^
  - Контроль правильности выполнения этапа задания исходных данных:



Упражнение 2: Реализация последовательного алгоритма умножения матрицы на вектор…

- **Задание 4** – Завершение процесса вычислений:…
  - Функция для корректного завершения процесса вычислений *ProcessTermination*:
    - Освобождает память, выделенную в ходе выполнения программы,
    - Входные параметры - матрица pMatrix и векторы pVector и pResult

Упражнение 2: Реализация последовательного алгоритма умножения матрицы на вектор…

- **Задание 4** – Завершение процесса вычислений:
  - Интерфейс и реализация функции *ProcessTermination*:

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

  - Вызов функции завершения вычислительного процесса:

```
  // Matrix and vector output
  <…>
  // Computational process termination
  ProcessTermination(pMatrix, pVector, pResult);
```

Упражнение 2: Реализация последовательного алгоритма умножения матрицы на вектор…

- **Задание 5** – Реализация умножения матрицы на вектор:…
  - Функция *ResultCalculation* выполняет умножение матрицы на вектор в соответствии с алгоритмом, изложенным в упражнении 1:

```
// 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];
  }
}
```

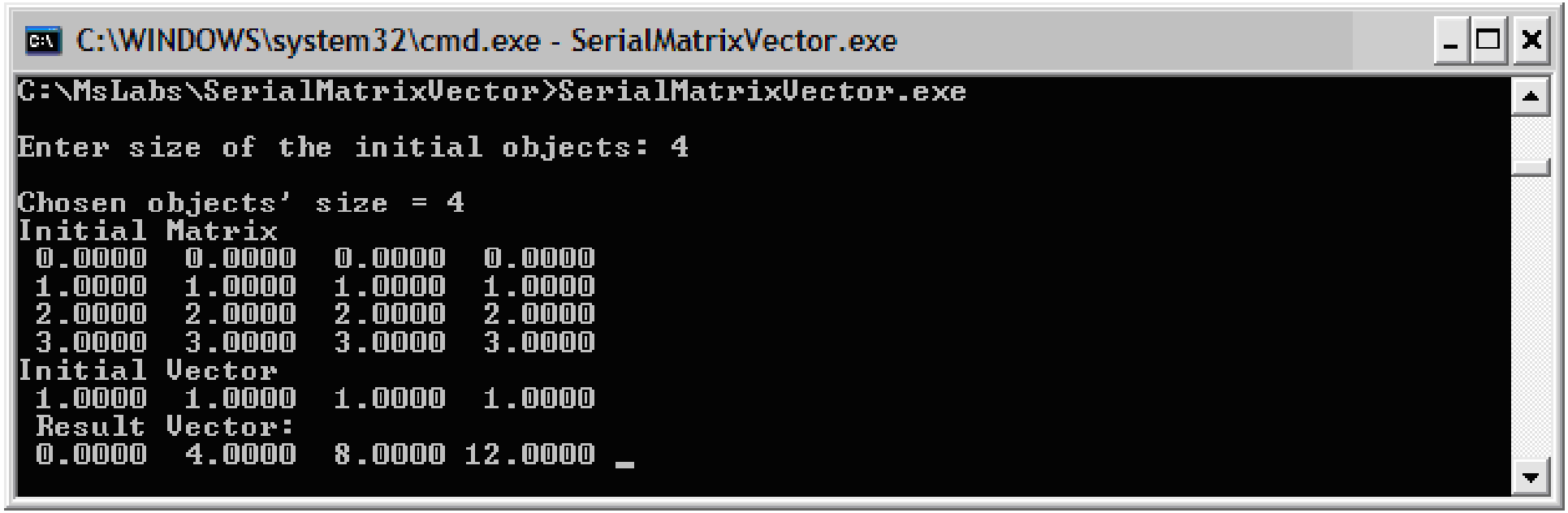
Упражнение 2: Реализация последовательного алгоритма умножения матрицы на вектор…

- **Задание 5** – Реализация умножения матрицы на вектор:…
  - Добавьте вызов функции выполнения матрично-векторного умножения в основную функцию программы,
  - Для контроля правильности выполнения умножения матрицы на вектор, распечатайте результирующий вектор:

```
// Matrix and vector output
<…>
// Matrix-vector multiplication
ResultCalculation(pMatrix, pVector, pResult, Size);
// Printing the result vector
printf (“\n Result Vector: \n”);
PrintVector(pResult, Size);
```

Упражнение 2: Реализация последовательного алгоритма умножения матрицы на вектор…

- **Задание 5** – Реализация умножения матрицы на вектор:



Упражнение 2: Реализация последовательного алгоритма умножения матрицы на вектор…

- **Задание 6** – Проведение вычислительных экспериментов:…
  - Функция для задания значений элементов матрицы и вектора при помощи датчика случайных чисел:

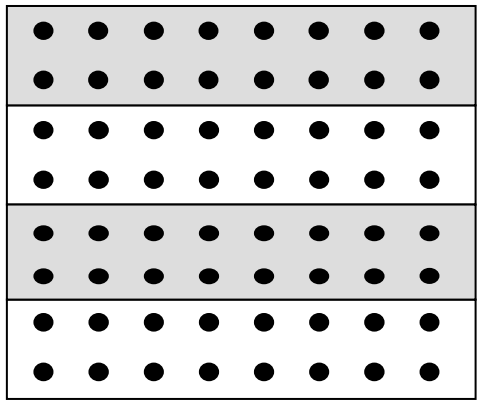
```
// Function for random initialization of object elements
void RandomDataInitialization (double* pMatrix,
  double* pVector,int Size) {
  int i, j;  // Loop variables
  srand(unsigned(clock()));
  for (i=0; i<Size; i++) {
    pVector[i] = rand()/double(1000);
    for (j=0; j<Size; j++)
      pMatrix[i*Size+j]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

Упражнение 2: Реализация последовательного алгоритма умножения матрицы на вектор

- **Задание 6** – Проведение вычислительных экспериментов:
  - Замените вызов функции *DummyDataInitialization* на *RandomDataInitialization* в функции *ProcessInitialization*,
  - Добавьте вычисление и вывод времени выполнения умножения,
  - Измерьте времена работы алгоритма умножения матрицы на вектор при различных количествах исходных данных,
  - Рассчитайте теоретическое время работы последовательного алгоритма,
  - Заполните таблицу результатов вычислений

Упражнение 3: Разработка параллельного алгоритма умножения матрицы на вектор…

- **Распределение данных** – ленточная схема (разбиение матрицы по строкам)

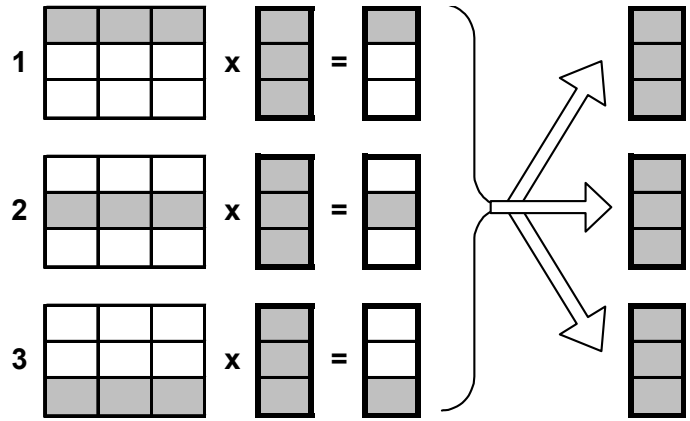


- **Базовая подзадача** - операция скалярного умножения одной строки матрицы на вектор

$$c_i = (a_i, b) = \sum_{j=1}^{n} a_{ij} b_j, \quad 0 \le i < m$$

Упражнение 3: Разработка параллельного алгоритма умножения матрицы на вектор…

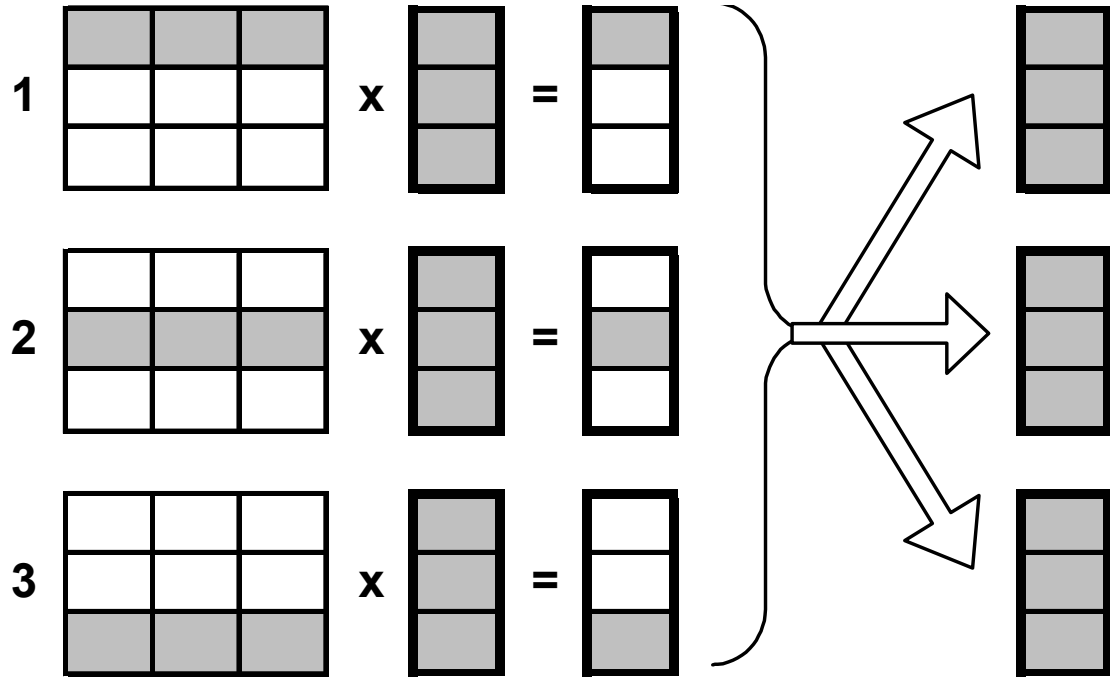
- **Выделение информационных зависимостей:**
  - Базовая подзадача для выполнения вычисления должна содержать строку матрицы ***A*** и копию вектора ***b***. После завершения вычислений каждая базовая подзадача будет содержать один из элементов вектора результата ***c***



  - Для объединения результатов расчетов и получения полного вектора ***c*** на каждом из процессоров вычислительной системы необходимо выполнить операцию обобщенного сбора данных

Упражнение 3: Разработка параллельного алгоритма умножения матрицы на вектор…

- Схема информационного взаимодействия



Упражнение 3: Разработка параллельного алгоритма умножения матрицы на вектор…

- **Масштабирование и распределение подзадач по процессорам:**
  - Если число процессоров ***p*** меньше числа базовых подзадач ***m (p<m)***, базовые подзадачи могут быть укрупнены с тем, чтобы каждый процессор выполнял несколько операций умножения строк матрицы ***A*** и вектора ***b***. В этом случае, по окончании вычислений каждая базовая подзадача будет содержать набор элементов результирующего вектора ***c***
  - Распределение подзадач между процессорами вычислительной системы может быть выполнено с учетом возможности эффективного выполнения операции обобщенного сбора данных

Упражнение 4: Реализация параллельного алгоритма умножения матрицы на вектор…

Понятие параллельной программы:

- Под *параллельной программой* в рамках MPI понимается множество одновременно выполняемых *процессов*:
  - Процессы могут выполняться на разных процессорах; вместе с этим, на одном процессоре могут располагаться несколько процессов,
  - Каждый процесс параллельной программы порождается на основе копии одного и того же программного кода (*модель SPMP*).
- Количество процессов и число используемых процессоров определяется в момент запуска параллельной программы средствами среды исполнения MPI программ. Все процессы программы последовательно перенумерованы. Номер процесса именуется *рангом* процесса.

Упражнение 4: Реализация параллельного алгоритма умножения матрицы на вектор…

Понятие коммуникаторов:

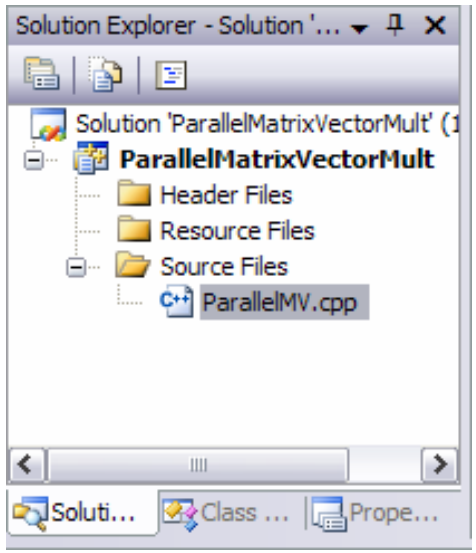
- *Коммуникатор* в MPI - специально создаваемый служебный объект, объединяющий в своем составе группу процессов и ряд дополнительных параметров (*контекст)*:
  - парные операции передачи данных выполняются для процессов, принадлежащих одному и тому же коммуникатору,
  - Коллективные операции применяются одновременно для всех процессов коммуникатора.
- Указание используемого коммуникатора является обязательным для операций передачи данных в MPI,
- В ходе вычислений могут создаваться новые и удаляться существующие коммуникаторы,
- Все имеющиеся в параллельной программе процессы входят в состав создаваемого по умолчанию коммуникатора с идентификатором MPI_COMM_WORLD.

Упражнение 4: Реализация параллельного алгоритма умножения матрицы на вектор…

- **Поэтапная разработка параллельного алгоритма:**
  - Задание 1 – Открытие проекта **ParallelMatrixVectorMult**
  - Задание 2 – Инициализация и завершение параллельной программы
  - Задание 3 – Определение количества процессов
  - Задание 4 – Ввод размера матрицы и вектора
  - Задание 5 – Ввод исходных данных
  - Задание 6 – Завершение процесса вычислений
  - Задание 7 – Распределение данных между процессами
  - Задание 8 – Реализация умножения матрицы на вектор
  - Задание 9 – Сбор результатов
  - Задание 10 – Проверка правильности работы программы
  - Задание 11 – Реализация вычислений для любых размеров матрицы
  - Задание 12 – Проведение вычислительных экспериментов

Упражнение 4: Реализация параллельного алгоритма умножения матрицы на вектор…

- **Задание 1** – Открытие проекта **ParallelMatrixVectorMult**:…
  - Запустите приложение Microsoft Visual Studio 2005
  - Откройте проект **ParallelMatrixVectorMult.sln**, расположенный в папке **C:\MsLabs\ParallelMatrixVectorMult**
  - При помощи окна **Solution Explorer** откройте файл **ParallelMV.cpp**



Упражнение 4: Реализация параллельного алгоритма умножения матрицы на вектор…

- **Задание 1** – Открытие проекта **ParallelMatrixVectorMult**:
  - Проект содержит функции:
    - *DummyDataInitialization* – начальное задание исходных данных,
    - *RandomDataInitialization* – задание исходных данных при помощи датчика случайных чисел,
    - *ResultCalculation* – последовательный алгоритм матрично-векторного умножения,
    - *PrintMatrix* и *PrintVector* – печать матрицы и вектора
  - В главной функции приложения объявлены переменные *pMatrix*, *pVector*, *pResult*, *Size*
  - Скомпилируйте и запустите приложение. Убедитесь, что на экран выводится сообщение

```
"Parallel matrix-vector multiplication program"
```

Упражнение 4: Реализация параллельного алгоритма умножения матрицы на вектор…

- **Задание 2** – Инициализация и завершение параллельной программы:…
  - Подключите заголовочный файл библиотеки MPI **mpi.h**,
  - Выполните инициализацию и завершение MPI программы выполняется при помощи функций:

```
void main (int argc, char* argv[]) {
  //Variable declaration
  <…>
  MPI_Init(&argc, &argv);
  printf("Parallel matrix-vector multiplication program\n");
  MPI_Finalize();
}
```

Упражнение 4: Реализация параллельного алгоритма умножения матрицы на вектор:…

- **Задание 2** – Инициализация и завершение параллельной программы:…
  - Скомпилируйте параллельное приложение,
  - Запустите программу на выполнение:
    - Нажмите кнопку **Пуск**, а затем **Выполнить**,
    - В появившемся диалоговом окне наберите название программы **cmd**,
    - В командной строке перейдите в папку, где содержится исполняемый модуль разработанной программы,
    - Запуск MPI-программы осуществляется при помощи вызова утилиты mpiexec. Формат вызова в общем виде выглядит следующим образом:

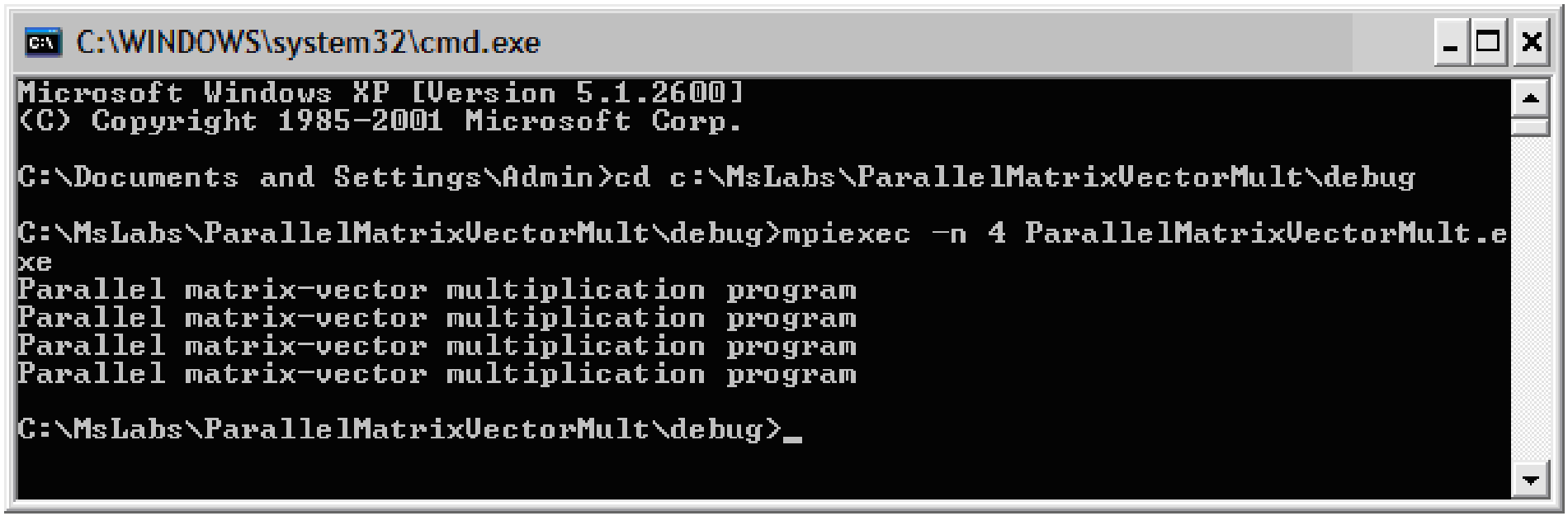
```
mpiexec –n <Num of processes> <Name of executive> <Arguments>
```

    - Для запуска параллельной программы, состоящей из 4 процессов, наберите команду:

```
mpiexec –n 4 ParallelMatrixVectorMult.exe
```

Упражнение 4: Реализация параллельного алгоритма умножения матрицы на вектор:…

- **Задание 2** – Инициализация и завершение параллельной программы:
  - Запуск параллельного приложения



Упражнение 4: Реализация параллельного алгоритма умножения матрицы на вектор…

- **Задание 3** – Определение количества процессов:…
  - Объявим глобальные переменные:
    - *ProcNum* – число процессов параллельной программы,
    - *ProcRank* – ранг текущего процесса.
  - Определим число процессов и ранг процесса при помощи специальных функций MPI:

```
int ProcNum;      // Number of available processes
int ProcRank;     // Rank of current process
void main(int argc, char* argv[]) {
  <…>
  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();
}
```

Упражнение 4: Реализация параллельного алгоритма умножения матрицы на вектор…

- **Задание 3** – Определение количества процессов:
  - Измените код главной функции параллельного приложения таким образом, чтобы:
    - Начальное сообщение и количество процессов было напечатано только ведущим процессом (процессом с рангом 0),
    - Каждый процесс распечатал свой ранг

Упражнение 4: Реализация параллельного алгоритма умножения матрицы на вектор…

- **Задание 4** – Ввод размера матрицы и вектора:…
  - Для ввода исходных данных, как и ранее, служит функция *ProcessInitialization*:

```
// Function for process initialization
void ProcessInitialization (double* &pMatrix,
  double* &pVector, double* &pResult, int &Size);
```

  - Размер матрицы и вектора должен быть больше числа доступных процессов и должен делиться на число процессов нацело
  - Ввод размеров матрицы и вектора должен быть организован диалог только в ведущем процессе параллельной программы

Упражнение 4: Реализация параллельного алгоритма умножения матрицы на вектор…

- **Задание 4** – Ввод размера матрицы и вектора:…

```
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));
  }
}
```

Упражнение 4: Реализация параллельного алгоритма умножения матрицы на вектор…

- **Задание 4** – Ввод размера матрицы и вектора:…
  - После того, как размер матрицы и вектора корректно определен на одном из процессов, необходимо передать значение переменной Size на все процессы при помощи функции широковещательной рассылки:

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

  - Функция MPI_Bcast определяет коллективную операцию, она должна быть вызвана во всех процессах коммуникатора.

Упражнение 4: Реализация параллельного алгоритма умножения матрицы на вектор…

- **Задание 4** – Ввод размера матрицы и вектора:
  - Добавьте вызов функции инициализации процесса вычислений в главную функцию параллельного приложения,
  - Распечатайте значение переменной *Size* на всех процессах,
  - Скомпилируйте приложение. Выполните несколько запусков, указывая различное число процессов и разные размеры объектов. Убедитесь в том, что размер задается корректно.

Упражнение 4: Реализация параллельного алгоритма умножения матрицы на вектор…

- **Задание 5** – Ввод исходных данных:…
  - Согласно вычислительной схеме параллельного алгоритма, матрица разделяется между процессами на горизонтальные полосы: на каждом процессе содержится полоса матрицы *pProcRows*, содержащая *RowNum* строк,
  - Исходный вектор *pVector* копируется на все процессы,
  - В результате умножения полосы матрицы на вектор, получается блок результирующего вектора *pProcResult*, который состоит из *RowNum* элементов

Упражнение 4: Реализация параллельного алгоритма умножения матрицы на вектор…

- **Задание 5** – Ввод исходных данных:…
  - Объявление 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* 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
  <…>
}
```

Упражнение 4: Реализация параллельного алгоритма умножения матрицы на вектор…

- **Задание 5** – Ввод исходных данных:
  - Выделение памяти для объектов происходит в функции *ProcessInitialization*:

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

  - Исходная матрица существует только на ведущем процессе, остальные объекты доступны на всех процессах параллельной программы
  - Задание значений элементов исходных матрицы и вектора осуществляется только на ведущем процессе при помощи функции *RandomDataInitialization* или *DummyDataInitialization*

Упражнение 4: Реализация параллельного алгоритма умножения матрицы на вектор…

- **Задание 6** – Завершение процесса вычислений:
  - Для завершения вычислительных процессов предназначена функция *ProcessTermination*,
  - Освобождает память, выделенную в процессе выполнения параллельной программы:

```
// Function for computational process termination
void ProcessTermination (double* pMatrix, double* pVector,
  double* pResult, double* pProcRows, double* pProcResult);
```

Упражнение 4: Реализация параллельного алгоритма умножения матрицы на вектор…

- **Задание 7** – Распределение данных между процессами:…
  - Для распределения данных предназначена функция *DataDistribution*,
  - Вектор *pVector* копируется с ведущего процесса на все процессы параллельного приложения при помощи широковещательной рассылки *MPI_Bcast*,
  - Матрица *pMatrix* разделяется между процессами при помощи функции обобщенной передачи данных от одного процесса всем процессам *MPI_Scatter*:

```
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.
```

Упражнение 4: Реализация параллельного алгоритма умножения матрицы на вектор…

- **Задание 7** – Распределение данных между процессами:…

```
// Function for data distribution 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);
}
```

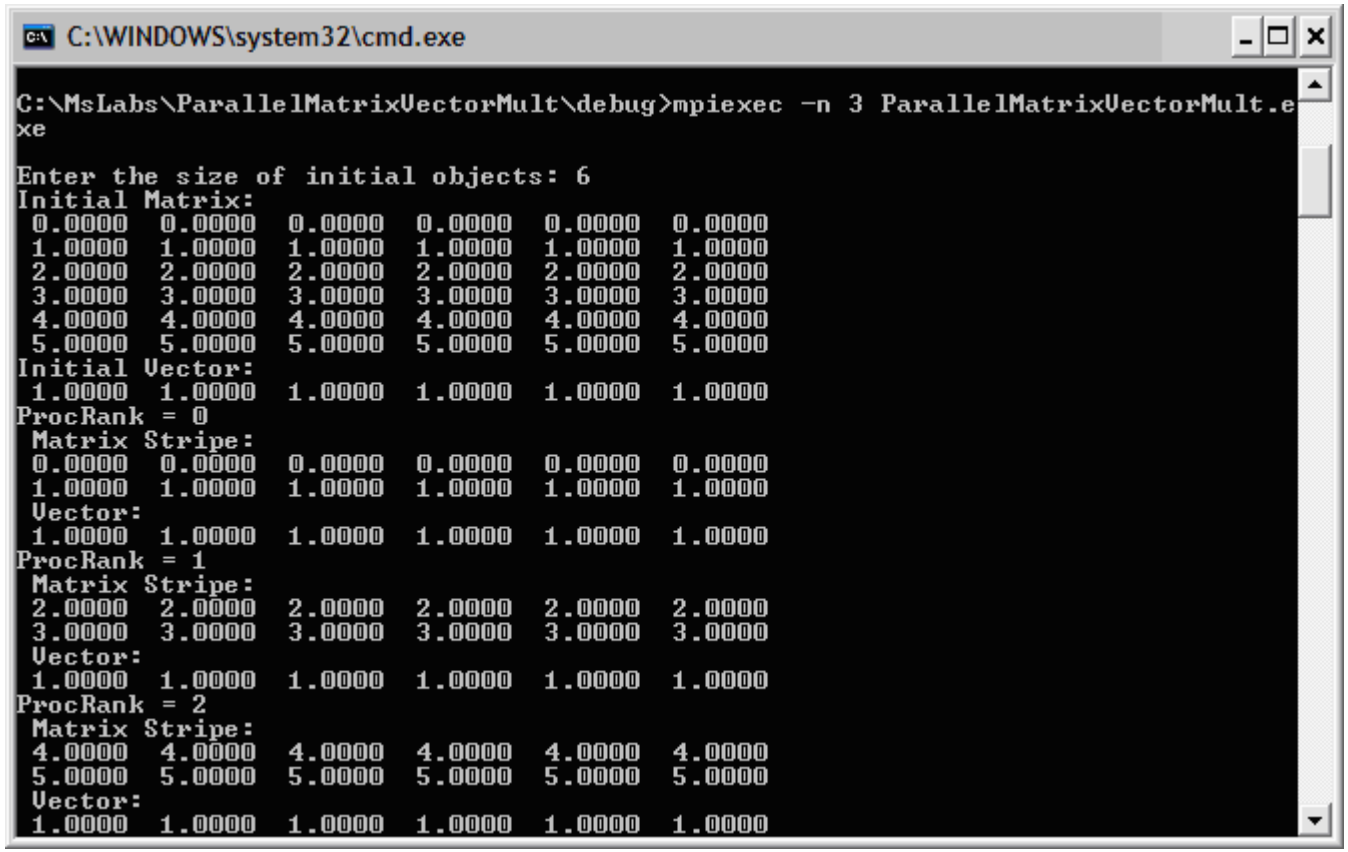
Упражнение 4: Реализация параллельного алгоритма умножения матрицы на вектор…

- **Задание 7** – Распределение данных между процессами:…
  - Добавьте вызов функции *DataDistribution* в главную функцию параллельного приложения,
  - Разработайте функцию проверки правильности выполнения этапа разделения данных *TestDistribution*:
    - Печать матрицы pMatrix и вектора pVector на ведущем процессе,
    - Последовательная печать полос матрицы pProcRows и вектора pVector в процессах параллельной программы

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

Упражнение 4: Реализация параллельного алгоритма умножения матрицы на вектор…

- **Задание 7** – Распределение данных между процессами:
  - Проверка правильности распределения данных



Упражнение 4: Реализация параллельного алгоритма умножения матрицы на вектор…

- **Задание 8** – Реализация умножения матрицы на вектор:…
  - Для выполнения умножения полосы матрицы на вектор каждым процессом реализуем функцию *ParallelResultCalculation*

```
// Function for parallel matrix-vector multiplication
void ParallelResultCalculation (double* pProcRows,
  double* pVector, double* pProcResult, int Size,
  int RowNum);
```

  - Функция вычисляет элементы блока результирующего вектора *pProcResult* путем скалярного умножения строк матрицы, принадлежащих полосе данного процесса, на вектор

Упражнение 4: Реализация параллельного алгоритма умножения матрицы на вектор…

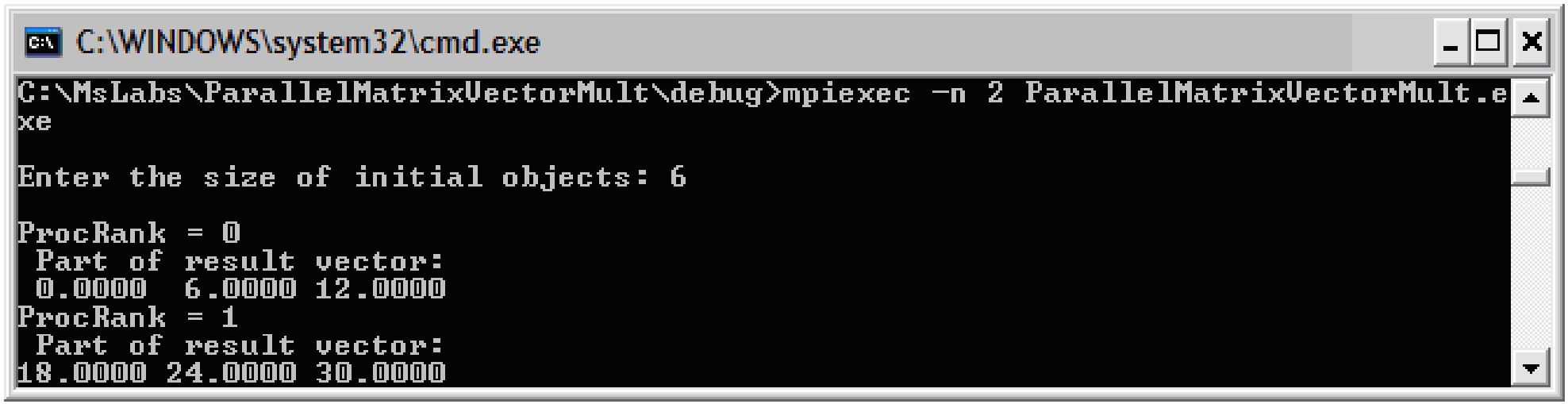
- **Задание 8** – Реализация умножения матрицы на вектор:…
  - Для контроля правильности выполнения умножения разработайте функцию *TestPartialResults*, которая последовательно распечатает блоки результирующего вектора со всех процессов:

```
void TestPartialResults (double* pProcResult, int RowNum);
```

  - Добавьте вызов функций *ParallelResultCalculation* и *TestPartialResults* в главную функцию параллельного приложения
  - Скомпилируйте и запустите приложение

Упражнение 4: Реализация параллельного алгоритма умножения матрицы на вектор…

- **Задание 8** – Реализация умножения матрицы на вектор:
  - На процессе с рангом *i* должен быть получен блок результирующего вектора, содержащий элементы в диапазоне от *Size*(i*RowNum)* до *Size*((i+1)*RowNum-1)*



Упражнение 4: Реализация параллельного алгоритма умножения матрицы на вектор…

- **Задание 9** – Сбор результатов:…
  - Собрать результирующий вектор из частей, расположенных на разных процессах, можно при помощи функции *MPI_Allgather*

```
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

Упражнение 4: Реализация параллельного алгоритма умножения матрицы на вектор…

- **Задание 9** – Сбор результатов:
  - 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);
}
```

  - 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.

Упражнение 4: Реализация параллельного алгоритма умножения матрицы на вектор…

- **Задание 10** – Проверка правильности работы программы:…
  - Функция *TestResult* сравнивает результаты работы последовательного и параллельного алгоритмов путем поэлементного сравнения полученных векторов:

```
// Testing the parallel matrix-vector multiplication
void TestResult (double* pMatrix, double* pVector,
  double* pResult, int Size)
```

  - 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.

Упражнение 4: Реализация параллельного алгоритма умножения матрицы на вектор…

- **Задание 10** – Проверка правильности работы программы:
  - Закомментируйте использовавшуюся ранее отладочную печать,
  - Реализуйте функцию *TestResult*,
  - Добавьте вызов функции *TestResult* в функцию *main*,
  - Замените вызов функции *DummyDataInitialization* на вызов функции *RandomDataInitialization* в функции *ProcessInitialization*,
  - Протестируйте работоспособность приложения

Упражнение 4: Реализация параллельного алгоритма умножения матрицы на вектор…

- **Задание 11** – Реализация вычислений для любых размеров матрицы:
  - Внесите необходимые изменения в функции *ProcessInitialization*, *DataDistribution*, *ResultReplication*,
  - Проверьте правильность умножения матрицы на вектор при помощи функции *TestResult*

Упражнение 4: Реализация параллельного алгоритма умножения матрицы на вектор

- **Задание 12** – Проведение вычислительных экспериментов:
  - Добавьте вычисление и вывод времени выполнения умножения матрицы на вектор,
  - Протестируйте работоспособность приложения,
  - Проведите вычислительные эксперименты,
  - Измерьте времена работы умножения матрицы на вектор при различных количествах исходных данных и различном числе процессов,
  - Определите получаемое ускорение,
  - Вычислите теоретическое время работы параллельного алгоритма,
  - Заполните таблицу результатов вычислений

Заключение

- Рассмотрен один из методов параллельного выполнения умножения матрицы на вектор
- Разработаны приложения, реализующие последовательный и параллельный алгоритмы умножения матрицы на вектор
- Проведены вычислительные эксперименты, проведено сравнение последовательного и параллельного алгоритмов

Темы заданий для самостоятельной работы

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

Литература

- **Dongarra, J.J., Duff, L.S., Sorensen, D.C., Vorst, H.A.V. (1999).** Numerical Linear Algebra for High Performance Computers (Software, Environments, Tools). Soc for Industrial & Applied Math/
- **Blackford,** L. S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J. J., Hammarling, S., Henry, G., Petitet, A., Stanley, D. Walker, R.C. Whaley, K. (1997). Scalapack Users' Guide (Software, Environments, Tools). Soc for Industrial & Applied Math.
- **Foster**, I. (1995). Designing and Building Parallel Programs: Concepts and Tools for Software Engineering. Reading, MA: Addison-Wesley.
- **Quinn**, M. J. (2004). Parallel Programming in C with MPI and OpenMP. – New York, NY: McGraw-Hill.

Следующая тема

- **Параллельные методы матричного умножения**

Авторский коллектив

Гергель В.П., профессор, д.т.н., руководитель

Гришагин В.А., доцент, к.ф.м.н.

Абросимова О.Н., ассистент (раздел 10)

Курылев А.Л., ассистент (лабораторные работы 4,5)

Лабутин Д.Ю., ассистент (система ПараЛаб)

Сысоев А.В., ассистент (раздел 1)

Гергель А.В., аспирант (раздел 12, лабораторная работа 6)

Лабутина А.А., аспирант (разделы 7,8,9; лабораторные работы 1,2,3; система ПараЛаб)

Сенин А.В. (раздел 11, лабораторные работы Microsoft Compute Cluster)

Ливерко С.В. (система ПараЛаб)

О проекте

Целью проекта является создание образовательного комплекса "Многопроцессорные вычислительные системы и параллельное программирование", обеспечивающий рассмотрение вопросов параллельных вычислений, предусматриваемых рекомендациями Computing Curricula 2001 Международных организаций IEEE-CS и ACM. Данный образовательный комплекс может быть использован для обучения на начальном этапе подготовки специалистов в области информатики, вычислительной техники и информационных технологий.

Образовательный комплекс включает **учебный курс "Введение в методы параллельного программирования" и лабораторный практикум "Методы и технологии разработки параллельных программ"**, что позволяет органично сочетать фундаментальное образование в области программирования и практическое обучение методам разработки масштабного программного обеспечения для решения сложных вычислительно-трудоемких задач на высокопроизводительных вычислительных системах.

Проект выполнялся в Нижегородском государственном университете им. Н.И. Лобачевского на кафедре математического обеспечения ЭВМ факультета вычислительной математики и кибернетики (http://www.software.unn.ac.ru). Выполнение проекта осуществлялось при поддержке компании Microsoft.