Лабораторная работа 5: Параллельные алгоритмы обработки графов

Лабораторная работа 5: Параллельные алгоритмы обработки графов ... 1
Цель лабораторной работы ... 1
Упражнение 1 – Постановка задачи поиска кратчайших путей ... 2
Упражнение 2 – Реализация последовательного алгоритма Флойда ... 3
Задание 1 – Открытие проекта SerialFloyd ... 3
Задание 2 – Ввод количества обрабатываемых вершин ... 4
Задание 3 – Завершение процесса вычислений ... 7
Задание 4 – Реализация алгоритма Флойда ... 8
Задание 5 – Проведение вычислительных экспериментов ... 9
Упражнение 3 – Разработка параллельного алгоритма Флойда ... 11
Определение подзадач ... 11
Выделение информационных зависимостей ... 11
Масштабирование и распределение подзадач по процессорам ... 12
Упражнение 4 – Реализация параллельного алгоритма Флойда ... 12
Задание 1 – Открытие проекта ParallelFloyd ... 12
Задание 2 – Инициализация и завершение параллельной программы ... 13
Задание 3 – Ввод исходных данных ... 15
Задание 4 – Завершение процесса вычислений ... 16
Задание 5 – Распределение данных между процессами ... 17
Задание 6 – Разработка алгоритма Флойда ... 18
Задание 7 – Выполнение итераций алгоритма Флойда ... 20
Задание 8 – Сбор результирующей матрицы ... 21
Задание 9 – Проверка правильности работы программы ... 21
Задание 10 – Реализация алгоритма Флойда для графов с произвольным количеством вершин 23
Задание 11 – Проведение вычислительных экспериментов ... 25
Контрольные вопросы ... 27
Задания для самостоятельной работы ... 27
Приложение 1. Программный код последовательного приложения, реализующего алгоритм Флойда ... 27
Файл SerialFloyd.cpp ... 27
Файл SerialFloydTest.cpp ... 29
Приложение 2. Программный код параллельного приложения, реализующего алгоритм Флойда 30
Файл ParallelFloyd.cpp ... 30
Файл ParallelFloydTest.cpp ... 35

Математические модели в виде *графов* широко используются при моделировании разнообразных явлений, процессов и систем. Как результат, многие теоретические и реальные прикладные задачи могут быть решены при помощи тех или иных процедур анализа графовых моделей. Среди множества этих процедур может быть выделен некоторый определенный набор *типовых алгоритмов обработки графов*. Рассмотрению вопросов теории графов, алгоритмов моделирования, анализу и решению задач на графах посвящено достаточно много различных изданий – см. раздел 11 "Параллельные алгоритмы обработки графов" учебных материалов курса.

Цель лабораторной работы

Целью данной лабораторной работы является разработка параллельной программы, решающей задачу поиска кратчайших путей используя алгоритм Флойда.

- Упражнение 1 – Постановка задачи поиска кратчайших путей.

- Упражнение 2 – Реализация последовательного алгоритма Флойда.
- Упражнение 3 – Разработка параллельного алгоритма Флойда.
- Упражнение 4 – Реализация параллельного алгоритма Флойда.

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 11 "Параллельные алгоритмы обработки графов" учебных материалов курса. Кроме того, предполагается, что выполнены как минимум ознакомительная лабораторная работа "Параллельное программирование с использованием MPI" и лабораторная работа 1 "Параллельные алгоритмы матрично-векторного умножения".

Упражнение 1 – Постановка задачи поиска кратчайших путей

При выполнении этого, начального, упражнения необходимо изучить последовательный алгоритм Флойда.

Пусть G есть граф

$G = (V, R)$,

для которого набор вершин v_i, $1 \le i \le n$, задается множеством V, а список дуг графа

$r_j = (v_{s_j}, v_{t_j})$, $1 \le j \le m$,

определяется множеством R. В общем случае дугам графа могут приписываться некоторые числовые характеристики (*веса*) w_j, $1 \le j \le m$ (*взвешенный граф*). Пример взвешенного графа приведен на рис. 5.1.

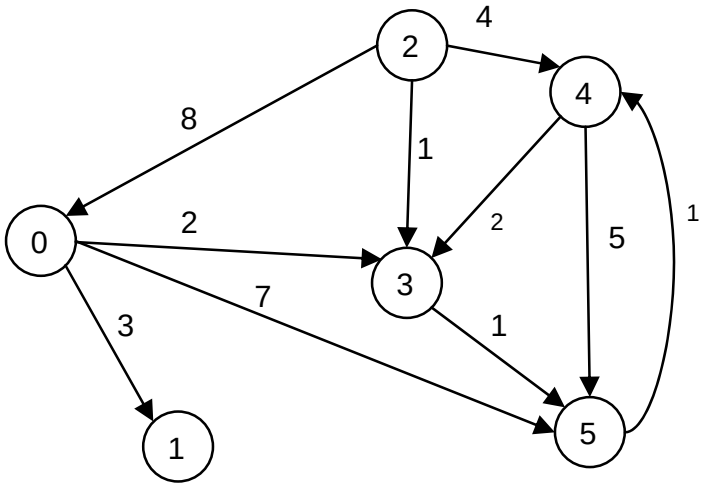

Рис. 5.1. Пример взвешенного ориентированного графа

Представление достаточно плотных графов, для которых почти все вершины соединены между собой дугами (т.е. $m \sim n^2$), может быть эффективно обеспечено при помощи *матрицы смежности*

$A = (a_{ij})$, $1 \le i$, $j \le n$,

ненулевые значения элементов которой соответствуют дугам графа

$$a_{ij} = \begin{cases} w(v_i, v_j), & \text{если } (v_i, v_j) \in R, \\ 0, & \text{если } i = j, \\ \infty, & \text{иначе.} \end{cases}$$

(для обозначения отсутствия ребра между вершинами в матрице смежности на соответствующей позиции используется знак бесконечности, при вычислениях знак бесконечности может быть заменен, например, на любое отрицательное число). Так, например, матрица смежности, соответствующая графу на рис. 5.1, приведена на рис. 5.2.

$$\begin{pmatrix} 0 & 3 & \infty & 2 & \infty & 7 \\ \infty & 0 & \infty & \infty & \infty & \infty \\ 8 & \infty & 0 & 1 & 4 & \infty \\ \infty & \infty & \infty & 0 & \infty & 1 \\ \infty & \infty & \infty & 2 & 0 & 5 \\ \infty & \infty & \infty & \infty & 1 & 0 \end{pmatrix}$$

Рис. 5.2. Матрица смежности для графа из рис. 5.1

Далее мы рассмотрим способы параллельной реализации алгоритмов на графах на примере *задачи поиска кратчайших путей* между всеми парами пунктов назначения. Задача состоит в том, что для имеющегося графа G требуется найти минимальные длины путей между каждой парой вершин графа. В качестве практического примера можно привести задачу составления маршрута движения транспорта между различными городами при заданном расстоянии между населенными пунктами и другие подобные задачи.

В качестве метода, решающего задачу поиска кратчайших путей между всеми парами пунктов назначения, далее используется *алгоритм Флойда* (*Floyd*) (см. раздел 11 "Параллельные алгоритмы обработки графов" учебных материалов курса).

Исходной информацией для *задачи поиска кратчайших путей* является взвешенный граф $G = (V, R)$, содержащий n вершин ($|V| = n$), в котором каждому ребру графа приписан неотрицательный вес. Граф будем полагать *ориентированным*, т.е., если из вершины i есть ребро в вершину j, то из этого не следует наличие ребра из j в i. В случае, если вершины все же соединены взаимообратными ребрами, то веса, приписанные им, могут не совпадать. Задача состоит в том, что для имеющегося графа G требуется найти минимальные длины путей между каждой парой вершин графа. В качестве практического примера можно привести задачу составления маршрута движения транспорта между различными городами при заданном расстоянии между населенными пунктами и другие подобные задачи.

Для поиска минимальных расстояний между всеми парами пунктов назначения Флойд предложил алгоритм, сложность которого имеет порядок n^3. В общем виде данный алгоритм может быть представлен следующим образом:

```
// Serial Floyd algorithm
for (k = 0; k < n; k++)
  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++)
      A[i,j] = min(A[i,j],A[i,k]+A[k,j]);
}
```

(реализация операции выбора минимального значения *min* должна учитывать способ указания в матрице смежности несуществующих дуг графа). Как можно заметить, в ходе выполнения алгоритма матрица смежности A изменяется, после завершения вычислений в матрице A будет храниться требуемый результат – длины минимальных путей для каждой пары вершин исходного графа.

Дополнительная информация по алгоритму Флойда может быть получена в разделе 11 "Параллельные алгоритмы обработки графов" учебных материалов курса.

Упражнение 2 – Реализация последовательного алгоритма Флойда

При выполнении этого упражнения необходимо реализовать последовательный алгоритм Флойда. Начальный вариант будущей программы представлен в проекте *SerialFloyd*, который содержит некоторую часть исходного кода. В ходе выполнения упражнения необходимо дополнить имеющийся вариант программы операциями ввода исходных данных, реализацией алгоритма Флойда и проверкой правильности результатов работы программы.

Задание 1 – Открытие проекта SerialFloyd

Откройте проект **SerialFloyd**, 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\ SerialFloyd**,

- Дважды щелкните на файле **SerialFloyd.sln** или, выбрав файл, выполните команду **Open**.

После открытия проекта в окне Solution Explorer (Ctrl+Alt+L) дважды щелкните на файле исходного кода **SerialFloyd.cpp**, как это показано на рис. 5.3. После этих действий программный код, который предстоит в дальнейшем расширить, будет открыт в рабочей области Visual Studio.

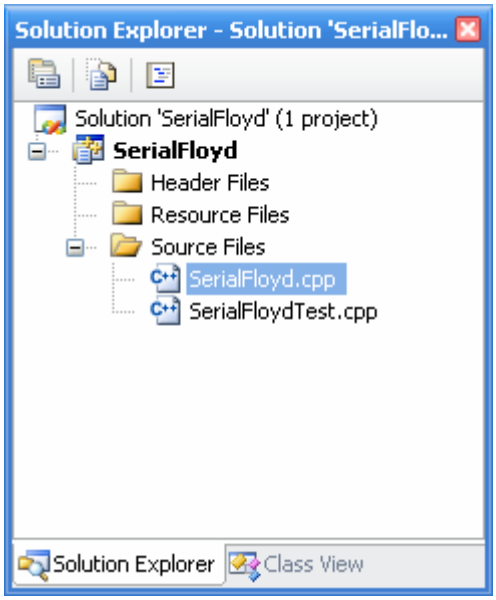

Рис. 5.3. Открытие файла SerialFloyd.cpp

В файле **SerialFloyd.cpp** подключаются необходимые библиотеки, а также содержится начальный вариант основной функции программы – функции *main*. Подготовленный вариант программы содержит объявление переменных и вывод на печать начального сообщения программы. В файле **SerialFloydTest.cpp** содержатся подготовленные варианты тестовых функций, которые понадобятся для проверки правильности разрабатываемых функций.

Рассмотрим переменные, которые используются в основной функции (*main*) нашего приложения. Переменная *pMatrix* есть указатель на матрицу смежности, после работы алгоритма Флойда эта переменная будет указывать на результат его работы. Переменная *Size* определяет размер матрицы смежности (или, что тоже самое, число вершин в графе).

```
  int *pMatrix;  // Adjacency matrix
  int  Size;     // Size of adjacency matrix
```

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*.

Вывод начального сообщения обеспечивается при помощи следующих действий программы:

```
  printf("Serial Floyd algorithm\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 Floyd algorithm`".

Задание 2 – Ввод количества обрабатываемых вершин

Для задания исходных данных последовательного алгоритма Флойда реализуем функцию *ProcessInitialization*. Эта функция предназначена для инициализации всех переменных, используемых в программе, в частности, для ввода количества обрабатываемых вершин (размера матрицы смежности), выделения памяти для матрицы смежности и для заполнения этой памяти начальными значениями. Таким образом, функция должна иметь следующий интерфейс:

```
// Function for allocating the memory and setting the initial values
```

```
void ProcessInitialization(double *&pMatrix, int& Size);
```

Добавим в программу код, позволяющий вводить размер матрицы смежности (задавать значение переменной *Size*) и проверять правильность произведенного ввода. Для этого добавьте на приведенный ниже код (выделенный полужирным шрифтом) в функцию *ProcessInitialization* и добавьте вызов этой функции в главную функцию (*main*) программы:

```
// Function for allocating the memory and setting the initial values
void ProcessInitialization(int *&pMatrix, int& Size) {
  do {
    printf("Enter the number of vertices: ");
    scanf("%d", &Size);
    if(Size <= 0)
      printf("The number of vertices should be greater than zero\n");
  } while(Size <= 0);

  printf("Using graph with %d vertices\n", Size);
}
```

Пользователю предоставляется возможность ввести количество вершин в матрице смежности, которое считывается из стандартного потока ввода *stdin* и сохраняется в целочисленной переменной *Size*. Далее, значение переменной *Size* проверяется (оно должно быть больше нуля), в случае ошибочного ввода последний повторяется, и, наконец, введенное значение выводится на печать (рис. 5.4).

Скомпилируйте и запустите приложение. Убедитесь в том, что в случае ввода отрицательного значения переменной *Size*, программа выдает диагностическое сообщение: "The number of vertices should be greater then zero".

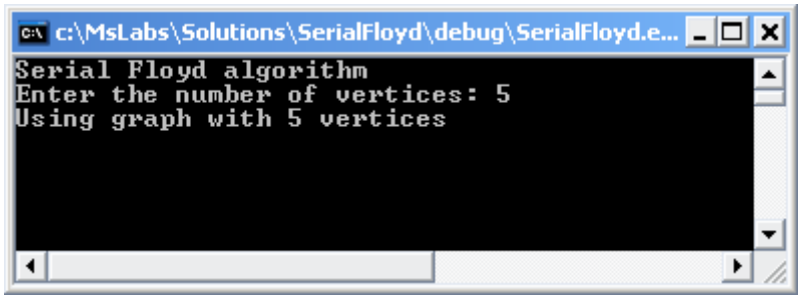

Рис. 5.4. Задание количества обрабатываемых вершин

Далее, необходимо дополнить функцию *ProcessInitialization* выделением памяти под матрицу смежности и проинициализировать эту матрицу. Для этого добавьте выделенный код в функцию *ProcessInitialization*:

```
// Function for allocating the memory and setting the initial values
void ProcessInitialization(int *&pMatrix, int& Size) {
  do {
    printf("Enter the number of vertices: ");

    scanf("%d", &Size);

    if(Size <= 0)
      printf("The number of vertices should be greater than zero\n");
  } while(Size <= 0);

  printf("Using graph with %d vertices\n", Size);

  // Allocate memory for the adjacency matrix
  pMatrix = new int[Size * Size];

  // Data initalization
  DummyDataInitialization(pMatrix, Size);
}
```

Непосредственно инициализацию матрицы смежности будет выполнять специальная функция. На первом этапе можно применить простой вариант реализации, который создает набор данных,

правильность обработки которого легко проверить. В тексте уже есть заготовка для такой функции (*DummyDataInitialization*), осталось дополнить её следующим кодом:

```
// Function for simple setting the initial data
void DummyDataInitialization(int *pMatrix, int Size) {
  for(int i = 0; i < Size; i++)
    for(int j = i; j < Size; j++) {
      if(i == j) pMatrix[i * Size + j] = 0;
      else
        if(i == 0) pMatrix[i * Size + j] = j;
        else       pMatrix[i * Size + j] = -1;

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

Как видно из представленного фрагмента кода, данная функция заполняет матрицу смежности достаточно простым образом: главная диагональ матрицы заполняется нулями, элементы, находящиеся в первой строке матрицы – номером столбца, остальные элементы, лежащие выше главной диагонали – минус единицами (обозначающими отсутствие ребра между вершинами). Элементы, лежащие ниже главной диагонали заполняются значениями симметричных относительно главной диагонали элементов. Таким образом, в случае, когда пользователь ввел размер матрицы смежности, равный 4, сама матрица будет определена следующим образом:

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 1 & 0 & -1 & -1 \\ 2 & -1 & 0 & -1 \\ 3 & -1 & -1 & 0 \end{pmatrix}$$

(задание матрицы при помощи датчика случайных чисел будет рассмотрено в задании 5).

Подобное задание элементов облегчает тестирование работы алгоритма, поскольку задает граф с простой структурой: нулевая вершина связывается двунаправленными ребрами со всеми остальными вершинами в графе, вес этого ребра равен номеру вершины, с которой соединяется ребро, исходящие из вершины с номером 0 (рис 5.5).

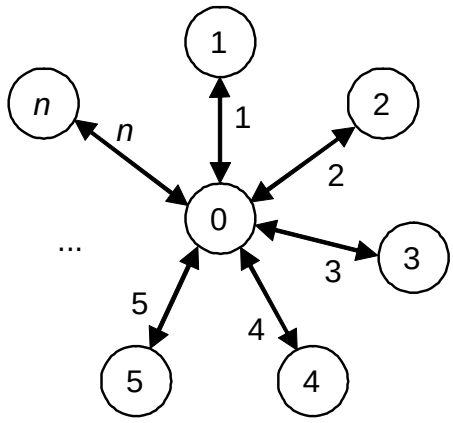

Рис. 5.5. Тестовый граф для алгоритма Флойда

Скомпилируйте приложение (выполните команду **Rebuild Solution** пункта меню **Build**). Если обнаружились ошибки, исправьте их, сверяя свой код с кодом, представленным в данном упражнении. После того, как все ошибки исправлены, запустите приложение.

Реализуем еще одну функцию, которая поможет в дальнейшем контролировать работу алгоритма. Это функция форматированного вывода матрицы смежности *PrintMatrix*. Заготовку этой функций можно найти в файле **SerialFloydTest.cpp**. Перейти к редактированию этого файла можно аналогично выбору для редактирования файла **SerialFloyd.cpp** в задании 1. 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*):

```
// Function for formatted matrix output
void PrintMatrix(int *pMatrix, int RowCount, int ColCount) {
  for(int i = 0; i < RowCount; i++) {
    for(int j = 0; j < ColCount; j++)
      printf("%7d", pMatrix[i * ColCount + j]);
    printf("\n");
  }
}
```

Добавим вызов функции *PrintMatrix* в функцию *main* приложения, сразу же после вызова функции *ProcessInitialization* для проверки правильности задания исходных данных:

```
...
  // Process initialization
  ProcessInitialization(pMatrix, Size);

  printf("The matrix before Floyd algorithm\n");
  PrintMatrix(pMatrix, Size, Size);
...
```

Скомпилируйте и запустите приложение. Убедитесь в том, что ввод данных происходит по описанным ранее правилам (рис. 5.6). Выполните несколько запусков приложения, задавайте различные размеры матрицы смежности.

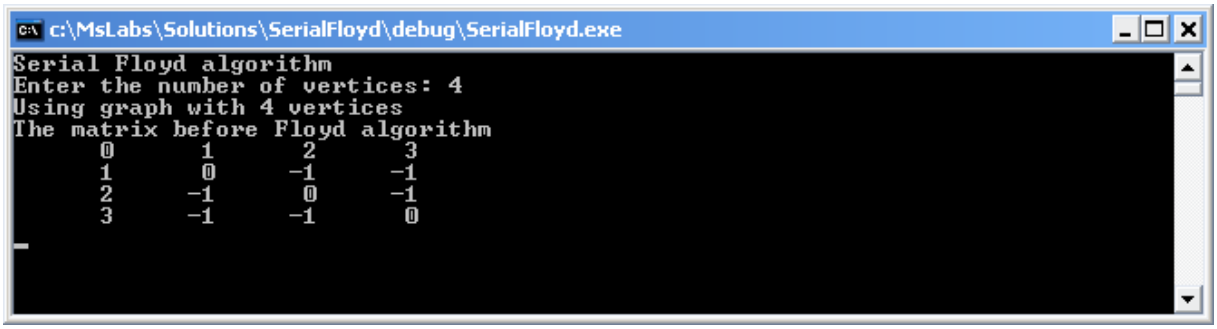

Рис. 5.6. Результат работы программы при завершении задания 2

Задание 3 – Завершение процесса вычислений

В данном задании перед выполнением алгоритма Флойда 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 матрицы смежности *pMatrix*. Следовательно, указатель на эту память необходимо передать в функцию *ProcessTermination* в качестве аргумента:

```
// Function for computational process termination
void ProcessTermination(int *pMatrix) {
  delete []pMatrix;
}
```

Вызов функции *ProcessTermination* необходимо добавить в программу непосредственно перед завершением основной функции *main*, когда матрица смежности уже больше не нужна:

```
...
  // Process termination
  ProcessTermination(pMatrix);

  return 0;
}
```

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

Задание 4 – Реализация алгоритма Флойда

Выполним теперь разработку основной вычислительной части программы. Для выполнения алгоритма Флойда реализуем функцию *SerialFloyd*, которая принимает на вход матрицу смежности *pMatrix* и размер этой матрицы *Size*.

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

```
// Serial Floyd algorithm
void SerialFloyd(int *pMatrix, int Size) {
  int t1, t2;
  for(int k = 0; k < Size; k++)
    for(int i = 0; i < Size; i++)
      for(int j = 0; j < Size; j++)
        if((pMatrix[i * Size + k] != -1) &&
           (pMatrix[k * Size + j] != -1)) {
          t1 = pMatrix[i * Size + j];
          t2 = pMatrix[i * Size + k] + pMatrix[k * Size + j];
          pMatrix[i * Size + j] = Min(t1, t2);
      }
}
```

Как уже отмечалось, функция выбора минимального значения должна быть согласована с выбранным способом обозначения бесконечности, в данной лабораторной работе в качестве бесконечно большого значения выбрано значение -1. Реализуем такую функцию, назовем ее *Min*:

```
int Min(int A, int B) {
  int Result = (A < B) ? A : B;

  if((A < 0) && (B >= 0)) Result = B;
  if((B < 0) && (A >= 0)) Result = A;
  if((A < 0) && (B < 0))  Result = -1;

  return Result;
}
```

Выполним вызов функции, реализующей алгоритм Флойда, из основной программы. Для контроля правильности выполнения алгоритма распечатаем получившуюся матрицу:

```
...
  // Process initialization
  ProcessInitialization(pMatrix, Size);

  printf("The matrix before Floyd algorithm\n");
  PrintMatrix(pMatrix, Size, Size);

  // Serial Floyd algorithm
  SerialFloyd(pMatrix, Size);

  printf("The matrix after Floyd algorithm\n");
  PrintMatrix(pMatrix, Size, Size);
...
```

Скомпилируйте и запустите приложение. Проанализируйте результат работы алгоритма Флойда. Если алгоритм реализован правильно, то результирующая матрица должна иметь следующую структуру: все элементы, лежащие на главной диагонали равны нулю, а остальные элементы равны сумме номера строки и столбца этого элемента, считая строки и столбцы занумерованными с нуля. Так, для рассматривавшейся в задании 2 матрицы результат должен быть следующим:

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 1 & 0 & 3 & 4 \\ 2 & 3 & 0 & 5 \\ 3 & 4 & 5 & 0 \end{pmatrix}$$

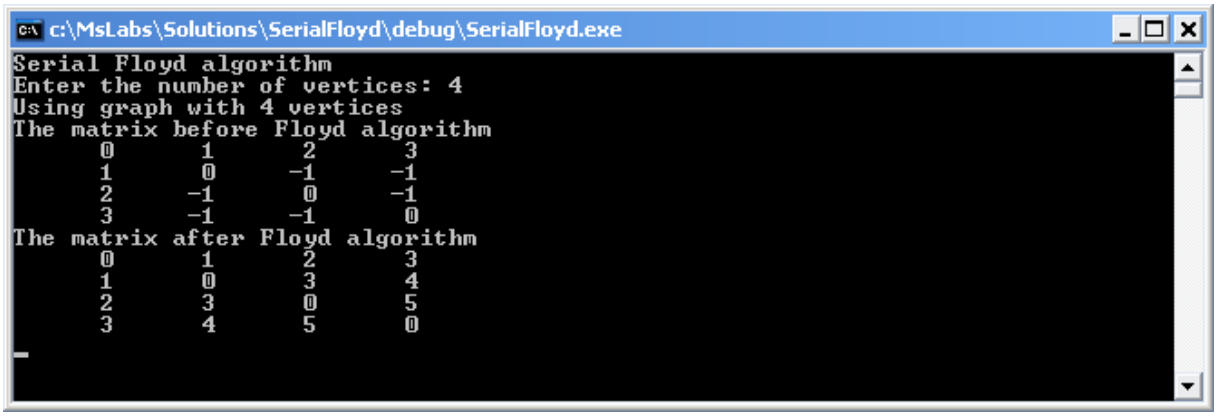

Рис. 5.7. Результат работы алгоритма Флойда для тестовой матрицы из 4-х элементов

Задание 5 – Проведение вычислительных экспериментов

Для последующего тестирования ускорения работы параллельного алгоритма необходимо провести эксперименты по вычислению времени выполнения последовательного алгоритма. Анализ времени выполнения алгоритма целесообразно проводить для достаточно больших размеров данных. Задавать эти данные будем при помощи датчика случайных чисел. Для этого реализуем еще одну функцию задания элементов *RandomDataInitialization* (датчик случайных чисел инициализируется текущим значением времени):

```
// Function for initializing the data by the random generator
void RandomDataInitialization(int *pMatrix, int Size) {
  srand( (unsigned)time(0) );

  for(int i = 0; i < Size; i++)
    for(int j = 0; j < Size; j++)
      if(i != j) {
        if((rand() % 100) < InfinitiesPercent)
          pMatrix[i * Size + j] = -1;
        else
          pMatrix[i * Size + j] = rand() + 1;
      }
      else
        pMatrix[i * Size + j] = 0;
}
```

Будем вызывать эту функцию вместо ранее разработанной функции *DummyDataInitialization* в функции *ProcessInitialization*, которая генерировала предсказуемые данные, при использовании которых было легко проверить правильность работы алгоритма:

```
// Function for allocating the memory and setting the initial values
void ProcessInitialization(int *&pMatrix, int& Size) {
...
  // Allocate memory for the adjacency matrix
  pMatrix = new int[Size * Size];

  // Data initalization
  //DummyDataInitialization(pMatrix, Size);
  RandomDataInitialization(pMatrix, 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. Как и в других лабораторных работах, воспользуемся для этого функцией:

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

Для вычисления времени работы нам понадобятся три дополнительные переменные, объявления которых нужно добавить в функцию *main*:

```
int main(int argc, char *argv[]) {
  int *pMatrix = 0;    // Adjacency matrix
  int  Size    = 0;    // Size of adjacency matrix

  time_t start, finish;
  double duration = 0.0;
...
```

Добавим в программный код вычисление и вывод времени непосредственного выполнения алгоритма Флойда, для этого поставим замеры времени до и после вызова функции *SerialFloyd*:

```
...
  // Process initialization
  ProcessInitialization(pMatrix, Size);

  printf("The matrix before Floyd algorithm\n");
  PrintMatrix(pMatrix, Size, Size);

  start = clock();
  // Parallel Floyd algorithm
  SerialFloyd(pMatrix, Size);
  finish = clock();

  printf("The matrix after Floyd algorithm\n");
  PrintMatrix(pMatrix, Size, Size);

  duration = (finish - start) / double(CLOCKS_PER_SEC);

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

Скомпилируйте и запустите приложение. Для проведения вычислительных экспериментов с большими объектами, отключите печать данных до и после алгоритма Флойда (закомментируйте соответствующие строки кода). Проведите вычислительные эксперименты и заполните третий столбец таблицы:

Таблица 5.1. Результаты вычислительных экспериментов для алгоритма Флойда

Номер теста	Количество вершин в графе	Время работы алгоритма (сек.)
1	10	
2	500	
3	600	
4	700	
5	800	
6	900	
7	1000	

По результатам выполненных экспериментов оцените характер зависимости времени работы алгоритма Флойда от количества вершин в графе – убедитесь, данная зависимость является кубической (при увеличении количества данных в два раза время работы алгоритма увеличивается в восемь раз и т.д.).

Оценим аналитически трудоемкость алгоритма Флойда (см. раздел 11 "Параллельные алгоритмы обработки графов" учебных материалов курса). Работа алгоритма предполагает повторение $Size^3$ однотипных операций:

$$T_1 = Size^3 \cdot \tau, \tag{5.1}$$

где τ есть время выполнения одной операции сравнения элементов матрицы смежности, проводимой алгоритмом.

Заполним таблицу сравнения реального времени выполнения со временем, которое может быть получено по формуле (5.1). Для вычисления времени выполнения одной операции применим следующую методику: выберем один из экспериментов как образец (например, эксперимент по обработке графа содержащего 800 вершин) и поделим время выполнения этого эксперимента на число выполненных

операций. Таким образом, вычислим время выполнения одной операции τ. Далее, используя это значение, вычислим теоретическое время выполнения для всех оставшихся экспериментов.

Проведите указанные вычисления, результаты занесите в таблицу:

Таблица 5.2. Сравнение экспериментального и теоретического времени работы алгоритма Флойда

Время выполнения одной операции τ (сек):			
Номер теста	Количество вершин в графе	Время работы (сек)	Теоретическое время (сек)
1	10		
2	500		
3	600		
4	700		
5	800		
6	900		
7	1000		

Заметим, что время выполнения одной операции сравнения элементов, вообще говоря, зависит от количества обрабатываемых вершин. Такая зависимость объясняется особенностями архитектуры компьютера. Если данных немного, то они полностью могут быть помещены в кэш-память процессора, доступ к этой памяти осуществляется очень быстро. Если алгоритм со средним количеством данных, 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 элементу оперативной памяти требуется больше времени, чем для обращения к кэш. Если же данных настолько много, что они не могут быть помещены в оперативную память, то включается механизм работы с файлами подкачки (swap file), данные 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 обработке графа с 800 вершинами.

Упражнение 3 – Разработка параллельного алгоритма Флойда

При выполнении этого упражнения необходимо изучить принципы распараллеливания алгоритма Флойда. Для этого необходимо провести декомпозицию задачи, выделить информационные взаимодействия между подзадачами, определить вычислительную схему и выполнить распределение набора подзадач по вычислительным устройствам.

Определение подзадач

Как следует из общей схемы алгоритма Флойда, основная вычислительная нагрузка при решении задачи поиска кратчайших путей состоит в выполнении операции выбора минимальных значения. Данная операция является достаточно простой и ее распараллеливание не приведет к заметному ускорению вычислений. Более эффективный способ организации параллельных вычислений может состоять в одновременном выполнении нескольких операций обновления значений матрицы A – корректность такого подхода к распараллеливанию показана в разделе 11 "Параллельные алгоритмы обработки графов" учебных материалов курса.

Как результат, необходимые условия для организации параллельных вычислений обеспечены и, тем самым, в качестве *базовой подзадачи* может быть использована операция обновления элементов матрицы A (для указания подзадач будем использовать индексы обновляемых в подзадачах элементов).

Выделение информационных зависимостей

Выполнение вычислений в подзадачах становится возможным только тогда, когда каждая подзадача (i,j) содержит необходимые для расчетов элементы A_{ij}, A_{ik}, A_{kj} матрицы A. Для исключения дублирования данных разместим в подзадаче (i,j) единственный элемент A_{ij}, тогда получение всех остальных необходимых значений может быть обеспечено только при помощи передачи данных. Таким образом, каждый элемент A_{kj} строки k матрицы A должен быть передан всем подзадачам (k,j), $1 \le j \le n$,

а каждый элемент A_{ik} столбца k матрицы A должен быть передан всем подзадачам (i,k), $1 \le i \le n$, - см. рис. 5.8.

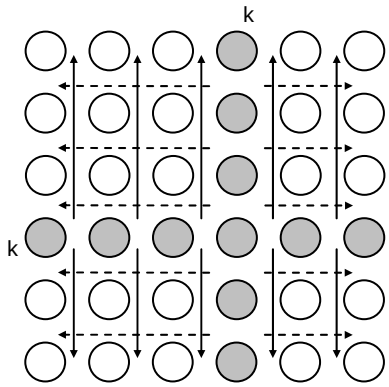

Рис. 5.8. Информационная зависимость базовых подзадач (стрелками показаны направления обмена значениями на итерации k)

Масштабирование и распределение подзадач по процессорам

Как правило, число доступных процессоров p существенно меньше, чем число базовых задач n^2 ($p << n^2$). Возможный способ укрупнения вычислений состоит в использовании *ленточной схемы* разбиения матрицы A – такой подход соответствует объединению в рамках одной базовой подзадачи вычислений, связанных с обновлением элементов одной или нескольких строк (*горизонтальное* разбиение) или столбцов (*вертикальное* разбиение) матрицы A. Эти два типа разбиения практически равноправны – учитывая дополнительный момент, что для алгоритмического языка С массивы располагаются по строкам, будем рассматривать далее только разбиение матрицы A на горизонтальные полосы.

Следует отметить, при таком способе разбиения данных на каждой итерации алгоритма Флойда потребуется передавать между подзадачами только элементы одной из строк матрицы A. Для эффективного выполнения подобной коммуникационной операции топология сети должна представлять собой гиперкуб или полный граф.

Упражнение 4 – Реализация параллельного алгоритма Флойда

При выполнении этого упражнения необходимо разработать параллельный алгоритм Флойда. Выполнение упражнения позволит:

- Расширить практические знания по разработке параллельных программ,
- Разработать параллельную программу, реализующую алгоритм Флойда.

Как и ранее, разрабатываемая параллельная программа будет состоять из следующих основных структурных частей:

- 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.

При выполнении упражнения предполагается знание раздела 4 "Параллельное программирование на основе MPI".

Задание 1 – Открытие проекта ParallelFloyd

Откройте проект **ParallelFloyd**, 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\ ParallelFloyd**,
- Дважды щелкните на файле **ParallelFloyd.sln** или подсветите его и выполните команду **Open**.

После открытия проекта в окне Solution Explorer (Ctrl+Alt+L) дважды щелкните на файле исходного кода **ParallelFloyd.cpp**, как это показано на рис. 5.9. После этих действий код, который предстоит модифицировать, будет открыт в рабочей области Visual Studio.

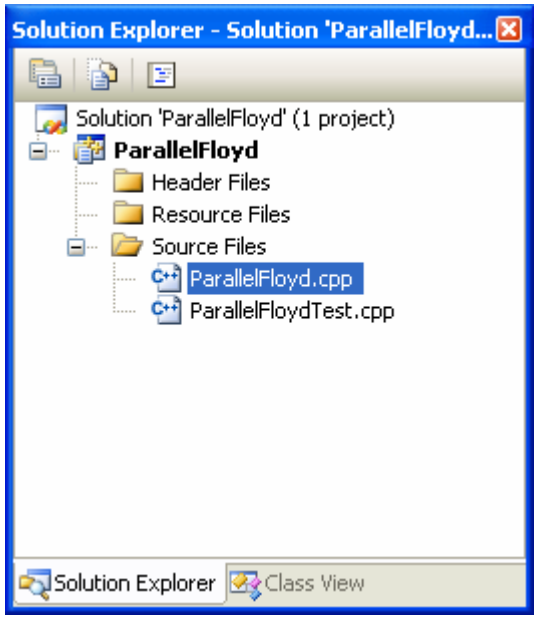

Рис. 5.9. Открытие файла ParallelFloyd.cpp

В файле **ParallelFloyd.cpp** расположена главная функция (*main*) будущего параллельного приложения, которая содержит объявления необходимых переменных. Также в файле **ParallelFloyd.cpp** расположены функции, перенесенные сюда из проекта, содержащего последовательный алгоритм Флойда: *DummyDataInitialization*, *RandomDataInitialization* и *Min*. Кроме того, в файле **ParallelFloydTest.cpp** содержится функция *PrintMatrix*, отвечающая за тестовый вывод данных (подробно о назначении этой функции рассказывается в упражнении 2 данной лабораторной работы). Эти функции можно будет использовать и в параллельной программе. Кроме того, в исходный код 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 Floyd algorithm`".

Задание 2 – Инициализация и завершение параллельной программы

Как уже отмечалось ранее, в параллельной программе должен присутствовать заголовочный файл "mpi.h".

```
#include <cstdlib>
#include <cstdio>
#include <cstring>
#include <ctime>
#include <cmath>
#include <algorithm>
#include <mpi.h>
```

Далее, в главной функции программы необходимо проинициализировать среду выполнения MPI-программы, определить число процессов, доступных для MPI-программы, определить ранг процесса в рамках коммуникатора MPI_COMM_WORLD, а также завести глобальные переменные для хранения этих значений (*ProcNum* и *ProcRank* соответственно). Добавьте выделенный код:

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

void main(int argc, char* argv[]) {
  int *pMatrix;    // Adjacency matrix
  int  Size;       // Size of adjacency matrix
  int *pProcRows;  // Process rows
  int  RowNum;    // Number of process rows
```

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

  if(ProcRank == 0)
    printf("Parallel Floyd algorithm \n");

  MPI_Finalize();
}
```

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. 5.10).

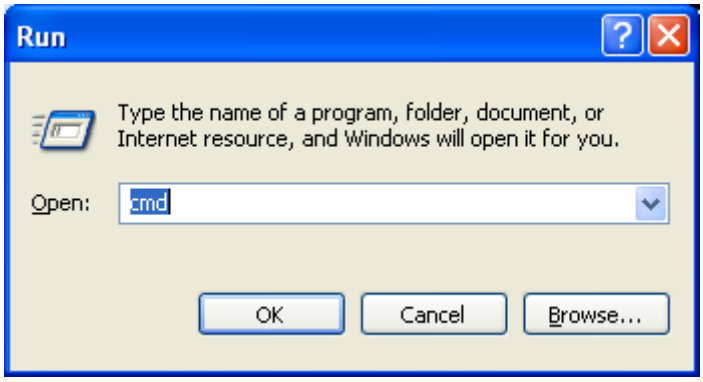

Рис. 5.10. Запуск Command Prompt

В командной строке перейдите в папку, где содержится исполняемый модуль разработанной программы (рис. 5.11):

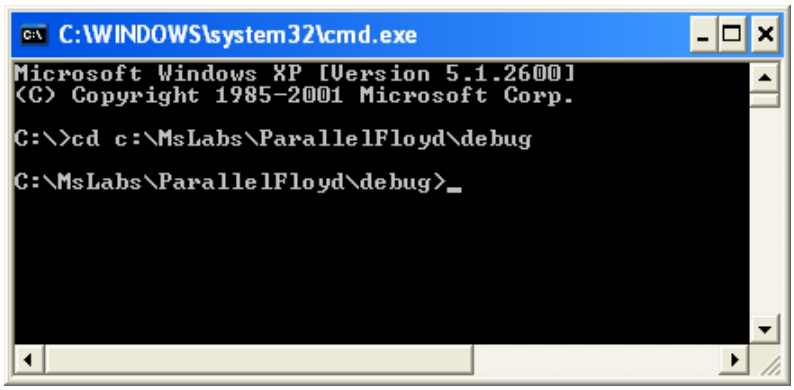

Рис. 5.11. Задание папки, в которой содержится исполняемый модуль параллельной программы

Аналогично запуску программ из предыдущих лабораторных работ, для запуска программы из 4 процессов наберите команду (рис. 5.12):

```
mpiexec -n 3 ParallelFloyd.exe
```

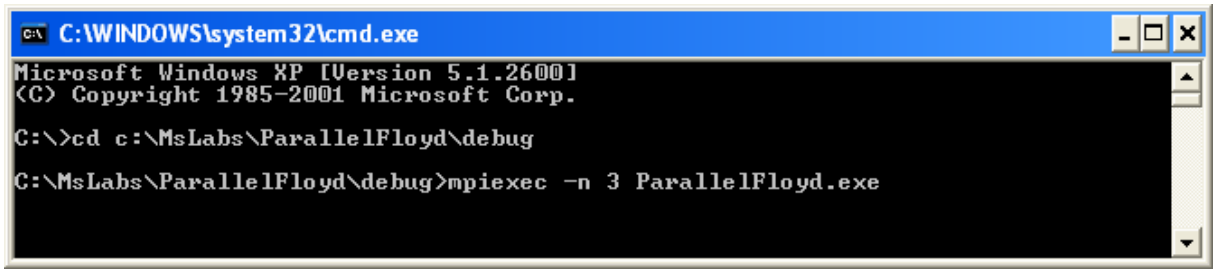

Рис. 5.12. Запуск параллельной программы

Убедитесь в том, что в командную консоль выводится приветствие: "`Parallel Floyd algorithm`".

Задание 3 – Ввод исходных данных

Перейдем к организации ввода данных. Необходимо дополнить нашу программу кодом, обеспечивающим задание количества вершин обрабатываемого графа и выделяющим память под матрицу смежности. Как и в других работах, определение начальных данных будет осуществляться одним из процессов (пусть этим процессом будет процесс с рангом 0). Далее, согласно схеме параллельных вычислений, изложенной в упражнении 3, матрица смежности распределяется между всеми процессами таким образом, что каждый процесс обрабатывает непрерывную последовательность (полосу) данных. Отметим, что первая версия разрабатываемой программы ориентирована на случай, когда число вершин в графе делится нацело на число процессов, то есть полосы на всех процессах содержат одно и то же число строк матрицы. Это количество будем хранить в переменной *RowNum*. Адрес буфера памяти, где содержится полоса каждого из процессов, будем хранить в переменной *pProcRows*. В результате работы алгоритма Флойда, каждый процесс получает *RowNum* строк результирующей матрицы. Затем эти данные необходимо будет вновь собрать на *ведущем* процессе (процессе с рангом 0).

Для инициализации вычислительных процессов, как и ранее, служит функция *ProcessInitialization*:

```
// Function for allocating the memory and setting the initial values
void  ProcessInitialization(int  *&pMatrix,  int  *&pProcRows,  int&  Size,
  int& RowNum);
```

Итак, сначала нужно определить количество вершин в графе, то есть задать значение переменной *Size*. Для определения этого количества необходимо реализовать диалог с пользователем. Как и в предыдущих лабораторных работах, реализуем проверку правильности вводимого значения. Добавьте выделенный фрагмент кода в тело функции *ProcessInitialization*:

```
// Function for allocating the memory and setting the initial values
void  ProcessInitialization(int  *&pMatrix,  int  *&pProcRows,  int&  Size,
  int& RowNum) {
  setvbuf(stdout, 0, _IONBF, 0);
  if(ProcRank == 0) {
    do {
      printf("Enter the number of vertices: ");
      scanf("%d", &Size);
      if(Size < ProcNum)
        printf("The number of vertices should be greater then "
          "number of processes\n");
      if(Size % ProcNum != 0)
        printf("The number of vertices should be divisible by "
          "number of processes\n");
    } while((Size < ProcNum) || (Size % ProcNum != 0));

    printf("Using graph with %d vertices\n", Size);
  }
}
```

Теперь необходимо разослать количество вершин остальным процессам. Для этого используем знакомую по предыдущим лабораторным работам функцию широковещательной рассылки *MPI_Bcast*. Добавьте следующий код в вашу программу (обратите внимание, что вызов *MPI_Bcast* должен выполняться всеми процессами):

```
// Function for allocating the memory and setting the initial values
void ProcessInitialization(int *&pMatrix, int *&pProcRows, int& Size,
  int& RowNum) {
  setvbuf(stdout, 0, _IONBF, 0);
  if(ProcRank == 0) {
    ...
    printf("Using graph with %d vertices\n", Size);
  }

  // Broadcast the number of vertices
  MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);

  // Number of rows for each process
  RowNum = Size / ProcNum;
}
```

Добавьте вызов функции инициализации вычислительных процессов:

```
int main(int argc, char* argv[]) {
  int *pMatrix;    // Adjacency matrix
  int  Size;       // Size of adjacency matrix
  int *pProcRows;  // Process rows
  int  RowNum ;    // Number of process rows

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

  if(ProcRank == 0)
    printf("Parallel Floyd algorithm program\n");

  // Process initialization
  ProcessInitialization(pMatrix, pProcRows, Size, RowNum);

  MPI_Finalize();

  return 0;
}
```

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 разное количество вершин в обрабатываемом графе.

После того, как количество вершин в графе определено, можно перейти к выделению памяти под матрицу смежности и полосы, принадлежащие процессам. Добавьте выделенный код в тело функции *ProcessInitialization*:

```
  // Broadcast the number of vertices
  MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);

  // Number of rows for each process
  RowNum = Size / ProcNum;

  // Allocate memory for the current process rows
  pProcRows = new int[Size * RowNum];

  if(ProcRank == 0) {
    // Allocate memory for the adjacency matrix
    pMatrix = new int[Size * Size];

    // Data initalization
    DummyDataInitialization(pMatrix, Size);
  }
}
```

Отметим, что для матрицы смежности на ведущем процессе была использована функция генерации *DummyDataInitialization*, которая была разработана при реализации последовательного приложения. Напомним, что эта функция заполняет массив предсказуемыми значениями, при которых легко проверить работу алгоритма.

Задание 4 – Завершение процесса вычислений

Для корректного завершения параллельной программы разработаем функцию остановки процесса вычислений. Для этого необходимо освободить память, выделенную динамически в процессе выполнения программы. Реализуем соответствующую функцию *ProcessTermination*.

На ведущем процессе выделялась память для матрицы смежности *pMatrix*, кроме того, на всех процессах выделялась память под полосы *pProcRows*. Оба этих указателя необходимо передать в функцию *ProcessTermination* в качестве аргументов:

```
// Function for computational process termination
void ProcessTermination(int *pMatrix, int *pProcRows) {
  if(ProcRank == 0)
```

```
    delete []pMatrix;
  delete []pProcRows;
}
```

Вызов функции остановки процесса вычислений необходимо добавить в функцию *main* непосредственно перед вызовом функции *MPI_Finalize*:

```
...
  // Process termination
  ProcessTermination(pMatrix, pProcRows);

  MPI_Finalize();
}
```

Добавьте в код основной функции команды для печати матрицы смежности на ведущем процессе (используйте функцию *PrinMatrix*, реализованную в ходе разработки последовательного приложения). Скомпилируйте и запустите приложение. Убедитесь в том, что исходные данные задаются верно.

Задание 5 – Распределение данных между процессами

В соответствии со схемой параллельных вычислений, изложенной в предыдущем упражнении, матрица смежности должна быть распределена между процессами равными полосами.

За распределение данных отвечает функция *DataDistribution*. На вход этой функции в качестве аргументов необходимо передать указатель на матрицу смежности *pMatrix*, количество вершин в графе (размер матрицы) и адрес буфера для хранения части данных, принадлежащих процессу *pProcRows*, а также размер полосы *RowNum*:

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

Далее необходимо, используя обобщенную операцию передачи данных от одного процесса всем процессам (распределение данных), распределить матрицу смежности между процессами:

```
// Data distribution among the processes
void DataDistribution(int *pMatrix, int *pProcRows, int Size, int RowNum) {
  MPI_Scatter(pMatrix, RowNum * Size, MPI_INT, pProcRows, RowNum * Size,
    MPI_INT, 0, MPI_COMM_WORLD);
}
```

Соответственно, вызывать эту функцию из основной программы следует после вызова функции инициализации вычислительного процесса *ProcessInitialization*:

```
  // Process initialization
  ProcessInitialization(pMatrix, pProcRows, Size, RowNum);

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

Теперь выполним проверку правильности разделения данных между процессами. Для этого после выполнения функции *DataDistribution* распечатаем исходную матрицу смежности, а затем полосы, 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 этого необходимо, кроме ранее разработанной функции *PrintMatrix*, реализовать еще одну функцию, производящую отладочную печать. Эта функция (назовем ее *ParallelPrintMatrix*) будет осуществлять последовательный вывод процессами своих полос матрицы, используя функцию *PrintMatrix*. Добавьте следующий код в разрабатываемое приложение:

```
// Function for formatted output of all stripes
void ParallelPrintMatrix(int *pProcRows, int Size, int RowNum) {
  for(int i = 0; i < ProcNum; i++) {
    if (ProcRank == i) {
      printf("ProcRank = %d\n", ProcRank);
      printf("Proc rows:\n");
      PrintMatrix(pProcRows, RowNum, Size);
    }
    MPI_Barrier(MPI_COMM_WORLD);
  }
}
```

Наконец, используя все ранее разработанные функции отладочной печати, функцию *TestDistribution* можно реализовать следующим образом:

```
// Function for testing the data distribution
void TestDistribution(int *pMatrix, int *pProcRows, int Size, int RowNum) {
  MPI_Barrier(MPI_COMM_WORLD);
  if (ProcRank == 0) {
    printf("Initial adjacency matrix:\n");
    PrintMatrix(pMatrix, Size, Size);
  }

  MPI_Barrier(MPI_COMM_WORLD);

  ParallelPrintMatrix(pProcRows, Size, RowNum);
}
```

Функция *TestDistribution* похожа на разработанные в других лабораторных работах функции аналогичного назначения: ведущий процесс распечатывает все данные, затем процессы параллельной программы печатают свои данные по порядку (сначала свои данные 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 data between processes
  DataDistribution(pMatrix, pProcRows, Size, RowNum);

  // Testing the distribution
  TestDistribution(pMatrix, pProcRows, Size, RowNum);
...
```

Скомпилируйте приложение. Если в приложении обнаружились ошибки, исправьте их, сверяя свой код с кодом, представленным в данной работе. Запустите приложение из трех процессов и установите количество вершин в обрабатываемом графе равным 6. Убедитесь в том, что распределение данных выполняется правильно (рис. 5.13).

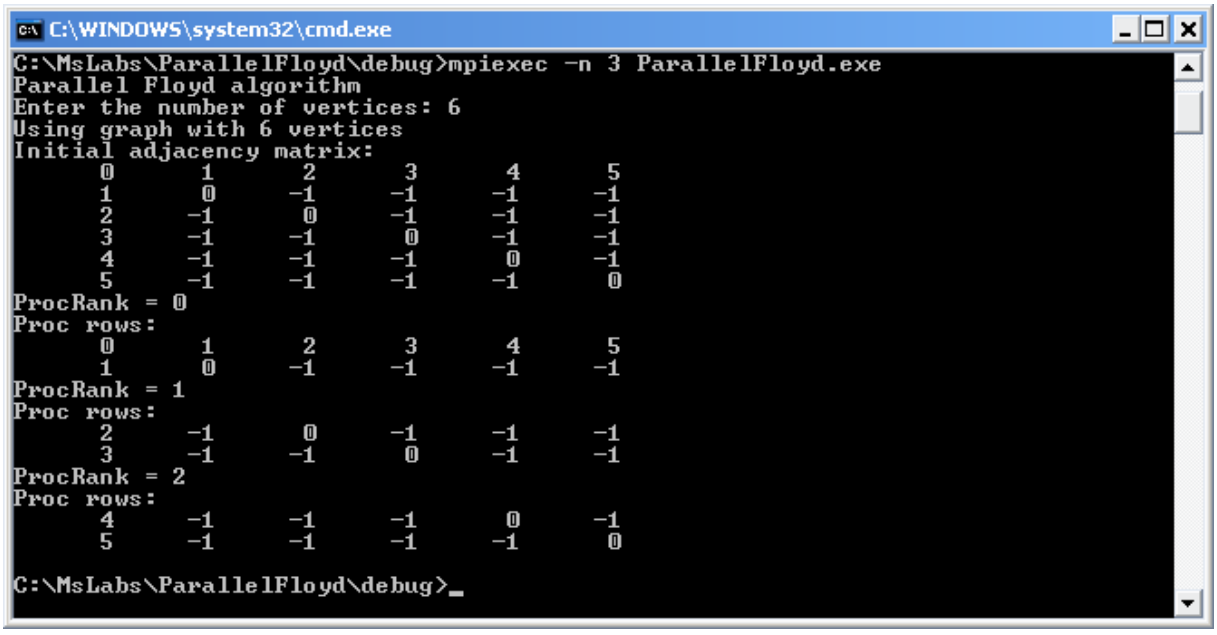

Рис. 5.13. Распределение данных для программы из трех процессов.

Задание 6 – Разработка алгоритма Флойда

Выполним реализацию алгоритма Флойда в ходе нескольких последовательных этапов, каждый из которых будет достаточно простым и правильность каждого из которых можно будет легко проверить.

Определим заголовок функции, реализующей рассматриваемый алгоритм. Для работы алгоритма необходимо иметь указатель на участок памяти *pProcRows*, где расположены строки матрицы смежности, принадлежащие этому процессу, размер этой матрицы *Size* и количество строк, принадлежащих процессу *RowNum*. Как результат, разрабатываемая функция будет иметь следующий заголовок:

```
// Parallel Floyd algorithm
void ParallelFloyd(int *pProcRows, int Size, int RowNum);
```

В соответствии с общей схемой параллельного алгоритма Флойда, необходимо *Size* раз выполнить однотипную операцию, обновляющую матрицу смежности. Для выполнения этой операции всем процессам необходима строка матрицы, номер которой совпадает с номером итерации. Эта строка необязательно должна храниться у текущего процесса, а значит, при выполнении операции необходимо провести обмен данными между процессами. Кроме того, для хранения этой строки необходимо выделить память. Следовательно, программа для параллельного алгоритма Флойда на первом этапе разработки будет выглядеть следующим образом:

```
// Parallel Floyd algorithm
void ParallelFloyd(int *pProcRows, int Size, int RowNum) {
  int *pRow = new int[Size];

  for(int k = 0; k < Size; k++) {
    // Distribute row among all processes
    RowDistribution(pProcRows, Size, RowNum, k, pRow);
  }
  delete []pRow;
}
```

Используемая выше функция *RowDistribution* должна по номеру строки *k* находить процесс, которому принадлежит *k*-я строка матрицы смежности и разослать эту строку всем остальным процессам. Как и ранее, воспользуемся операцией *MPI_Bcast* для такого рода обмена:

```
// Function for row broadcasting among all processes
void RowDistribution(int *pProcRows, int Size, int RowNum, int k,
  int *pRow) {
  int ProcRowRank = k / RowNum;  // Process rank with the row k
  int ProcRowNum  = k - ProcRowRank * RowNum; // Process row number

  if(ProcRowRank == ProcRank)
    // Copy the row to pRow array
    copy(&pProcRows[ProcRowNum*Size],&pProcRows[(ProcRowNum+1)*Size],pRow);

  // Broadcast row to all processes
  MPI_Bcast(pRow, Size, MPI_INT, ProcRowRank, MPI_COMM_WORLD);
}
```

Проверим разработанную часть алгоритма Флойда. Для этого организуем отладочную печать, выводящую рассылаемую строку процессом с рангом 0. Для этого добавьте следующий код в тело функции *ParallelFloyd*:

```
...
  for(int k = 0; k < Size; k++) {
    // Distribute row among all processes
    RowDistribution(pProcRows, Size, RowNum, k, pRow);

    if(ProcRank == 0) {
      printf("Row %d after distribution:", k);
      PrintMatrix(pRow, Size, 1);
    }
  }
...
```

В результате выполнения этого кода, на консоли должны последовательно появиться строки матрицы смежности (структура тестовой матрицы смежности была описана в задании 2 упражнения 2).

Закомментируйте вызов функции *TestDistribution* в функции *main*, скомпилируйте приложение. Запустите приложение и удостоверьтесь, что распределение строки данных производится верно. Для тестовой матрицы смежности размером 6 на 6 вывод программы приведен на рисунке 5.14.

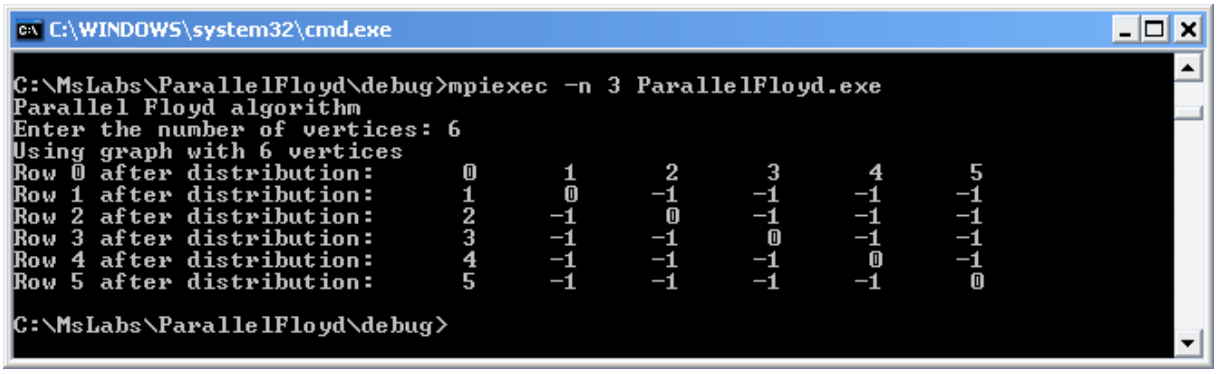

Рис. 5.14. Отладочная печать для проверки работы функции *RowDistribution*

Задание 7 – Выполнение итераций алгоритма Флойда

В соответствии с общей схемой параллельного алгоритма Флойда, после рассылки очередной строки матрицы смежности между процессами необходимо выполнить обновление матрицы смежности. Приведите функцию *ParallelFloyd* к следующему виду:

```
// Parallel Floyd algorithm
void ParallelFloyd(int *pProcRows, int Size, int RowNum) {
  int *pRow = new int[Size];
  int t1, t2;

  for(int k = 0; k < Size; k++) {
    // Distribute row among all processes
    RowDistribution(pProcRows, Size, RowNum, k, pRow);

    // Update adjacency matrix elements
    for(int i = 0; i < RowNum; i++)
      for(int j = 0; j < Size; j++)
        if( (pProcRows[i * Size + k] != -1) &&
            (pRow     [j]            != -1)) {
          t1 = pProcRows[i * Size + j];
          t2 = pProcRows[i * Size + k] + pRow[j];
          pProcRows[i * Size + j] = Min(t1, t2);
      }
  }
  delete []pRow;
}
```

Этот этап, как и все предыдущие, необходимо проверить. Снова используем отладочную печать с помощью функции *ParallelPrintMatrix*. Закомментируйте все предыдущие вызовы этой функции и добавьте новый вызов этой функции непосредственно после вызова алгоритма Флойда *ParallelFloyd*:

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

  // Parallel Floyd algorithm
  ParallelFloyd(pProcRows, Size, RowNum);
  ParallelPrintMatrix(pProcRows, Size, RowNum);
```

Для данных, задаваемых при помощи функции *DummyDataInitialization*, результат работы алгоритма Флойда заранее известен и рассмотрен в задании 4 упражнения 2. Так, например, при размере тестовой матрицы смежности 6 на 6 и при запуске приложения из трех процессов, результат должен быть следующим (рис. 5.15).

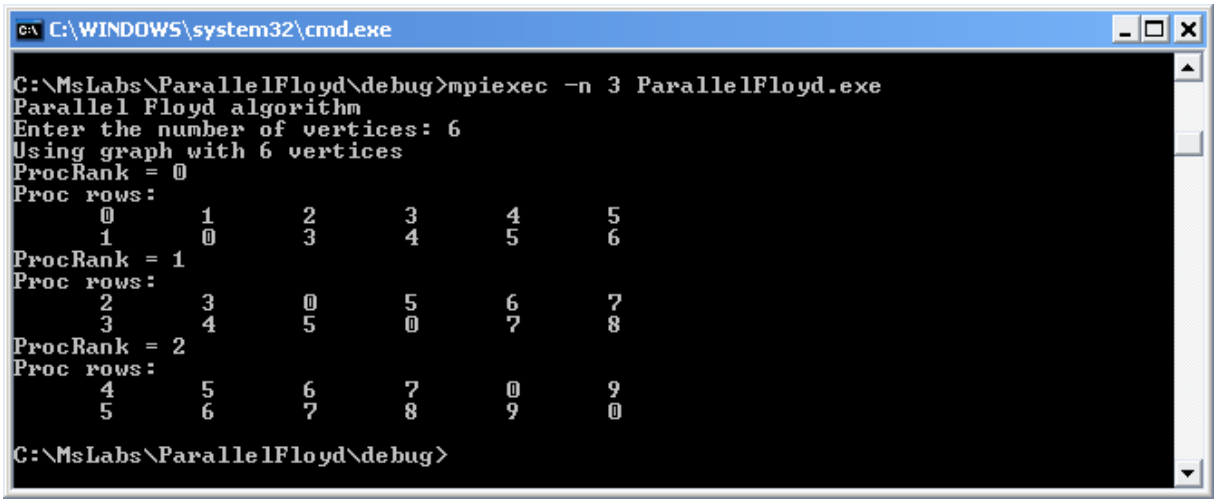

Рис. 5.15. Результат проверки работы алгоритма Флойда в случае, когда программа запускается на трех процессах, и количество вершин в графе равно шести

Скомпилируйте и запустите приложение. Проверьте правильность получения частичных результатов, задавая разное количество процессов и разное вершин в тестовом графе.

Задание 8 – Сбор результирующей матрицы

В завершение всех выполненных действий, организуем сбор полученной матрицы на ведущем процессе (на процессе с рангом 0). Следует отметить, что этот этап не является обязательным при выполнении алгоритма, поскольку количество обрабатываемых данных может оказаться столь значительным, что может и не поместиться в оперативной памяти одного компьютера. В данной лабораторной работе этот этап приведен как пример еще одного учебного задания, а также для итогового сравнения результатов параллельного и последовательного алгоритма Флойда.

Для сбора данных введем функцию *ResultCollection*, которая состоит практически только из вызова ранее уже применявшейся функции *MPI_Gather*:

```
// Function for process result collection
void ResultCollection(int *pMatrix, int *pProcRows, int Size, int RowNum) {
  MPI_Gather(pProcRows, RowNum * Size, MPI_INT, pMatrix, RowNum * Size,
    MPI_INT, 0, MPI_COMM_WORLD);
}
```

Вызов функции из основной программы:

```
  // Parallel Floyd algorithm
  ParallelFloyd(pProcRows, Size, RowNum);

  ParallelPrintMatrix(pProcRows, Size, RowNum);

  // Process data collection
  ResultCollection(pMatrix, pProcRows, Size, RowNum);
```

После выполнения сбора, добавьте в код основной функции приложения печать получившейся матрицы при помощи функции *PrintMatrix* на ведущем процессе параллельного приложения. Скомпилируйте и запустите приложение. Проверьте правильность работы программы.

Задание 9 – 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 ранее разработанную функцию *SeriaFloyd*, находящуюся в файле **ParallelFloydTest.cpp**.

Для того чтобы последовательный алгоритм *SerialFloyd* оперировал теми же данными, что и разработанный параллельный алгоритм *ParallelFloyd*, необходимо создать копию этих данных, используя функцию *CopyMatrix* (располагающуюся также в файле **ParallelFloyd.cpp**):

```
// Function for copying the matrix
void CopyMatrix(int *pMatrix, int Size, int *pMatrixCopy) {
  copy(pMatrix, pMatrix + Size * Size, pMatrixCopy);
```

```
}
```

Для проверки правильности сравним поэлементно результат последовательного алгоритма Флойда с результатом, полученным разработанным параллельным алгоритмом, при помощи функции *CompareMatrices*, также располагающейся в файле **ParallelFloydTest.cpp**:

```
// Function for comparing the matrices
bool CompareMatrices(int *pMatrix1, int *pMatrix2, int Size) {
  return equal(pMatrix1, pMatrix1 + Size * Size, pMatrix2);
}
```

Добавим вызовы этих функций в исходный код. В функции *main* необходимо объявить переменную, предназначенную для хранения копии матрицы смежности, участвующей в последовательном алгоритме Флойда, а также создать саму эту копию:

```
...
  int  RowNum;  // Number of process rows

  int *pSerialMatrix = 0;

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

  if(ProcRank == 0)
    printf("Parallel Floyd algorithm program\n");

  // Process initialization
  ProcessInitialization(pMatrix, pProcRows, Size, RowNum);

  if (ProcRank == 0) {
    // Matrix copying
    pSerialMatrix = new int[Size * Size];
    CopyMatrix(pMatrix, Size, pSerialMatrix);
  }
...
```

Кроме того, необходимо удалить занимаемую память, когда она становится больше не нужной:

```
...
  // Process termination
  ProcessTermination(pMatrix, pProcRows);
  if (ProcRank == 0)
    delete []pSerialMatrix;

  MPI_Finalize();

  return 0;
}
```

Далее, добавьте функцию *TestResult* в исходный текст программы:

```
// Function for testing the result of parallel Floyd algorithm
void TestResult(int *pMatrix, int *pSerialMatrix, int Size) {
  MPI_Barrier(MPI_COMM_WORLD);

  if(ProcRank == 0) {
    SerialFloyd(pSerialMatrix, Size);
    if(!CompareMatrices(pMatrix, pSerialMatrix, Size)) {
      printf("The results of serial and parallel algorithms are "
        "NOT identical. Check your code\n");
    }
    else {
      printf("The results of serial and parallel algorithms are "
        "identical\n");
    }
  }
}
```

Результатом работы этой функции является печать диагностического сообщения. Используя эту функцию, можно проверять результат работы параллельного алгоритма независимо от того, насколько велико количество вершин в графе.

Закомментируйте вызовы функций, использующих отладочную печать, которые ранее использовались для контроля правильности выполнения этапов параллельного приложения (функции *TestDistribution*, *ParallelPrintMatrix, PrintMatrix*). Вместо функции *DummyDataInitialization*, которая генерирует исходные данные простого вида, вызовите функцию *RandomDataInitialization*, которая генерирует исходные данные при помощи датчика случайных чисел. Скомпилируйте и запустите приложение. Выполните эксперименты для различных объемов исходных данных. Убедитесь в том, что программа работает правильно.

Задание 10 – Реализация алгоритма Флойда для графов с произвольным количеством вершин

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

Теперь рассмотрим случай, когда число вершин *Size* не кратно 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 смежности (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 количество вершин в графе не делится нацело на число процессов. Теперь необходимо определить, сколько строк матрицы смежности 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 $(Size - \lfloor Size/ProcNum \rfloor \cdot (ProcNum - 1))$ строки. Однако, в этом случае, возможно, что нагрузка будет распределена между процессами неравномерно. Так, например, если порядок матрицы смежности 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 allocating the memory and setting the initial values
void ProcessInitialization(int *&pMatrix, int *&pProcRows, int& Size,
  int& RowNum) {
...
  // Broadcast the number of vertices
  MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);

  // Number of rows for each process
  int RestRows = Size;
  for(int i = 0; i < ProcRank; i++)
    RestRows = RestRows - RestRows / (ProcNum - i);
  RowNum = RestRows / (ProcNum - ProcRank);

  // Allocate memory for the current process rows
  pProcRows = new int[Size * RowNum];
...
}
```

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 набор элементов всем процессам коммуникатора блоками разного размера.

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(int *pMatrix, int *pProcRows, 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

  // Allocate 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_INT,
    pProcRows, pSendNum[ProcRank], MPI_INT, 0, MPI_COMM_WORLD);

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

Аналогично для сбора данных, вместо функции *MPI_Gather*, ориентированной на сбор данных одинакового объема со всех процессов коммуникатора, будем использовать более общую функцию *MPI_Gatherv*.

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

```
// Function for process result collection
void ResultCollection(int *pMatrix, int *pProcRows, int Size, int RowNum) {
  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 gathered yet

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

  // Define the disposition of the result vector block of current process
  RowNum = Size / ProcNum;
  pReceiveInd[0] = 0;
  pReceiveNum[0] = RowNum * Size;

  for(int i = 1; i < ProcNum; i++) {
    RestRows      -= RowNum;
    RowNum = RestRows / (ProcNum - i);
    pReceiveNum[i] = RowNum * Size;
```

```
    pReceiveInd[i] = pReceiveInd[i - 1] + pReceiveNum[i - 1];
  }

  // Gather the whole matrix on process with rank 0
  MPI_Gatherv(pProcRows, pReceiveNum[ProcRank], MPI_INT,
    pMatrix, pReceiveNum, pReceiveInd, MPI_INT, 0, MPI_COMM_WORLD);

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

Кроме того, придется несколько изменить функцию *RowDistribution*, поскольку размеры полос могут теперь различаться и вычисление ранга процесса, на котором расположена строка матрицы с номером *k*, становится более сложным. Измените функцию *RowDistribution* для учета этого факта:

```
// Function for row broadcasting among all processes
void RowDistribution(int *pProcRows, int Size, int RowNum, int k,
  int *pRow) {
  int ProcRowRank;  // Process rank with the row k
  int ProcRowNum;   // Process row number

  // Finding the process rank with the row k
  int RestRows = Size;
  int Ind = 0;
  int Num = Size / ProcNum;

  for(ProcRowRank = 1; ProcRowRank < ProcNum + 1; ProcRowRank ++) {
    if(k < Ind + Num ) break;
    RestRows -= Num;
    Ind      += Num;
    Num       = RestRows / (ProcNum - ProcRowRank);
  }
  ProcRowRank = ProcRowRank - 1;
  ProcRowNum  = k - Ind;

  if(ProcRowRank == ProcRank)
    // Copy the row to pRow array
    copy(&pProcRows[ProcRowNum*Size],&pProcRows[(ProcRowNum+1)*Size],pRow);

  // Broadcast row to all processes
  MPI_Bcast(pRow, Size, MPI_INT, ProcRowRank, MPI_COMM_WORLD);
}
```

Скомпилируйте и запустите приложение. Проверьте правильность работы алгоритма Флойда при помощи функции *TestResult*.

Задание 11 – 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. Процессы параллельной программы могут быть запущены на разных процессорах вычислительной системы. При этом 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. Поскольку параллельный алгоритм включает этап распределения данных, работы алгоритма Флойда на каждом процессе и сбора обработанных значений, то отсчет времени должен начинаться непосредственно перед вызовом функции *DataDistribution*, и останавливаться сразу после выполнения функции *DataCollection*:

```
...
  double start, finish;
  double duration = 0.0;
...
  // Process initialization
  ProcessInitialization(pMatrix, pProcRows, Size, RowNum);
```

```
  start = MPI_Wtime();
  // Distributing the initial data between processes
  DataDistribution(pMatrix, pProcRows, Size, RowNum);

  // Testing the distribution
  //TestDistribution(pMatrix, pProcRows, Size, RowNum);

  // Parallel Floyd algorithm
  ParallelFloyd(pProcRows, Size, RowNum);

  //ParallelPrintMatrix(pProcRows, Size, RowNum);

  // Process data collection
  ResultCollection(pMatrix, pProcRows, Size, RowNum);
  finish = MPI_Wtime();

  //TestResult(pMatrix, pSerialMatrix, Size);

  duration = finish - start;
  if(ProcRank == 0)
    printf("Time of execution: %f\n", duration);
...
```

Очевидно, что таким образом будет распечатано то время, которое было затрачено на выполнение вычислений нулевым процессом. Возможно, что время выполнения алгоритма другими процессами может оказаться несколько отличным. Но эти отличия не должны быть существенными, поскольку на этапе разработки параллельного алгоритма было уделено особое внимание равномерной загрузке (*балансировке*) процессов.

Добавьте выделенный фрагмент кода в тело основной функции приложения. Скомпилируйте и запустите приложение. Выполните вычислительные эксперименты и заполните таблицу результатов:

Таблица 5.3. Результаты вычислительных экспериментов для параллельного алгоритма Флойда

Номер теста	Количество вершин	Последовательный алгоритм Флойда	Параллельный алгоритм Флойда для количества процессов		
			2	4	8
1	10				
2	500				
3	600				
4	700				
5	800				
6	900				
7	1000				

В графу "Последовательный алгоритм Флойда" внесите время выполнения последовательного алгоритма Флойда, замеренное при проведении тестирования последовательной программы в упражнении 2. Далее, рассчитайте получившееся ускорение вычислений как отношение времени последовательного алгоритма ко времени параллельного алгоритма и заполните таблицу.

Таблица 5.4. Ускорение вычислений, получаемое для параллельного алгоритма Флойда

Номер теста	Ускорение		
	2 процесса	4 процесса	8 процессов
1			
2			
3			
4			
5			
6			
7			

Для того чтобы оценить теоретическое время выполнения параллельного алгоритма, реализованного согласно вычислительной схеме, приведенной в упражнении 3, можно воспользоваться следующим соотношением:

$$T_p = n^2 \cdot \lceil n/p \rceil \cdot \tau + n \cdot \lceil \log_2(p) \rceil (\alpha + w \cdot n/\beta) \quad (5.2)$$

(подробный вывод данного выражения приведен в подразделе 11.1.5 раздела 11 "Параллельные алгоритмы обработки графов" учебных материалов курса). Здесь n – количество вершин графа, p – количество процессов, τ – время выполнения базовой операции выбора наименьшего значения (значение было нами вычислено при тестировании последовательного алгоритма), α – латентность и 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 (5.2). Результаты занесите в таблицу:

Таблица 5.5. Сравнение экспериментального и теоретического времени параллельного алгоритма Флойда

Номер теста	Количество данных	Время выполнения параллельного алгоритма					
		2 процесса		4 процесса		8 процессов	
		Модель	Эксперимент	Модель	Эксперимент	Модель	Эксперимент
1	10						
2	500						
3	600						
4	700						
5	800						
6	900						
7	1000						

Контрольные вопросы

- Насколько сильно отличаются времена, затраченные на выполнение последовательного алгоритма Флойда и параллельного алгоритма? Почему?
- Получилось ли ускорение при обработке графа из 10 вершин? Почему?
- Насколько хорошо совпадают время, полученное теоретически, и реальное время выполнения алгоритма? В чем может состоять причина несовпадений?

Задания для самостоятельной работы

- Изучите другие параллельные алгоритмы обработки графов (алгоритм Прима для решения задачи о нахождении минимального охватывающего дерева, метод Дейкстры для решения задачи о нахождении кратчайших путей от одной из вершин графа до всех остальных – см. раздел 11 "Параллельные алгоритмы обработки графов" учебных материалов курса). Разработайте программы, реализующие эти алгоритмы.

Приложение 1. Программный код последовательного приложения, реализующего алгоритм Флойда

Файл SerialFloyd.cpp

```
#include <cstdlib>
#include <cstdio>
#include <ctime>
#include <algorithm>

#include "SerialFloyd.h"
#include "SerialFloydTest.h"

using namespace std;
```

```
const double InfinitiesPercent = 50.0;
const double RandomDataMultiplier = 10;

int Min(int A, int B) {
  int Result = (A < B) ? A : B;

  if((A < 0) && (B >= 0)) Result = B;
  if((B < 0) && (A >= 0)) Result = A;
  if((A < 0) && (B < 0))  Result = -1;

  return Result;
}

int main(int argc, char* argv[]) {
  int *pMatrix;    // Adjacency matrix
  int  Size;       // Size of adjacency matrix

  time_t start, finish;
  double duration = 0.0;

  printf("Serial Floyd algorithm\n");

  // Process initialization
  ProcessInitialization(pMatrix, Size);

  printf("The matrix before Floyd algorithm\n");
  PrintMatrix(pMatrix, Size, Size);

  start = clock();
  // Parallel Floyd algorithm
  SerialFloyd(pMatrix, Size);
  finish = clock();

  printf("The matrix after Floyd algorithm\n");
  PrintMatrix(pMatrix, Size, Size);

  duration = (finish - start) / double(CLOCKS_PER_SEC);

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

  // Ending of processing
  ProcessTermination(pMatrix);

  return 0;
}

// Function for allocating the memory and setting the initial values
void ProcessInitiliazation(int *&pMatrix, int& Size) {
  do {
    printf("Enter the number of vertices: ");

    scanf("%d", &Size);

    if(Size <= 0)
      printf("The number of vertices should be greater then zero\n");
  } while(Size <= 0);

  printf("Using graph with %d vertices\n", Size);

  // Allocate memory for the adjacency matrix
  pMatrix = new int[Size * Size];
```

```
  // Data initalization
  DummyDataInitialization(pMatrix, Size);
  //RandomDataInitialization(pMatrix, Size);
}

// Function for computational process termination
void ProcessTermination(int *pMatrix) {
  delete []pMatrix;
}

// Function for simple setting the initial data
void DummyDataInitialization(int *pMatrix, int Size) {
  for(int i = 0; i < Size; i++)
    for(int j = i; j < Size; j++) {
      if(i == j) pMatrix[i * Size + j] = 0;
      else
        if(i == 0) pMatrix[i * Size + j] = j;
        else       pMatrix[i * Size + j] = -1;

      pMatrix[j * Size + i] = pMatrix[i * Size + j];
    }
}

// Function for initializing the data by the random generator
void RandomDataInitialization(int *pMatrix, int Size) {
  srand( (unsigned)time(0) );

  for(int i = 0; i < Size; i++)
    for(int j = 0; j < Size; j++)
      if(i != j) {
        if((rand() % 100) < InfinitiesPercent)
          pMatrix[i * Size + j] = -1;
        else
          pMatrix[i * Size + j] = rand() + 1;
      }
      else
        pMatrix[i * Size + j] = 0;
}

// Serial Floyd algorithm
void SerialFloyd(int *pMatrix, int Size) {
  int t1, t2;
  for(int k = 0; k < Size; k++)
    for(int i = 0; i < Size; i++)
      for(int j = 0; j < Size; j++)
        if((pMatrix[i * Size + k] != -1) &&
           (pMatrix[k * Size + j] != -1)) {
          t1 = pMatrix[i * Size + j];
          t2 = pMatrix[i * Size + k] + pMatrix[k * Size + j];
          pMatrix[i * Size + j] = Min(t1, t2);
      }
}
```

Файл SerialFloydTest.cpp

```
#include <cstdio>
#include "SerialFloydTest.h"

using namespace std;

// Function for formatted matrix output
void PrintMatrix(int *pMatrix, int RowCount, int ColCount) {
  for(int i = 0; i < RowCount; i++) {
```

```
    for(int j = 0; j < ColCount; j++)
      printf("%7d", pMatrix[i * ColCount + j]);
    printf("\n");
  }
}
```

Приложение 2. Программный код параллельного приложения, реализующего алгоритм Флойда

Файл ParallelFloyd.cpp

```
#include <cstdlib>
#include <cstdio>
#include <ctime>
#include <algorithm>
#include <mpi.h>

#include "ParallelFloyd.h"
#include "ParallelFloydTest.h"

using namespace std;

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

const double InfinitiesPercent = 50.0;
const double RandomDataMultiplier = 10;

int Min(int A, int B) {
  int Result = (A < B) ? A : B;

  if((A < 0) && (B >= 0)) Result = B;
  if((B < 0) && (A >= 0)) Result = A;
  if((A < 0) && (B < 0))  Result = -1;

  return Result;
}

int main(int argc, char* argv[]) {
  int *pMatrix;      // Adjacency matrix
  int  Size;         // Size of adjacency matrix
  int *pProcRows;    // Process rows
  int  RowNum;       // Number of process rows

  double start, finish;
  double duration = 0.0;

  int *pSerialMatrix = 0;

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

  if(ProcRank == 0)
    printf("Parallel Floyd algorithm\n");

  // Process initialization
  ProcessInitialization(pMatrix, pProcRows, Size, RowNum);

  if (ProcRank == 0) {
      // Matrix copying
    pSerialMatrix = new int[Size * Size];
    CopyMatrix(pMatrix, Size, pSerialMatrix);
```

```
  }

  start = MPI_Wtime();
  // Distributing the initial data between processes
  DataDistribution(pMatrix, pProcRows, Size, RowNum);
  // Testing the distribution
  //TestDistribution(pMatrix, pProcRows, Size, RowNum);

  // Parallel Floyd algorithm
  ParallelFloyd(pProcRows, Size, RowNum);
  //ParallelPrintMatrix(pProcRows, Size, RowNum);

  // Process data collection
  ResultCollection(pMatrix, pProcRows, Size, RowNum);
  //if(ProcRank == 0)
  //  PrintMatrix(pMatrix, Size, Size);
  finish = MPI_Wtime();

  //TestResult(pMatrix, pSerialMatrix, Size);

  duration = finish - start;
  if(ProcRank == 0)
    printf("Time of execution: %f\n", duration);

  if (ProcRank == 0)
    delete []pSerialMatrix;

  // Process termination
  ProcessTermination(pMatrix, pProcRows);

  MPI_Finalize();
  return 0;
}

// Function for allocating the memory and setting the initial values
void ProcessInitialization(int *&pMatrix, int *&pProcRows, int& Size, int&
RowNum) {
  setvbuf(stdout, 0, _IONBF, 0);

  if(ProcRank == 0) {
    do {
      printf("Enter the number of vertices: ");

      scanf("%d", &Size);

      if(Size < ProcNum)
        printf("The number of vertices should be greater then number of
processes\n");
    } while(Size < ProcNum);

    printf("Using graph with %d vertices\n", Size);
  }

  // Broadcast the number of vertices
  MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);

  // Number of rows for each process
  int RestRows = Size;
  for(int i = 0; i < ProcRank; i++)
    RestRows = RestRows - RestRows / (ProcNum - i);
  RowNum = RestRows / (ProcNum - ProcRank);

  // Allocate memory for the current process rows
```

```
  pProcRows = new int[Size * RowNum];

  if(ProcRank == 0) {
    // Allocate memory for the adjacency matrix
    pMatrix = new int[Size * Size];

    // Data initalization
    DummyDataInitialization(pMatrix, Size);
    //RandomDataInitialization(pMatrix, Size);
  }
}

// Function for computational process termination
void ProcessTermination(int *pMatrix, int *pProcRows) {
  if(ProcRank == 0)
    delete []pMatrix;

  delete []pProcRows;
}

// Function for simple setting the initial data
void DummyDataInitialization(int *pMatrix, int Size) {
  for(int i = 0; i < Size; i++)
    for(int j = i; j < Size; j++) {
      if(i == j) pMatrix[i * Size + j] = 0;
      else
        if(i == 0) pMatrix[i * Size + j] = j;
        else       pMatrix[i * Size + j] = -1;

      pMatrix[j * Size + i] = pMatrix[i * Size + j];
    }
}

// Function for setting the data by the random generator
void RandomDataInitialization(int *pMatrix, int Size) {
  srand( (unsigned)time(0) );

  for(int i = 0; i < Size; i++)
    for(int j = 0; j < Size; j++)
      if(i != j) {
        if((rand() % 100) < InfinitiesPercent)
          pMatrix[i * Size + j] = -1;
        else
          pMatrix[i * Size + j] = rand() + 1;
      }
      else
        pMatrix[i * Size + j] = 0;
}

// Data distribution among the processes
void DataDistribution(int *pMatrix, int *pProcRows, 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 havenTt been distributed yet

  // Allocate 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_INT,
    pProcRows, pSendNum[ProcRank], MPI_INT, 0, MPI_COMM_WORLD);

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

// Function for process result collection
void ResultCollection(int *pMatrix, int *pProcRows, int Size, int RowNum) {
  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 havenTt been gathered yet

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

  // Define the disposition of the result vector block of current process
  RowNum = Size / ProcNum;
  pReceiveInd[0] = 0;
  pReceiveNum[0] = RowNum * Size;

  for(int i = 1; i < ProcNum; i++) {
    RestRows      -= RowNum;
    RowNum = RestRows / (ProcNum - i);
    pReceiveNum[i] = RowNum * Size;
    pReceiveInd[i] = pReceiveInd[i - 1] + pReceiveNum[i - 1];
  }

  // Gather the whole matrix on process with rank 0
  MPI_Gatherv(pProcRows, pReceiveNum[ProcRank], MPI_INT,
    pMatrix, pReceiveNum, pReceiveInd, MPI_INT, 0, MPI_COMM_WORLD);

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

// Parallel Floyd algorithm
void ParallelFloyd(int *pProcRows, int Size, int RowNum) {
  int *pRow = new int[Size];
  int t1, t2;
  for(int k = 0; k < Size; k++) {
    // Distribute row among all processes
    RowDistribution(pProcRows, Size, RowNum, k, pRow);

    // Update adjacency matrix elements
    for(int i = 0; i < RowNum; i++)
      for(int j = 0; j < Size; j++)
        if( (pProcRows[i * Size + k] != -1) &&
            (pRow     [j]            != -1)) {
          t1 = pProcRows[i * Size + j];
```

```
          t2 = pProcRows[i * Size + k] + pRow[j];

        pProcRows[i * Size + j] = Min(t1, t2);
      }
  }

  delete []pRow;
}

// Function for row broadcasting among all processes
void RowDistribution(int *pProcRows, int Size, int RowNum, int k, int
*pRow) {
  int ProcRowRank;  // Process rank with the row k
  int ProcRowNum;   // Process row number

  // Finding the process rank with the row k
  int RestRows = Size;
  int Ind = 0;
  int Num = Size / ProcNum;

  for(ProcRowRank = 1; ProcRowRank < ProcNum + 1; ProcRowRank ++) {
    if(k < Ind + Num ) break;
    RestRows -= Num;
    Ind      += Num;
    Num       = RestRows / (ProcNum - ProcRowRank);
  }
  ProcRowRank = ProcRowRank - 1;
  ProcRowNum  = k - Ind;

  if(ProcRowRank == ProcRank)
    // Copy the row to pRow array
    copy(&pProcRows[ProcRowNum*Size],&pProcRows[(ProcRowNum+1)*Size],pRow);

  // Broadcast row to all processes
  MPI_Bcast(pRow, Size, MPI_INT, ProcRowRank, MPI_COMM_WORLD);
}

// Function for formatted output of all stripes
void ParallelPrintMatrix(int *pProcRows, int Size, int RowNum) {
  for(int i = 0; i < ProcNum; i++) {
    MPI_Barrier(MPI_COMM_WORLD);
    if (ProcRank == i) {
      printf("ProcRank = %d\n", ProcRank);
      fflush(stdout);
      printf("Proc rows:\n");
      fflush(stdout);
      PrintMatrix(pProcRows, RowNum, Size);
      fflush(stdout);
    }
    MPI_Barrier(MPI_COMM_WORLD);
  }
}

// Function for testing the data distribution
void TestDistribution(int *pMatrix, int *pProcRows, int Size, int RowNum) {
  MPI_Barrier(MPI_COMM_WORLD);
  if (ProcRank == 0) {
    printf("Initial adjacency matrix:\n");
    PrintMatrix(pMatrix, Size, Size);
  }

  MPI_Barrier(MPI_COMM_WORLD);
```

```
  ParallelPrintMatrix(pProcRows, Size, RowNum);
}

// Testing the result of parallel Floyd algorithm
void TestResult(int *pMatrix, int *pSerialMatrix, int Size) {
  MPI_Barrier(MPI_COMM_WORLD);

  if(ProcRank == 0) {
    SerialFloyd(pSerialMatrix, Size);
    if(!CompareMatrices(pMatrix, pSerialMatrix, Size)) {
      printf("Results of serial and parallel algorithms are "
        "NOT identical. Check your code\n");
    }
    else {
      printf("Results of serial and parallel algorithms are "
        "identical\n");
    }
  }
}
```

Файл ParallelFloydTest.cpp

```
#include <cstdio>
#include <algorithm>
using namespace std;

#include "ParallelFloyd.h"
#include "ParallelFloydTest.h"

// Function for copying the matrix
void CopyMatrix(int *pMatrix, int Size, int *pMatrixCopy) {
  copy(pMatrix, pMatrix + Size * Size, pMatrixCopy);
}

// Function for comparing the matrices
bool CompareMatrices(int *pMatrix1, int *pMatrix2, int Size) {
  return equal(pMatrix1, pMatrix1 + Size * Size, pMatrix2);
}

// Serial Floyd algorithm
void SerialFloyd(int *pMatrix, int Size) {
  int t1, t2;
  for(int k = 0; k < Size; k++)
    for(int i = 0; i < Size; i++)
      for(int j = 0; j < Size; j++)
        if((pMatrix[i * Size + k] != -1) &&
           (pMatrix[k * Size + j] != -1)) {
          t1 = pMatrix[i * Size + j];
          t2 = pMatrix[i * Size + k] + pMatrix[k * Size + j];
          pMatrix[i * Size + j] = Min(t1, t2);
      }
}

// Function for formatted matrix output
void PrintMatrix(int *pMatrix, int RowCount, int ColCount) {
  for(int i = 0; i < RowCount; i++) {
    for(int j = 0; j < ColCount; j++) {
      printf("%7d", pMatrix[i * ColCount + j]);
      fflush(stdout);
    }
    printf("\n");
    fflush(stdout);
  }
```

```
}
```