Лабораторная работа 6: Параллельные алгоритмы решения дифференциальных уравнений в частных производных

Дифференциальные уравнения в частных производных представляют собой широко применяемый математический аппарат при разработке моделей в самых разных областях науки и техники. К сожалению, явное решение этих уравнений в аналитическом виде оказывается возможным только в частных простых случаях, и, как результат, возможность анализа математических моделей, построенных на основе дифференциальных уравнений, обеспечивается при помощи приближенных численных методов решения. Объем выполняемых при этом вычислений обычно является значительным, и использование высокопроизводительных вычислительных систем является традиционным для данной области вычислительной математики. Проблематика численного решения дифференциальных уравнений в частных производных является областью интенсивных исследований.

Цель лабораторной работы

Целью данной лабораторной работы является разработка параллельной программы, которая обеспечивает решение одной из задач, описываемой дифференциальным уравнением в частных производных – *задачи Дирихле для уравнения Пуассона*.

- Упражнение 1 – Постановка задачи Дирихле
- Упражнение 2 – Реализация последовательного алгоритма Гаусса – Зейделя решения задачи Дирихле
- Упражнение 3 – Разработка параллельного алгоритма Гаусса – Зейделя решения задачи Дирихле
- Упражнение 4 - Реализация параллельного алгоритма Гаусса – Зейделя решения задачи Дирихле

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 12 "Параллельные методы решения дифференциальных уравнений в частных производных". Кроме того, предполагается, что выполнена ознакомительная лабораторная работа "Параллельное программирование с использованием MPI".

Упражнение 1 – Постановка задачи Дирихле

В лабораторной работе рассматривается *проблема численного решения задачи Дирихле для уравнения Пуассона*, определяемая как задача нахождения функции *u=u(x,y)*, удовлетворяющей в области определения *D* уравнению:

$$\frac{\delta^2 u}{\delta x^2} + \frac{\delta^2 u}{\delta y^2} = f(x,y), \quad (x.y) \in D,$$

и принимающей значения *g(x,y)* на границе D^0 области *D* (*f* и *g* являются функциями, задаваемыми при постановке задачи). Подобная модель может быть использована для описания установившегося течения жидкости, стационарных тепловых полей, процессов теплопередачи с внутренними источниками тепла и деформации упругих пластин и др. Данный пример часто используется в качестве учебно-практической задачи при изложении возможных способов организации эффективных параллельных вычислений (см. раздел 12 "Параллельные методы решения дифференциальных уравнений в частных производных").

Для простоты изложения материала в качестве области задания функции далее будет использоваться единичный квадрат:

$$D = \{(x,y) \in R^2 : 0 \le x, y \le 1\}.$$

Рассматриваемый в лабораторной работе *метод конечных разностей* (*метод сеток*) для решения поставленной задачи является одним из наиболее распространенных подходов численного решения дифференциальных уравнений. Следуя этому методу, область решения D представляется в виде дискретного (как правило, равномерного) набора (*сетки*) точек (*узлов*). Так, например, прямоугольная сетка в области D может быть задана в виде (рис. 12.1)

$$\begin{cases} D_h = \{(x_i, y_j) : x_i = ih,\ y_i = jh,\ 0 \le i, j \le N+1, \\ \qquad\qquad h = 1/(N+1), \end{cases}$$

где величина N задает количество внутренних узлов по каждой из координат области D.

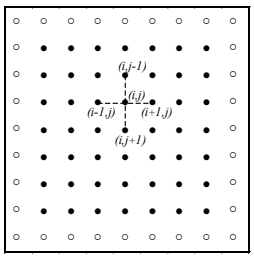

Рис. 6.1. Прямоугольная сетка в области D (темные точки представляют внутренние узлы сетки, нумерация узлов в строках слева направо, а в столбцах - сверху вниз)

Обозначим оцениваемую при подобном дискретном представлении аппроксимацию функции $u(x,y)$ в точках (x_i, y_j) через u_{ij}. Тогда, используя *пятиточечный шаблон* (см. рис. 6.1) для вычисления значений производных, мы можем представить уравнение Пуассона в *конечно-разностной форме*

$$\frac{u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{ij}}{h^2} = f_{ij}.$$

Данное уравнение может быть разрешено относительно u_{ij}

$$u_{ij} = 0.25(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - h^2 f_{ij}).$$

Разностное уравнение, записанное в подобной форме, позволяет определять значение u_{ij} по известным значениям функции $u(x,y)$ в соседних узлах используемого шаблона. Данный результат служит основой для построения различных *итерационных схем* решения задачи Дирихле, в которых в начале вычислений формируется некоторое приближение для значений u_{ij}, а затем эти значения последовательно уточняются в соответствии с приведенным соотношением. Так, например, *метод Гаусса-Зейделя* для проведения итераций уточнения использует правило

$$u_{ij}^k = 0.25(u_{i-1,j}^k + u_{i+1,j}^{k-1} + u_{i,j-1}^k + u_{i,j+1}^{k-1} - h^2 f_{ij}),$$

по которому очередное *k*-ое приближение значения u_{ij} вычисляется по последнему *k*-ому приближению значений $u_{i-1,j}$ и $u_{i,j-1}$ и предпоследнему (*k-1*)-ому приближению значений $u_{i+1,j}$ и $u_{i,j+1}$. Выполнение итераций обычно продолжается до тех пор, пока получаемые в результате итераций изменения значений u_{ij} не станут меньше некоторой заданной величины (*требуемой точности вычислений*). Последовательность решений, получаемых методом сеток, равномерно сходится к решению задачи Дирихле, а погрешность решения имеет порядок h^2.

Псевдокод для рассмотренного алгоритма Гаусса – Зейделя решения задачи Дирихле может быть представлен следующим образом:

```
// Serial Gauss-Seidel algorithm
do {
  dmax = 0; // maximal variation of the values u
  for ( i=1; i<N+1; i++ )
    for ( j=1; j<N+1; j++ ) {
      temp = u[i][j];
      u[i][j] = 0.25*(u[i-1][j]+u[i+1][j]+
                u[i][j-1]+u[i][j+1]-h*h*f[i][j]);
      dm = fabs(temp-u[i][j]);
      if ( dmax < dm ) dmax = dm;
    }
} while ( dmax > eps );
```

В дальнейшем при разработке программ для снижения сложности выполняемой лабораторной работы будем полагать, что функций *f* тождественно равно нулю, т.е. *f(x,y)≡0*.

Упражнение 2 – Реализация последовательного алгоритма Гаусса – Зейделя решения задачи Дирихле

При выполнении этого упражнения необходимо реализовать последовательный алгоритм Гаусса – Зейделя решения задачи Дирихле. Начальный вариант будущей программы представлен в проекте *SerailGaussSeidel*, который содержит часть исходного кода и в котором заданы необходимые параметры проекта. В ходе выполнения упражнения необходимо дополнить имеющийся вариант программы операциями ввода исходных данных, решения задачи Дирихле с использованием алгоритма Гаусса-Зейделя и вывода результатов.

Задание 1 – Открытие проекта SerialGaussSeidel

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

После открытия проекта в окне Solution Explorer (Ctrl+Alt+L) дважды щелкните на файле исходного кода *SerialGS.cpp*, как это показано на рис. 6.2. После этих действий код, который предстоит в дальнейшем расширить, будет открыт в рабочей области Visual Studio.

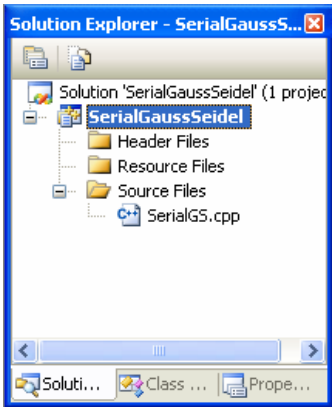

Рис. 6.2. Открытие файла SerialGS.cpp

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

Рассмотрим переменные, которые используются в основной функции (*main*) нашего приложения. Первая переменная (*pMatrix*) – это матрица, в которой хранятся значения точек заданной области. Переменная *Size* определяет размер матрицы (предполагаем, что матрица *pMatrix* квадратная, имеет размерность *Size×Size)*. Переменная *Eps* используется для хранения требуемой точности решения задачи. Для запоминания количества выполненных итераций алгоритма Гаусса-Зейделя используется переменная *Iterations*.

```
double* pMatrix;     // Matrix of the grid nodes
int     Size;        // Matrix size
double  Eps;         // Required accuracy
int     Iterations; // Iteration number
```

Заметим, что для хранения матрицы *pMatrix* используется одномерный массив, в котором матрица хранится построчно. Таким образом, элемент, расположенный на пересечении *i*-ой строки и *j*-ого столбца матрицы, в одномерном массиве имеет индекс *i*Size+j*.

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

```
printf ("Serial Gauss - Seidel algorithm\n");
getch();
```

Теперь можно осуществить первый запуск приложения. Выполните команду **Rebuild Solution** в меню **Build** – эта команда позволяет произвести полное перекомпилирование и линковку проекта . Если приложение скомпилировано успешно (в нижней части окна Visual Studio появилось сообщение "`Rebuild All: 1 succeeded, 0 failed, 0 skipped`"), нажмите клавишу **F5** или выполните команду **Start Debugging** пункта меню **Debug**.

Сразу после запуска кода, в командной консоли появится сообщение:

"`Serial Gauss-Seidel algorithm`".

Для того, чтобы завершить выполнение программы, нажмите любую клавишу.

Задание 2 – Ввод исходных данных

Для задания исходных данных последовательного алгоритма Гаусса – Зейделя решения задачи Дирихле разработаем функцию *ProcessInitialization*. Эта функция предназначена для определения размера сетки в области решения (размера матрицы *pMatrix*), требуемой точности *Eps* решения задачи и выделения необходимой памяти. Функция будет иметь следующий интерфейс:

```
// Function for memory allocation and initialization of grid nodes
void ProcessInitialization (double* &pMatrix, int &Size, double &Eps);
```

На самом первом этапе необходимо определить размер сетки в области поиска (задать значение переменной *Size*). В тело функции *ProcessInitialization* добавьте выделенный фрагмент кода:

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

Пользователю предоставляется возможность ввести размер сетки, который запоминается далее в целочисленной переменной *Size*. Затем печатается значение переменной *Size* (рис. 6.3).

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

```
void main() {
  double* pMatrix;    // Matrix of the grid nodes
  int     Size;       // Matrix size
..double  Eps;        // Required accuracy
  int     Iterations; // Iteration number

  printf ("Serial Gauss - Seidel algorithm\n");

  // Process initialization
  ProcessInitialization(pMatrix, Size, Eps);
  getch();
}
```

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

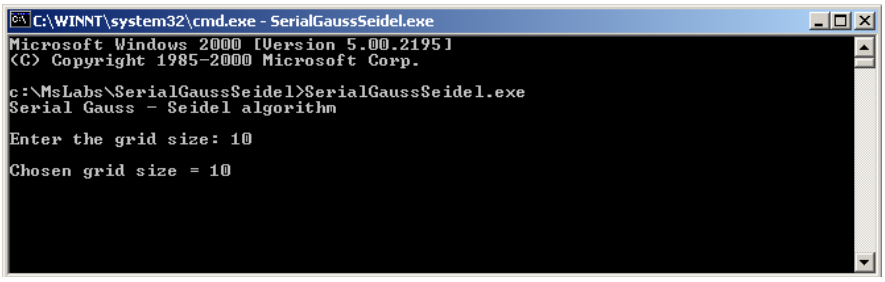

Рис. 6.3. Задание размера сетки

Как и в других лабораторных работах, выполним контроль правильности ввода данных. Организуем проверку размера сетки и, в случае ошибки (заданный размер меньше 3), продолжим запрашивать размер сетки до тех пор, пока не будет введено допустимое значение. Для реализации такого поведения поместим фрагмент кода, который производит ввод размера сетки, в цикл с постусловием:

```
  // Setting the grid size
  do {
    printf("\nEnter the grid size: ");
    scanf("%d", &Size);
    printf("\nChosen grid size = %d", Size);
    if (Size <= 2)
      printf("\nSize of grid must be greater than 2!\n");
  }
  while (Size <= 2);
```

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

Организуем теперь ввод значения требуемой точности *Eps*. Добавьте следующий код в текст функции *ProcessInitialization*:

```
  // Setting the required accuracy
  do {
    printf("\nEnter the required accuracy: ");
    scanf("%lf", &Eps);
    printf("\nChosen accuracy = %lf", Eps);
    if (Eps <= 0)
      printf("\nAccuracy must be greater than 0!\n");
  }
  while (Eps <= 0);
```

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

Задание 3 – Задание начальных значений

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

```
// Function for memory allocation and initialization of grid nodes
void ProcessInitialization (double* &pMatrix, int &Size) {
    <…>

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

Далее необходимо задать значения всех элементов матрицы *pMatrix*. Для выполнения этих действий реализуем еще одну функцию *DummyDataInitialization*. Интерфейс и реализация этой функции представлены ниже:

```
// Function for simple setting the grid node values
void DummyDataInitialization (double* pMatrix, int Size) {
  int i, j;  // Loop variables
  // Setting the grid node values
  for (i=0; i<Size; i++) {
    for (j=0; j<Size; j++)
      if ((i==0) || (i== Size-1) || (j==0) || (j==Size-1))
        pMatrix[i*Size+j] = 100;
      else
        pMatrix[i*Size+j] = 0;
  }
}
```

Как видно из представленного фрагмента кода, данная функция осуществляет задание значений функции *u* в узлах сетки. При этом для всех внутренних узлов сетки устанавливается значение 0, для всех граничных узлов задается значение 100 (т.е. функция *g* для задания значений на границе области является равной *g≡100*).

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

```
// Function for memory allocation and initialization of grid nodes
void ProcessInitialization (double* &pMatrix, int &Size, double &Eps) {

  // Memory allocation
  <…>

  // Setting the grid node values
  DummyDataInitialization(pMatrix, Size);
}
```

Реализуем функцию, которая поможет контролировать задание исходных данных. Это функция форматированного вывода матрицы *PrintMatrix*. 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 (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");
  }
}
```

Добавим вызов этой функции в основную функцию приложения:

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

  // Matrix output
  printf ("Initial Matrix: \n");
  PrintMatrix (pMatrix, Size, Size);
```

Скомпилируйте и запустите приложение. Убедитесь в том, что задание данных происходит по описанным правилам (рис. 6.4). Выполните несколько запусков приложения, задавайте различные размеры сетки.

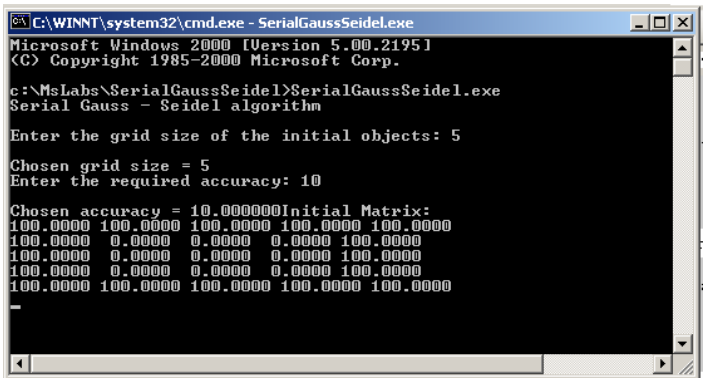

Рис. 6.4. Результат работы программы при завершении задания 3

Задание 4 – Завершение процесса вычислений

Перед реализацией алгоритма Гаусса-Зейделя для решения поставленной задачи 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 исходной матрицы *pMatrix* и этот массив необходимо передать в функцию *ProcessTermination* в качестве аргумента:

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

Вызов функции *ProcessTermination* необходимо выполнить перед завершением функции *main*:

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

  // Matrix output
  printf ("Initial Matrix: \n");
  PrintMatrix (pMatrix, Size, Size);

  // Computational process termination
  ProcessTermination(pMatrix);
```

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

Задание 5 – Реализация алгоритма Гаусса-Зейделя

Выполним теперь разработку основной вычислительной части программы. Для реализации алгоритма Гаусса-Зейделя для решения задачи Дирихле реализуем функцию *ResultCalculation*, которая принимает на вход исходные матрицу *pMatrix*, размер матрицы *Size*, требуемую точность *Eps* решения задачи. Как выходной параметр добавим переменную *Iterations*, в которой функция будет возвращать количество выполненных итераций алгоритма Гаусса-Зейделя до достижения требуемой точности.

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

```
// Function for the Gauss-Seidel algoritm
void ResultCalculation(double* pMatrix, int Size, double &Eps,
  int &Iterations) {
  int i, j;  // Loop variables
  double dm, dmax,temp;
  Iterations = 0;
  do {
    dmax = 0;
    for (i = 1; i < Size - 1; i++)
      for(j = 1; j < Size - 1; j++) {
        temp = pMatrix[Size * i + j];
        pMatrix[Size * i + j] = 0.25 * (pMatrix[Size * i + j + 1] +
                                        pMatrix[Size * i + j - 1] +
                                        pMatrix[Size * (i + 1) + j] +
                                        pMatrix[Size * (i - 1) + j]);
        dm = fabs(pMatrix[Size * i + j] - temp);
        if (dmax < dm) dmax = dm;
      }
      Iterations++;
  }
  while (dmax > Eps);
}
```

Выполним вызов функции, реализующий алгоритм Гаусса – Зейделя, из основной программы. Для контроля правильности работы алгоритма распечатаем матрицу значений после выполнения алгоритма:

```
  // Memory allocation and setting the initial values
  ProcessInitialization(pMatrix, Size);

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

  // The Gauss-Seidel method
  ResultCalculation(pMatrix, Size, Eps, Iterations);

  // Printing the result
  printf("\n Number of iterations: %d\n",Iterations);
  printf ("\n Result matrix: \n");
  PrintMatrix (pMatrix, Size, Size);

  // Computational process termination
  ProcessTermination(pMatrix);
```

Скомпилируйте и запустите приложение. Проанализируйте результат работы алгоритма Гаусса-Зейделя. Проведите несколько вычислительных экспериментов, изменяя размеры вычислительной сетки.

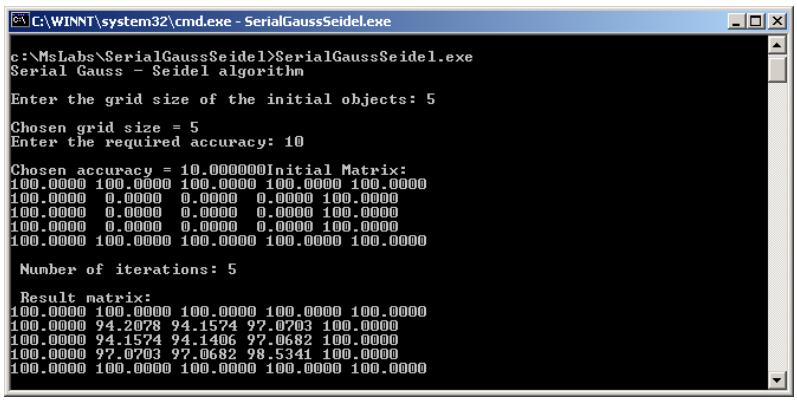

Рис. 6.5. Результат работы алгоритма Гаусса-Зейделя

Задание 6 – Проведение вычислительных экспериментов

Для последующей оценки получаемого ускорения решения задачи при использовании параллельного алгоритма необходимо провести эксперименты по вычислению времени выполнения последовательного алгоритма. Анализ времени выполнения алгоритма целесообразно проводить для достаточно больших размеров вычислительной сетки.

Для определения времени добавьте в разработанную программу вызовы функций, позволяющие узнать время работы программы или её части. Воспользуемся функцией:

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

Эта функция возвращает количество тактов (тиков) процессора, прошедших с момента старта системы. Добавим в программный код вычисление и вывод времени непосредственного решения задачи Дирихле, для этого поставим замеры времени до и после вызова функции *ResultCalculation*:

```
    // The Gauss-Seidel method
    start = clock();
    ResultCalculation(pMatrix, Size, Eps, Iterations);
    finish = clock();
    duration = (finish-start)/double(CLOCKS_PER_SEC);

    // Printing the results
    printf("\n Number of iterations: %d\n",Iterations);
    // Printing the time spent by the Gauss-Seidel method
    printf("\n Time of execution: %f", duration);
```

Скомпилируйте и запустите приложение. Для проведения вычислительных экспериментов с большими вычислительными сетками, отмените печать матрицы результатов (закомментируйте соответствующие строки кода). Проведите вычислительные эксперименты, результаты занесите в таблицу 6.1.

Таблица 6.1. Результаты вычислительных экспериментов для метода Гаусса-Зейделя

Номер теста	Размер матрицы	Количество итераций	Время работы (сек)
1	10		
2	100		
3	1000		
4	2000		
5	3000		
6	4000		

Оценим аналитически трудоемкость алгоритма Гаусса-Зейделя (см. раздел 12 "Параллельные методы решения дифференциальных уравнений в частных производных" учебных материалов курса). В общем виде время решения задачи может быть оценено в соответствии с выражением:

$$T_1 = kmN^2\tau \,. \qquad (6.1)$$

где N есть количество узлов по каждой из координат области D, m - число операций, выполняемых методом для одного узла сетки ($m=6$), k - количество итераций метода до выполнения условия остановки, а τ есть время выполнения базовой вычислительной операции.

Заполним таблицу сравнения реального времени выполнения со временем, которое может быть получено по формуле (6.1). Для вычисления времени выполнения базовой вычислительной операции применим следующую методику: выберем один из экспериментов как образец. Пусть, например, в качестве образца выступает эксперимент, где размер матриц равен 2000. Время выполнения этого эксперимента поделим на число выполненных операций (число операций может быть вычислено по формуле (6.1)). Таким образом, вычислим время выполнения одной операции. Далее, используя это значение, вычислим теоретическое время выполнения для всех оставшихся экспериментов. Результаты занесите в таблицу 6.2.

Таблица 6.2. Сравнение экспериментального и теоретического времени метода Гаусса-Зейделя

Время выполнения базовой вычислительной операции τ (сек):			
Номер теста	Размер матрицы	Время работы (сек)	Теоретическое время (сек)
1	10		
2	100		
3	1000		
4	2000		
5	3000		
6	4000		

Упражнение 3 – Разработка параллельного алгоритма Гаусса-Зейделя решения задачи Дирихле

При разработке параллельных алгоритмов необходимо выбрать способ разделения обрабатываемых данных между вычислительными серверами. При построении параллельных способов решения задачи Дирихле возможны два различных способа разделения данных – *одномерная* или *ленточная* схема или *двухмерное* или *блочное* разбиение вычислительной сетки.

В дальнейшем выполним более подробное рассмотрение ленточной схемы организации вычислений (см. раздел 12 "Параллельные методы решения дифференциальных уравнений в частных производных" учебных материалов курса).

Определение подзадач

При ленточном разбиении область расчетов делится на горизонтальные или вертикальные полосы (рис. 6.6а и 6.6б). Число полос определяется количеством параллельных процессов, размер полос обычно является одинаковым, узлы горизонтальных границ (первая и последняя строки) включаются в первую и последнюю полосы соответственно. Полосы для обработки распределяются между процессами.

Разделение строк и столбцов на полосы в большинстве случаев происходит на *непрерывной* (*последовательной*) основе и именно такой подход используется в данной лабораторной работе. При таком подходе , например, для горизонтального разбиения по строкам матрица A представляется в виде:

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

где $a_i = (a_{i1}, a_{i2}, \ldots a_{in})$, $0 \le i < m$, есть i-я строка матрицы A (предполагается, что количество строк m кратно числу процессоров p, т.е. $m = k \cdot p$).

Основной момент при организации вычислений с подобным разделением данных состоит в том, что на процессор, выполняющий обработку какой-либо полосы, должны быть продублированы граничные строки предшествующей и следующей полос вычислительной сетки. Продублированные граничные строки полос используются только при проведении расчетов, пересчет же этих строк происходит в полосах своего исходного месторасположения. Тем самым дублирование граничных строк должно осуществляться перед началом выполнения каждой очередной итерации метода сеток.

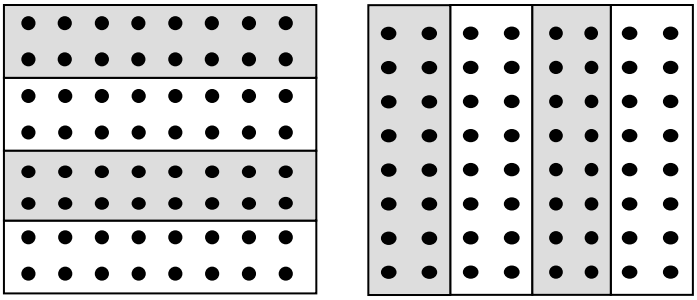

Рис. 6.6. Способы распределения элементов матрицы между процессорами вычислительной системы

В качестве начального варианта рассмотрим предельный случай, когда количество процессоров совпадает с числом внутренних строк сетки, т.е. $p=N$. В такой ситуации полоса каждого процессора состоит из трех строк, из которых только одна является перевычисляемой, а две других строки дублируются с соседних процессов. Примем далее все вычисления, связанные с обработкой каждой из таких полос, в качестве *базовой вычислительной подзадачи*.

Выделение информационных зависимостей

Параллельный вариант метода сеток при ленточном разделении данных состоит в обработке полос на всех имеющихся процессорах одновременно в соответствии со следующей схемой работы:

```
// Схема Гаусса-Зейделя, ленточное разделение данных
// действия, выполняемые на каждом процессоре
do {
  // <обмен граничных строк полос с соседями>
  // <обработка полосы>
  // <вычисление общей погрешности вычислений dmax>}
 while ( dmax > eps ); // eps - точность решения
```

Для конкретизации представленных в алгоритме действий введем обозначения:

- **ProcNum** – номер процессора, на котором выполняются описываемые действия,
- **PrevProc, NextProc** – номера соседних процессоров, содержащих предшествующую и следующую полосы,
- **NP** – количество процессоров,
- **M** – количество строк в полосе (без учета продублированных граничных строк),
- **N** – количество внутренних узлов в строке сетки (т.е. всего в строке N+2 узла).

Для нумерации строк полосы будем использовать нумерацию, при которой строки 0 и M+1 есть продублированные из соседних полос граничные строки, а строки собственной полосы процессора имеют номера от 1 до M.

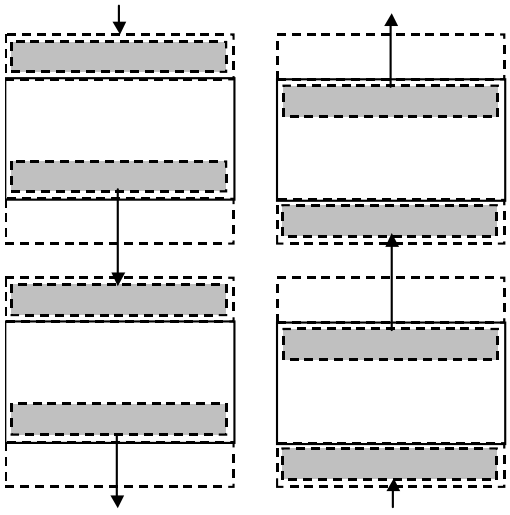

Рис. 6.7. Схема передачи граничных строк между соседними процессорами

Процедура обмена граничных строк между соседними процессорами может быть разделена на две последовательные операции, во время первой из которых каждый процессор передает свою нижнюю граничную строку следующему процессору и принимает такую же строку от предыдущего процессора (см. рис. 6.7). Вторая часть передачи строк выполняется в обратном направлении: процессоры передают свои верхние граничные строки своим предыдущим соседям и принимают переданные строки от следующих процессоров.

Выполнение подобных операций передачи данных в общем виде может быть представлено следующим образом (для операций передачи данных используется псевдокод, близкий к функциям MPI):

```
// передача нижней граничной строки следующему процессору
// и прием передаваемой строки от предыдущего процессора
Sendrecv(u[M][*],N+2,NextProc,u[0][*],N+2,PrevProc);
```

Реализация подобной объединенной функции *Sendrecv* обычно осуществляется таким образом, чтобы обеспечить и корректную работу на крайних процессорах, когда не нужно выполнять одну из операций передачи или приема, и организацию чередования процедур передачи на процессорах для ухода от тупиковых ситуаций, и возможности параллельного выполнения всех необходимых пересылок данных.

Для вычисления общей для всех процессоров погрешности вычислений может быть использована каскадная схема, для выполнения которой в MPI имеется функция *MPI_Allreduce*.

Общая схема вычислений на каждом процессоре может быть представлена на псевдокоде в следующем виде**:**

```
// Схема Гаусса-Зейделя, ленточное разделение данных
// действия, выполняемые на каждом процессоре
do {
  // обмен граничных строк полос с соседями
Sendrecv(u[M][*],N+2,NextProc,u[0][*],N+2,PrevProc);
Sendrecv(u[1][*],N+2,PrevProc,u[M+1][*],N+2,NextProc);
  // <обработка полосы с оценкой погрешности dm>
  // вычисление общей погрешности вычислений dmax
  Allreduce(dm,dmax,MAX,0);
} while ( dmax > eps ); // eps - точность решения
```

Масштабирование и распределение подзадач по процессорам

Как правило, число доступных процессоров p существенно меньше, чем число базовых подзадач N ($p << N$). 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.

Упражнение 4 – Реализация параллельного алгоритма Гаусса – Зейделя для решения задачи Дирихле

При выполнении этого упражнения Вам будет предложено разработать параллельный алгоритм Гаусса – Зейделя решения задачи Дирихле. При работе с этим упражнением Вы:

- Получите дополнительные навыки разработки параллельных программ, познакомитесь с вариантами разделения данных, коллективными операциями обмена данных,
- Подготовите первый вариант параллельной программы, реализующей алгоритм Гаусса- Зейделя решения задачи Дирихле.

Разрабатываемая параллельная программа, как и в предыдущих лабораторных работах, 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.

Задание 1 – Открытие проекта ParallelGaussSeidel

Откройте проект *ParallelGaussSeidel*, 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\ParallelGaussSeidel**,
- Дважды щелкните на файле **ParallelGaussSeidel.sln** или выделите имя этого файла и выполните команду **Open**.

После того, как Вы открыли проект, в окне Solution Explorer (Ctrl+Alt+L) дважды щелкните на файле исходного кода *ParallelGS.cpp*, как это показано на рисунке 6.8. После этих действий код, который вам предстоит модифицировать, будет открыт в рабочей области Visual Studio.

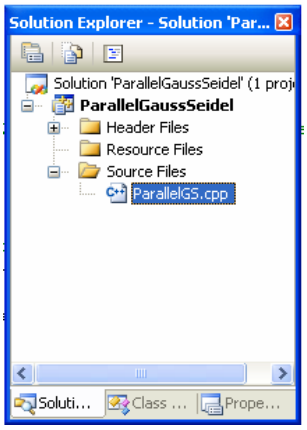

Рис. 6.8. Открытие файла ParallelGS.cpp с использованием Solution Explorer

В файле *ParallelGS.cpp* расположена главная функция (*main*) будущего параллельного приложения, которая содержит объявления необходимых переменных. Также в файле *ParallelGS.cpp* расположены функции, перенесенные сюда из проекта, содержащего последовательный алгоритм Гаусса – Зейделя: *DummyDataInitialization*, *ResultCalculation*, *PrintMatrix* (подробно о назначении этих функций рассказывается в упражнении 2 данной лабораторной работы). Эти функции можно будет использовать и в параллельной программе. Кроме того, в файле содержатся 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 сообщение: "`Parallel Gauss - Seidel program`".

Задание 2 – Инициализация и завершение параллельной программы

В текст параллельной программы необходимо добавить заголовочный файл MPI, чтобы использовать функции MPI в своем приложении. Добавьте выделенную строку в список подключаемых библиотек в файле исходного кода параллельной программы:

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

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:

```
static int ProcNum = 0;    // Number of available processes
static int ProcRank = -1;  // Rank of current process

void main(int argc, char* argv[]) {
  double* pMatrix;  // Matrix of the grid
  int     Size;     // Matrix size
```

```
  double  Eps;      // Requied accuracy
  double  Start, Finish, Duration;

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

  if(ProcRank == 0)
    printf("Parallel Gauss - Seidel 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. 6.9).

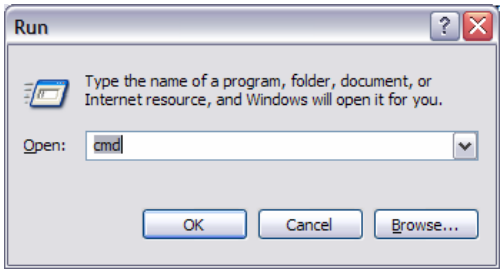

Рис. 6.9. Запуск Command Prompt

В командной строке перейдите в папку, где содержится исполняемый модуль разработанной программы (рис. 6.10):

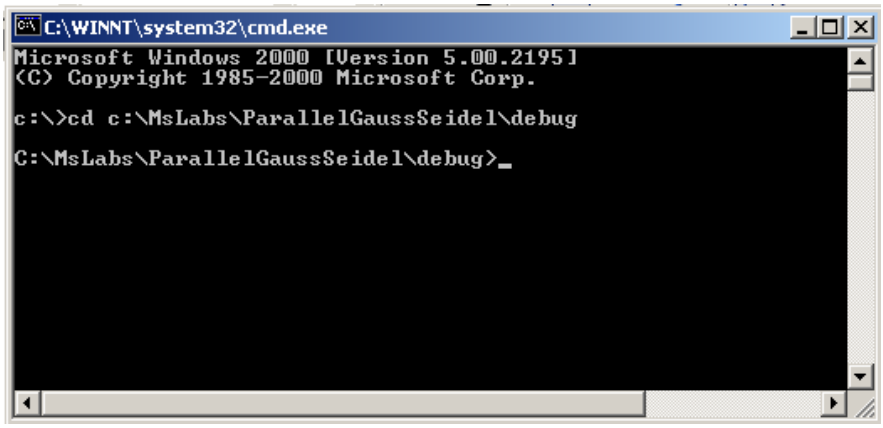

Рис. 6.10. Задание папки, в которой содержится исполняемый модуль параллельной программы

Для запуска программы, состоящей из 4 процессов, выполните команду (рис. 6.11):

```
mpiexec -n 4 ParallelGaussSeidel.exe
```

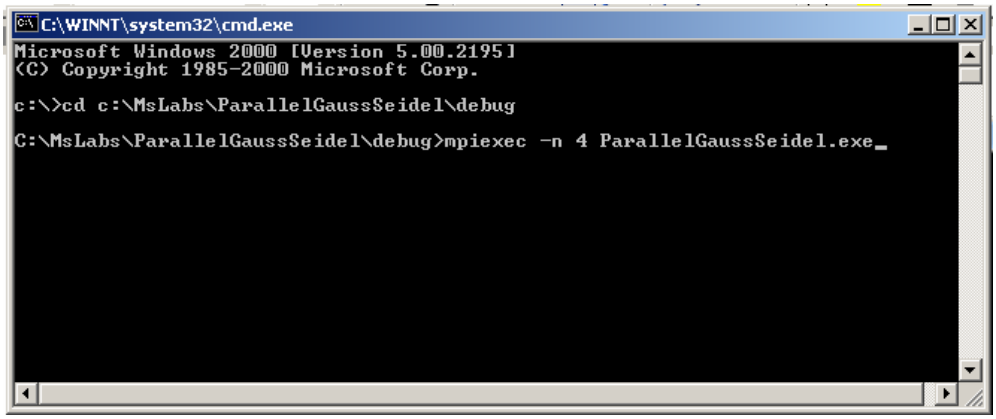

Рис. 6.11. Запуск параллельной программы

Убедитесь в том, что на командную консоль выводится сообщение

`"Parallel Gauss-Seidel algorithm".`

Задание 3 – Ввод исходных данных

На следующем этапе разработки параллельной программы необходимо задать размер используемой сетки, выделить память для хранения данных и выполнить задание начальных значений.

Организация диалога с пользователем для определения размера матрицы должен проводить только один процесс (пусть, как и ранее, этим процессом будет процесс с рангом 0).

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

```
// Function for memory allocation and initialization of grid nodes
void ProcessInitialization (double* &pMatrix, double* &pProcRows,
  int &Size, int &RowNum, double &Eps);
```

Сначала необходимо определить размер исходной матрицы, то есть задать значение переменной *Size*. Реализуем диалог с пользователям для ввода размера исходной матрицы. Как и в предыдущих лабораторных работах, реализуем проверку правильности вводимого значения. Добавьте выделенный фрагмент кода в тело функции *ProcessInitialization*:

```
// Function for memory allocation and and initialization of grid nodes
void ProcessInitialization (double* &pMatrix, double* &pProcRows,
  int &Size, int &RowNum, double &Eps) {
  if (ProcRank == 0) {
    do {
      printf("\nEnter the grid size: ");
      scanf("%d", &Size);
      if (Size <= 2) {
         printf("\nThe grid size must be greater than 2! \n");
      }
      if (Size < ProcNum) {
        printf("\n Grid size must be greater than"
               "the number of processes! \n ");
      }
      if ((Size-2)%ProcNum != 0) {
        printf("\n Number of inner rows of the grid must be divisible by"
               "the number of processes! \n");
      }
    } while ( (Size <= 2) || (Size < ProcNum) || ((Size-2)%ProcNum != 0));

    // Setting the required accuracy
    do {
      printf("\nEnter the required accuracy: ");
      scanf("%lf", &Eps);
      printf("\nChosen accuracy = %lf", Eps);
      if (Eps <= 0)
        printf("\nAccuracy must be greater than 0!\n");
    } while (Eps <= 0);
```

```
  }
}
```

После того, как значения переменных *Size* и *Eps* определены корректно, необходимо передать эти значения остальным процессам и определить количество строк сетки, располагаемых на каждом процессе. Для этого используем функцию широковещательной рассылки от одного процесса остальным *MPI_Bcast*. Добавьте следующий код в вашу программу; обратите внимание, что вызов *MPI_Bcast* должен выполняться всеми процессами:

```
  if (ProcRank == 0) {
    <…>
  }
  // Broadcasting the grid size
  MPI_Bcast(&Size, 1, MPI_INT,    0, MPI_COMM_WORLD);
  MPI_Bcast(&Eps,  1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  // Calculating the number of matrix rows stored on each process
  RowNum = (Size-2)/ProcNum + 2;
```

Добавьте вызов функции инициализации в функцию *main*. Скомпилируйте и запустите приложение. Убедитесь в том, что все ошибочные ситуации обрабатываются корректно. Для этого выполните несколько запусков приложения, задавая различное количество параллельных процессов (при помощи параметра запуска утилиты **mpiexec**) и разные значения вводимых данных.

Перейдем к следующему этапу - выделим память и зададим значения элементов матрицы. Определение начальных данных осуществляется процессом рангом 0. Далее, согласно схеме параллельных вычислений, изложенной в упражнении 3, исходная матрица распределяется между всеми процессами таким образом, чтобы каждый процесс обрабатывал непрерывную последовательность строк (горизонтальную полосу). Отметим, что первая версия разрабатываемой программы ориентирована на случай, когда размер сетки делится нацело на число процессов, то есть полосы матрицы на всех процессах содержат одно и то же количество строк. Для запоминания этого значения используем переменную *RowNum*. Адрес буфера памяти, где содержатся горизонтальная полоса строк на каждом из процессов, будем хранить в переменной *pProcRows* (*pProcRows* – матрица, которая содержит *RowNum* строк и *Size* столбцов и хранится построчно).

В основной функции программы объявим переменные:

```
void main(int argc, char* argv[]) {
  double* pMatrix;   // Matrix of the grid
  int     Size;      // Grid size
  double  Eps;       // Requied accuracy
  double* pProcRows; // Stripe of the matrix on current process
  int     RowNum;    // Number of rows in matrix stripe
  double  Start, Finish, Duration;
```

Выделим память для хранения данных и проинициализируем исходную матрицу на ведущем процессе. Для задания элементов на ведущем процессе используем функцию *DummyDataInitialization*.

Добавьте выделенный код в тело функции *ProcessInitialization*:

```
    <…>
  // Calculating the number of matrix rows stored on each process
  RowNum = (Size-2)/ProcNum + 2;

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

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

Для контроля правильности ввода исходных данных можно воспользоваться функцией *PrintMatrix. В* основной функции программы после вызова функции *ProcessInitialization* добавьте вызов функции *PrintMatrix* для матрицы *pMatrix* на нулевом процессе. Скомпилируйте и запустите приложение. Убедитесь в том, что данные задаются корректно.

Задание 4 – Завершение процесса вычислений

Для того, чтобы на каждом этапе разработки приложение было завершенным, разработаем функцию для корректной остановки процесса вычислений. Реализуем функцию *ProcessTermination,* которая освободит память, выделенную для хранения исходной матрицы *pMatrix* (только на ведущем процессе) и полосы матрицы *pProcRows*. Все эти массивы необходимо передать в функцию *ProcessTermination* в качестве аргументов:

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

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

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

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

Задание 5 – Распределение данных между процессами

На этом этапе необходимо выполнить разделение данных между процессам. В соответствии со схемой параллельных вычислений, изложенной в предыдущем упражнении, матрица должна быть разделена между процессами равными горизонтальными полосами. Реализуем функцию *DataDistribution* для выполнения подобного разделения данных между процессами. На вход этой функции в качестве аргументов необходимо передать исходную матрицу *pMatrix*, массив для хранения горизонтальной полосы матрицы *pProcRows*, а также размеры этих массивов (размер матрицы и вектора *Size* и число полос в горизонтальной полосе *RowNum*):

```
void DataDistribution(double* pMatrix, double* pProcRows, int Size,
  int RowNum);
```

Рассылку значений исходной матрицы *pMatrix* обеспечивается при помощи функции *MPI_Scatter*.

Добавьте в тело функции *DataDistribution* вызов функции *MPI_Scatter*:

```
// Function for distribution of the grid rows among the processes
void DataDistribution(double* pMatrix, double* pProcRows, int Size,
  int RowNum) {
  MPI_Status status;
  MPI_Scatter(pMatrix+Size, (RowNum-2)*Size, MPI_DOUBLE, pProcRows+Size,
    (RowNum-2)*Size, MPI_DOUBLE, 0,MPI_COMM_WORLD);

  // Copying the upper boundary row to the process 0
  if (ProcRank == 0 ){
    for(int i=0;i<Size;i++)
      pProcRows[i]=pMatrix[i];
  }
  // Sending the lower boundaty row to the process ProcNum-1
  if (ProcRank == 0)
    MPI_Send(pMatrix + Size * (Size-1), Size, MPI_DOUBLE,
      ProcNum - 1, 5, MPI_COMM_WORLD);
  if (ProcRank == ProcNum - 1)
    MPI_Recv(pProcRows + (RowNum - 1 ) * Size,
      Size, MPI_DOUBLE, 0, 5, MPI_COMM_WORLD, &status);
}
```

Вызов функции *DataDistribution* нужно поставить непосредственно после вызова функции инициализации вычислительного процесса *ProcessInitialization*:

```
  // Memory allocation and data initialization
  ProcessInitialization(pMatrix, pProcRows, Size, RowNum);
```

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

Для проверки правильности разделения данных между процессами после вызова функции *DataDistribution* распечатаем исходную матрицу, а затем полосы матрицы, содержащиеся на каждом из процессов. Добавим в код приложения еще одну функцию, которая служит для проверки правильности выполнения этапа распределения данных, и назовем ее *TestDistribution*.

Для того, чтобы организовать форматированный вывод матрицы, воспользуемся методам *PrintMatrix*:

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

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

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

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

Скомпилируйте приложение. Если в приложении обнаружились ошибки, исправьте их, сверяя свой код с кодом, представленным в данном упражнении. Запустите приложение с использованием трех процессов и установите размер данных 5. Убедитесь в том, что распределение данных выполняется правильно (рис. 6.12).

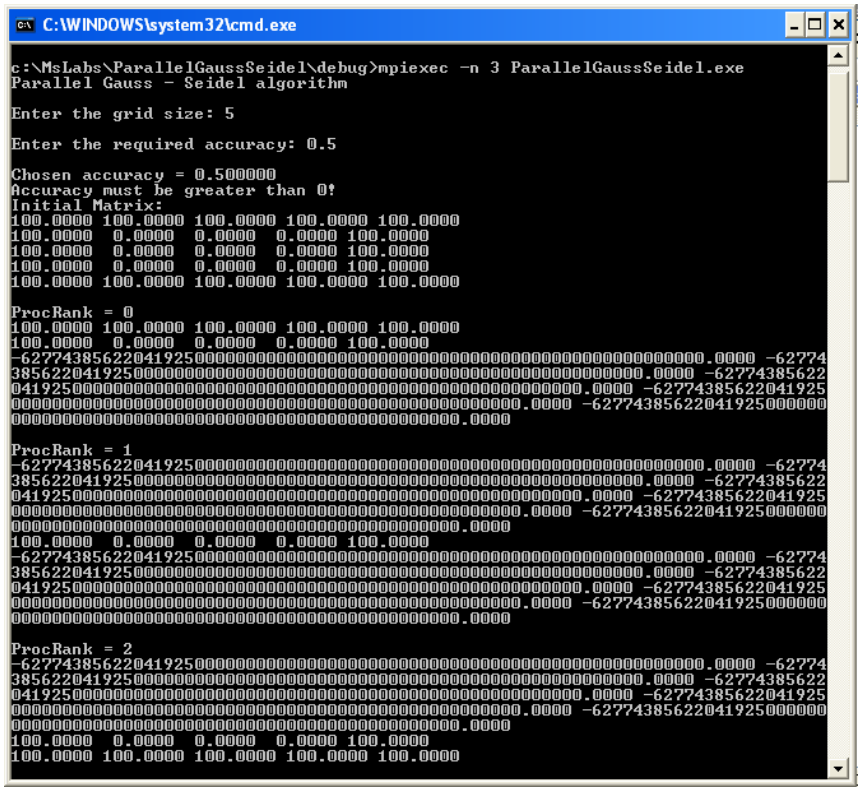

Рис. 6.12. Распределение данных в случае, когда приложение запускается на трех процессах, порядок матрицы равен 5

Следует отметить, что значения граничных строк в полосах процессов после начального распределения остались неинициализированными (кроме верхней граничной строки процесса 0 и нижней граничной строки процесса *ProcNum-1*). Как результат, при печати этих граничных строк могут быть выведены произвольные значения.

Задание 6 – Обмен граничных строк соседних процессов

Реализуем параллельный алгоритм Гаусса-Зейделя в функции *ParallelResultCalculation*. В качестве аргументов этой функции необходимо указать полосу матрицы *pProcRows*, размер строки исходной матрицы *Size*, количество строк полосы *RowNum*, заданную точность *Eps*, количество выполненных итераций *Iterations* (выходной параметр):

```
// Function for the parallel Gauss-Seidel method
ParallelResultCalculation(double *pProcRows, int Size, int RowNum,
  double Eps, int &Iterations);
```

Вызов функции *ParallelResultCalculation* следует выполнить сразу после распределения данных:

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

  // Parallel Gauss-Seidel method
  ParallelResultCalculation(pProcRows, Size, RowNum, Eps, Iterations);
  TestDistribution(pMatrix, pProcRows, Size, RowNum);
```

Отметим, что для проверки работоспособности алгоритма можно воспользоваться ранее разработанной функцией печати данных *TestDistribution*.

Первое действие, которое должно выполняться на каждой итерации параллельного алгоритма Гаусса – Зейделя, состоит в обмене граничных строк между соседними процессами (см. упражнение 3):

```
// Function for the parallel Gauss-Seidel method
void ParallelResultCalculation(double* pProcRows, int Size, int RowNum,
  double Eps, int &Iterations) {
  double ProcDelta, Delta;
  Iterations=0;
```

```
//  do {
    Iterations++;
    // Exchanging the boundary rows of the process stripe
    ExchangeData(pProcRows, Size, RowNum);
//  } while(Iteration < 2);
}
```

(отметим, что в качестве предварительного варианта максимальное количество итераций алгоритма установлено равным 2).

В качестве аргументов функции обмена строк *ExchangeData* указывается полоса матрица *pProcRows*, размер строки *Size* и количество строк полосы *RowNum*. Процедура обмена граничных строк между соседними процессорами в наиболее простом виде может быть реализована с использованием функции *MPI_Sendrecv*:

```
// Function for exchanging the boundary rows of the process stripes
void ExchangeData(double* pProcRows, int Size, int RowNum) {
  MPI_Status status;
  int NextProcNum = (ProcRank == ProcNum-1)? MPI_PROC_NULL : ProcRank+1;
  int PrevProcNum = (ProcRank == 0)         ? MPI_PROC_NULL : ProcRank-1;
  // Send to NextProcNum and receive from PrevProcNum
  MPI_Sendrecv(pProcRows+Size*(RowNum-2), Size, MPI_DOUBLE,NextProcNum, 4,
    pProcRows, Size, MPI_DOUBLE, PrevProcNum, 4, MPI_COMM_WORLD, &status);
  // Send to PrevProcNum and receive from NextProcNum
  MPI_Sendrecv(pProcRows + Size, Size, MPI_DOUBLE, PrevProcNum, 5,
    pProcRows + (RowNum-1)*Size, Size, MPI_DOUBLE, NextProcNum, 5,
    MPI_COMM_WORLD, &status);
}
```

Скомпилируйте приложение. Если в приложении обнаружились ошибки, исправьте их, сверяя свой код с кодом, представленным в данном упражнении. Запустите приложение с использованием 3 параллельных процессов и установите размер сетки равный 6. Убедитесь в том, что обмен граничных строк выполняется правильно (рис. 6.13).

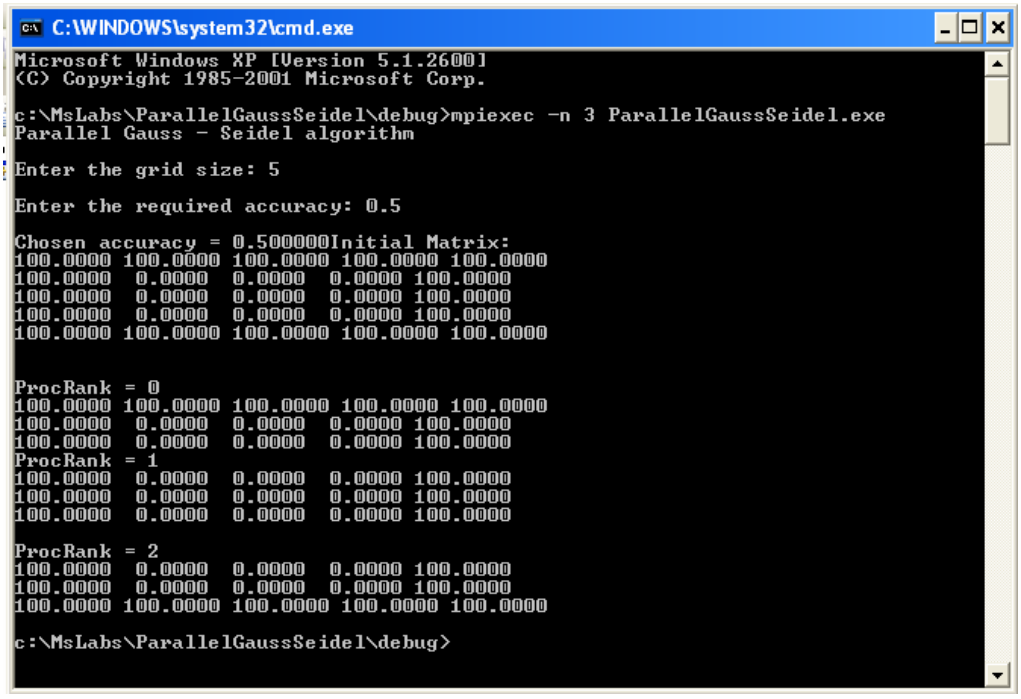

Рис. 6.13. Распределение данных после обмена граничных строк в случае, когда приложение запускается на трех процессах, порядок матрицы равен 5

Отметим, что теперь после обмена граничных строк значения этих строк являются установленными и печатаются правильно.

Задание 7 – Выполнение итераций параллельного алгоритма

Дополним реализацию функции *ParallelResultCalculation* функцией *IterationCalculation* для выполнения итераций параллельного алгоритма Гаусса-Зейделя для решения задачи Дирихле. В качестве параметров функции следует передать полосу исходной матрицы *pProcRows*, размер строки *Size* и количество строк полосы *RowNum*:

```
// Function for the execution of the Gauss-Seidel method iteration
  double IterationCalculation(double* pProcRows, int Size, int RowNum);
```

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

```
// Function for the execution of the Gauss-Seidel method iteration
double IterationCalculation(double* pProcRows, int Size, int RowNum) {
  int i, j;  // Loop variables
  double dm, dmax, temp;
  dmax = 0;
  for (i = 1; i < RowNum-1; i++)
    for(j = 1; j < Size-1; j++) {
      temp = pProcRows[Size * i + j];
      pProcRows[Size * i + j] = 0.25 * (pProcRows[Size * i + j + 1] +
                                        pProcRows[Size * i + j - 1] +
                                        pProcRows[Size * (i + 1) + j] +
                                        pProcRows[Size * (i - 1) + j]);
      dm = fabs(pProcRows[Size * i + j] - temp);
      if (dmax < dm) dmax = dm;
  }
  return dmax;
}
```

Добавьте вызов функции *IterationCalculation* в функцию *ParallelResultCalculation*:

```
// Function for the parallel Gauss-Seidel method
void ParallelResultCalculation(double* pProcRows, int Size, int RowNum,
  double Eps, int &Iterations) {
    <…>
//  do {
    Iterations++;
    // Exchanging the boundary rows of the process stripe
    ExchangeData(pProcRows, Size, RowNum);

    // The Gauss-Seidel method iteration
    ProcDelta = IterationCalculation(pProcRows, Size, RowNum);
    TestDistribution(pMatrix, pProcRows, Size, RowNum);
//  } while(Iteration < 2);
}
```

Как можно заметить из приведенного программного кода, вызов функции печати *TestDistribution* следует переместить на строку после вызова вновь разработанной функции *IterationCalculation*.

Снова скомпилируйте приложение. Выполните эксперименты и убедитесь, что вычисления выполняются правильно. Следует отметить, что получаемые результаты могут отличаться от результатов последовательного алгоритма и единственный способ контроля состоит в "ручной" проверке вычислений (для более простой проверки результатов оператор цикла можно закомментировать).

При запуске приложения с использованием трех процессов и размере сетки равном 5 результаты вычислений должны совпадать с данными рис. 6.14. Для продолжения проверки раскоментируйте оператор цикла в функции *ParallelResultCalculation* (изменяя константу в условии завершения цикла можно устанавливать необходимое число выполняемых итераций параллельного метода Гаусса-Зейделя).

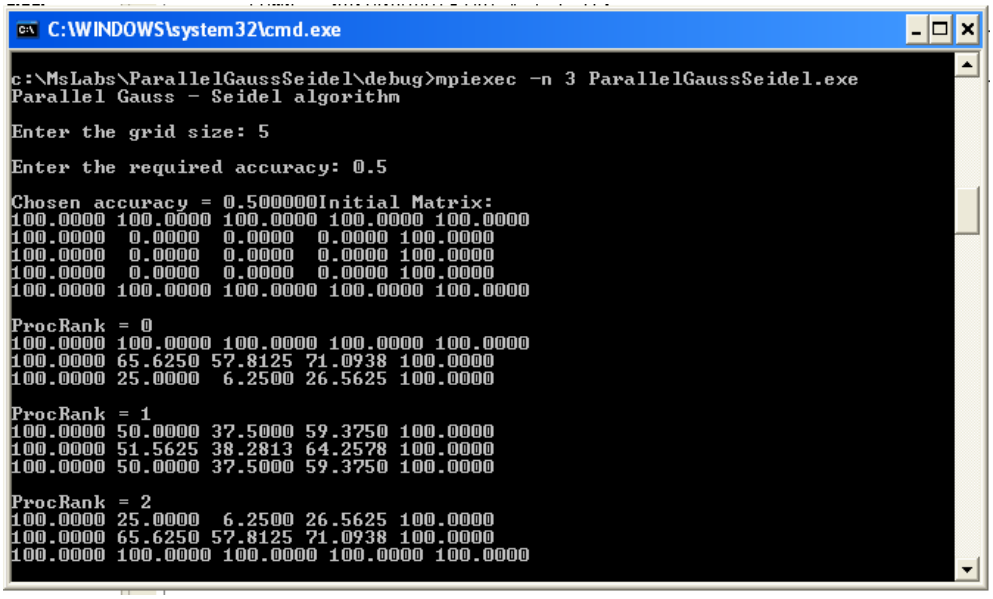

Рис. 6.14. Результаты вычислений в случае, когда приложение запускается на трех процессах, порядок матрицы равен 5.

Задание 8 – Нахождение максимального погрешности

Для полной реализации метода Гаусса-Зейделя осталось добавить вычисление максимального изменения результатов расчетов, получаемых на итерации алгоритма. Необходимые изменения функции *ParallelResultCalculation* состоят в следующем (для выделения добавляемого кода использован полужирный шрифт):

```
// Function for the parallel Gauss-Seidel method
void ParallelResultCalculation(double* pProcRows, int Size, int RowNum,
  double Eps, int &Iterations) {
    <…>
  do {
    <…>
    // Calculating the maximum value of the deviation
    MPI_Allreduce(&ProcDelta, &Delta, 1, MPI_DOUBLE, MPI_MAX,
      MPI_COMM_WORLD);
  } while( Delta > Eps );
}
```

Обратите внимание на изменение условия завершения цикла повторения итераций метода Гаусса-Зейделя. Для уменьшения объема отладочного вывода удалите вызов функции *TestDistribution*.

Скомпилируйте приложение и проверьте правильность выполняемых вычислений. Так, например, если параллельное приложение запускается с использованием 3 параллельных процессов, размер сетки равен 5 и требуемая точность равна 0.1, то процессы должны получить результаты, приведенные на рис. 6.15.

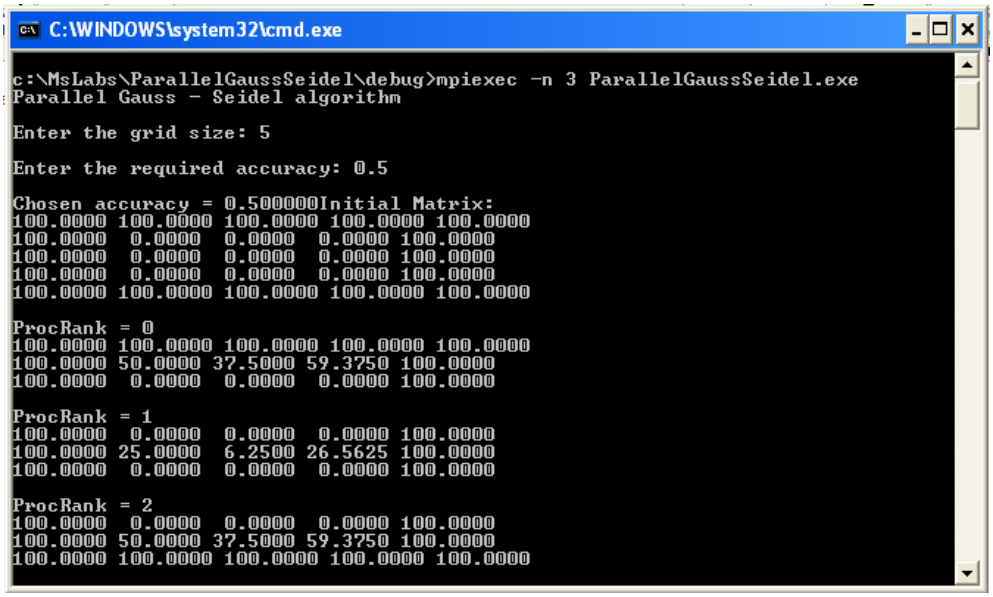

Рис. 6.15. Результат вычислений алгоритма Гаусса-Зейделя, в случае, когда для приложения используется три процесса, и размер сетки равен пяти

Задание 9 – Сбор результатов вычислений

После завершения метода Гаусса-Зейделя необходимо собрать полосы сетки, расположенных на разных процессах, на ведущей процессе (на процессе с рангом 0). Используем функцию *MPI_Gather*, которая собирает из блоков, расположенных на разных процессах коммуникатора, единый массив.

Функция сбора результатов *ResultCollection* будет состоять только из вызова функции *MPI_Gather*:

```
// Function for gathering the calculation results
void ResultCollection(double* pMatrix, double* pProcRows,int Size,
  int RowNum) {
  MPI_Gather(pProcRows+Size, (RowNum-2)*Size, MPI_DOUBLE, pMatrix+Size,
    (RowNum-2)*Size, MPI_DOUBLE, 0,MPI_COMM_WORLD);
}
```

Включите вызов функции *ResultCollection* в основную программу *main*:

```
  // Parallel Gauss-Seidel method
  ParallelResultCalculation(pProcRows, Size, RowNum, Eps, Iterations);
  // TestDistribution(pMatrix, pProcRows, Size, RowNum);

  // Gathering the calculation results
  ResultCollection(pMatrix, pProcRows, Size, RowNum);
  TestDistribution(pMatrix, pProcRows, Size, RowNum);
```

Скомпилируйте и запустите приложение. Оцените правильность его работы – как и ранее, используйте для проверки функции печати *TestDistribution*.

Задание 10 – Проверка правильности работы программы

Теперь, после выполнения функции сбора, необходимо проверить правильность выполнения алгоритма. Для этого разработаем функцию *TestResult*, которая сравнит результаты последовательного и параллельного алгоритмов. Для выполнения последовательного алгоритма можно использовать функцию *SerialResultCalculation*, разработанную в упражнении 2.

Для того чтобы функция *SerialResultCalculation* использовала те же исходные данные, что и функция *ParallelResultCalculation,* необходимо сделать копию этих данных, используя функцию *CopyData*:

```
// Function to copy the initial data
void CopyData(double *pMatrix, int Size, double *pSerialMatrix) {
  copy(pMatrix, pMatrix + Size, pSerialMatrix);
}
```

Добавим вызов этой функции в исходный код. В функции *main* необходимо объявить переменную, предназначенную для хранения копии данных, используемой в последовательном алгоритме, а также создать саму эту копию:

```
...
  double* pMatrix;       // Matrix of the grid
  double* pProcRows;     // Stripe of the matrix on current process
  double* pSerialMatrix; // Result of the serial method

  <…>

  // Creating the copy of the initial data
  if (ProcRank == 0) {
    pMarixCopy = new double[Size*Size];
    CopyData(pMatrix, Size, pSerialMatrix);
  }
```

При завершении работы приложения необходимо освободить выделенную память для последовательного алгоритма:

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

Функция *TestResult* должна иметь доступ к исходным матрицам *pMatrix* и *pCMatrix* и должна быть выполнена только на ведущем процессе:

```
// Function for testing the computation result
void TestResult(double* pMatrix, double* pSerialMatrix, int Size,
  double Eps) {
  int equal = 0;  // =1, if the matrices are not equal
  int Iter;

  if (ProcRank == 0) {
    SerialResultCalculation(pSerialMatrix, Size, Eps, Iter);
    for (int i=0; i<Size*Size; i++) {
      if (fabs(pSerialMatrix[i]-pMatrix[i]) >= Eps) {
        equal = 1; break;
      }
    }
    if (equal == 1)
      printf("The results of serial and parallel algorithms"
        "are NOT identical. Check your code.");
    else
      printf("The results of serial and parallel algorithms"
        "are identical.");      }
}
```

Результатом работы этой функции является печать диагностического сообщения. Используя эту функцию, можно проверять результат работы параллельного алгоритма при любых значениях исходных данных.

Следует отметить, что, в общем случае, результаты последовательного и параллельного вариантов метода Гаусса - Зейделя могут отличаться, так как в них последовательности обработки узлов сетки могут быть разными. Однако различия получаемых результатов расчетов должны находиться в пределах заданной точности вычислений (см. раздел 12 "Параллельные методы решения дифференциальных уравнений в частных производных").

Закомментируйте вызовы функции отладочной печати *TestDistribution*, которая ранее использовалась для контроля правильности выполнения этапов разработки параллельного приложения. Вместо функции *DummyDataInitialization*, которая генерирует начальные данные простого вида, используйте функцию *RandomDataInitialization*, которая генерирует начальные значения во внутренних узлах сетки при помощи датчика случайных чисел:

```
// Function for setting the grid node values by a random generator
void RandowmDataInitialization (double* pMatrix, int Size) {
```

```
  int i, j;  // Loop variables
  srand(unsigned(clock()));
  // Setting the grid node values
  for (i=0; i<Size; i++) {
    for (j=0; j<Size; j++)
      if ((i==0) || (i== Size-1) || (j==0) || (j==Size-1))
        pMatrix[i*Size+j] = 100;
      else
        pMatrix[i*Size+j] = rand()/double(1000);
  }
}
```

Скомпилируйте и запустите приложение. Задавайте различные размеры сетки и значения требуемой точности вычислений. Убедитесь в том, что приложение работает правильно.

Задание 11 – Реализация вычислений для любых размеров сетки

Параллельное приложение, которое разрабатывалось в ходе выполнения предыдущих заданий, было ориентировано на случай, когда количество внутренних узлов сетки *(Size-2)* нацело делится на число процессоров *ProcNum*. В этом случае матрица делится между процессами на равные полосы и число строк *RowNum*, которые обрабатывает процесс, для всех процессов было одним и тем же.

Теперь рассмотрим случай, когда количество внутренних узлов сетки *(Size-2)* не кратно числу процессов *ProcNum*. В этом случае, число строк в полосе на каждом процессе будет свое: некоторые процессы получат $\lfloor (Size-2)/ProcNum \rfloor + 2$, а остальные - $\lceil (Size-2)/ProcNum \rceil + 2$ строк матрицы (операция $\lfloor\ \rfloor$ означает округление значения до ближайшего меньшего целого числа, операция $\lceil\ \rceil$ – округление до ближайшего большего целого числа).

В функции *ProcessInitialization* уберем обработку ошибочной ситуации, которая возникает в случае, когда количество внутренних узлов сетки не делится нацело на число процессов. Используем 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 $\lfloor (Size-2)/ProcNum + 2 \rfloor$ строк (результат операции $\lfloor\ \rfloor$ совпадает с результатом целочисленного деления). После выполнения этой операции остается распределить $Size - \lfloor (Size-2)/ProcNum \rfloor - 2$ 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. $\lfloor (RestRows-2)/(ProcNum-i) + 2 \rfloor$ строк.

Изменим определение значения переменной *RowNum* в функции *ProcessInitialization*:

```
// Function for memory allocation and initialization of the grid nodes
void ProcessInitialization (double* &pMatrix, double* &pProcRows,int &Size,
  int &RowNum, double &Eps) {
  int RestRows; // Number of rows, that haven't been distributed yet
  <…>
  // Define the number of matrix rows stored on each process
  RestRows = Size - 2;
  for (i=0; i<ProcRank; i++)
    RestRows = RestRows - RestRows / ( ProcNum – i );
  RowNum = RestRows/ ( ProcNum – ProcRank )+2;
  <…>
}
```

В случае, когда матрица распределяется между процессами не поровну, для распределения данных нельзя использовать функцию *MPI_Scatter*. Вместо нее следует использовать более общую функцию *MPI_Scatterv*, которая дает возможность одному процессу распределить данные непрерывными блоками элементов разного размера между всеми процессами коммуникатора.

Чтобы вызвать функцию *MPI_Scatterv*, необходимо определить два вспомогательных массива для задания смещения и размеров передаваемых блоков. Внесем необходимые изменения в код функции *DataDistribution*:

```
// Function for distribution of the grid rows among the processes
void DataDistribution(double* pMatrix, double* pProcRows, int RowNum,
  int Size) {
```

```
  int *pSendNum; // Number of elements sent to the process
  int *pSendInd; // Index of the first data element sent to the process
  int RestRows = Size;
  // Alloc memory for temporary objects
  pSendInd = new int [ProcNum];
  pSendNum = new int [ProcNum];
  // Define the disposition of the matrix rows for current process
  RowNum = ( Size – 2 ) / ProcNum + 2;
  pSendNum[0] = RowNum * Size;
  pSendInd[0] = 0;
  for (int i=1; i < ProcNum; i++) {
    RestRows = RestRows - RowNum + 2;
    RowNum =  ( RestRows – 2 ) / ( ProcNum – i ) + 2;
    pSendNum[i] = RowNum * Size;
    pSendInd[i] = pSendInd[i-1] + pSendNum[i-1] - Size;
  }
  // Scatter the rows
  MPI_Scatterv(pMatrix , pSendNum, pSendInd, MPI_DOUBLE, pProcRows,
    pSendNum[ProcRank], MPI_DOUBLE, 0, MPI_COMM_WORLD);
  delete []pSendInd;
  delete []pSendNum;
}
```

Аналогично для сбора данных вместо функции *MPI_Gather*, ориентированной на сбор данных одинакового объема со всех процессов коммуникатора, будем использовать более общую функцию *MPI_Allgatherv*. Как и при использовании *MPI_Scatterv*, использование *MPI_Allgatherv* требует использования двух дополнительных массивов:

```
// Function for gathering the result vector
void ResultCollection(double *pMatrix, double* pProcResult,
  int Size, int RowNum) {
  int *pReceiveNum;  // Number of elements, that current process sends
  int *pReceiveInd;  // Index of the first element of the received block
  int RestRows = Size;
  int i;             // Loop variable

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

  // Define the disposition of the result vector block of current processor
  pReceiveInd[0] = 0;
  RowNum = ( Size – 2 ) / ProcNum + 2;
  pReceiveNum[0] = RowNum * Size;
  for ( i=1; i < ProcNum; i++){
    RestRows = RestRows - RowNum + 2;
    RowNum = ( RestRows – 2 ) / ( ProcNum – i ) + 2;
    pReceiveNum[i] = RowNum * Size;
    pReceiveInd[i] = pReceiveInd[i-1] + pReceiveNum[i-1] - Size;
  }
  // Gather the whole result vector on every processor
  MPI_Allgatherv(pProcRows, pReceiveNum[ProcRank], MPI_DOUBLE, pMatrix,
    pReceiveNum, pReceiveInd, MPI_DOUBLE, MPI_COMM_WORLD);

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

Скомпилируйте и запустите приложение. Проверьте правильность выполнения параллельного варианта алгоритма Гаусса-Зейделя при помощи функции *TestDistribution*.

Задание 12 – Проведение вычислительных экспериментов

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

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

```
// Memory allocation and data initialization
ProcessInitialization (pMatrix,pProcRows,Size,RowNum,Eps);
Start = MPI_Wtime();
// Data distribution among the processes
DataDistribution(pMatrix, pProcRows, Size,RowNum);

// Paralle Gauss-Seidel method
ParallelResultCalculation(pProcRows, Size,RowNum,Eps, Iterations);

// Gathering the calculation results
ResultCollection(pProcRows, pMatrix, Size, RowNum);
Finish = MPI_Wtime();
Duration = Finish-Start;

<…>
```

Очевидно, что таким образом будет определено то время, которое было затрачено на выполнение вычислений нулевым процессом. Возможно, что время выполнения алгоритма другими процессами немного от него отличается. Но поскольку на этапе разработки параллельного алгоритма особое внимание уделялось равномерной загрузке (*балансировке*) процессов, то теперь есть все основания полагать, что время выполнения алгоритма другими процессами несущественно отличается от приведенного.

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

Таблица 6.3. Результаты вычислительных экспериментов для параллельного алгоритма Гаусса-Зейделя

Размер сетки	Последовательный алгоритм	Параллельный алгоритм					
		2 процессора		4 процессора		8 процессоров	
		Время	Ускорение	Время	Ускорение	Время	Ускорение
10							
100							
1000							
2000							
3000							
4000							
5000							
6000							
7000							
8000							
9000							
10000							

В графу “Последовательный алгоритм” внесите время выполнения последовательного алгоритма, полученное при проведении тестирования последовательного приложения в упражнении 2. Для того, чтобы вычислить ускорение, разделите время выполнения последовательного алгоритма на время выполнения параллельного алгоритма. Результат поместите в соответствующую графу таблицы.

В отличие от ранее выполненных лабораторных работ проведите самостоятельно теоретическую оценку времени выполнения параллельного алгоритма Гаусса-Зейделя. Полученные оценки внесите в таблицу 6.4 и сравните с реальным временем выполнения экспериментов.

Таблица 6.4. Ускорение вычислений, получаемое для параллельного алгоритма Гаусса-Зейделя

Размер матрицы	2 процессора		4 процессора		8 процессоров	
	Модель	Эксперимент	Модель	Эксперимент	Модель	Эксперимент
10						
100						
1000						
2000						
3000						
4000						
5000						
6000						
7000						
8000						
9000						
10000						

Контрольные вопросы

- Насколько хорошо совпадают время, полученное теоретически, и реальное время выполнения алгоритма? В чем может состоять причина несовпадений?

Задания для самостоятельной работы

1. Изучите параллельный алгоритм Гаусса-Зейделя решения задачи Дирихле, основанный на ленточном вертикальном разделении матрицы. Напишите программу, реализующую этот алгоритм.

2. Изучите параллельный алгоритм Гаусса-Зейделя решения задачи Дирихле, основанный на блочном разделении матрицы. Напишите программу, реализующую этот алгоритм.

Приложение 1. Программный код последовательного алгоритма Гаусса-Зейделя

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

// Function for the Gauss-Seidel algoritm
void ResultCalculation(double* pMatrix, int Size, double &Eps,
  int &Iterations) {
  double dm, dmax,temp;
  int i, j;  // Loop variables
  Iterations = 0;
  do {
    dmax = 0;
    for (i = 1; i < Size - 1; i++)
      for(j = 1; j < Size - 1; j++) {
        temp = pMatrix[Size * i + j];
        pMatrix[Size * i + j] = 0.25 * (pMatrix[Size * i + j + 1] +
                                        pMatrix[Size * i + j - 1] +
                                        pMatrix[Size * (i + 1) + j] +
                                        pMatrix[Size * (i - 1) + j]);
        dm = fabs(pMatrix[Size * i + j] - temp);
        if (dmax < dm) dmax = dm;
```

```
      }
      Iterations++;
  }
  while (dmax > Eps);
}

// Function for computational process termination
void ProcessTermination(double* pMatrix) {
  delete [] pMatrix;
}
// Function for formatted matrix output
void PrintMatrix (double* pMatrix, int RowCount, int ColCount) {
  int i, j; // Loop variables
  for (i=0; i<RowCount; i++) {
    for (j=0; j<ColCount; j++)
      printf("%7.4f ", pMatrix[i*ColCount+j]);
    printf("\n");
  }
}
// Function for simple setting the grid node values
void DummyDataInitialization (double* pMatrix, int Size) {
  int i, j;  // Loop variables
  double h = 1.0 / (Size - 1);
  // Setting the grid node values
  for (i=0; i<Size; i++) {
    for (j=0; j<Size; j++)
      if ((i==0) || (i== Size-1) || (j==0) || (j==Size-1))
        pMatrix[i*Size+j] = 100;
      else
        pMatrix[i*Size+j] = 0;
  }
}
// Function for memory allocation and initialization of grid nodes
void ProcessInitialization (double* &pMatrix, int &Size, double &Eps) {
  // Setting the grid size
  do {
    printf("\nEnter the grid size: ");
    scanf("%d", &Size);
    printf("\nChosen grid size = %d", Size);
    if (Size <= 2)
      printf("\nSize of grid must be greater than 2!\n");
  } while (Size <= 2);
  // Setting the required accuracy
  do {
    printf("\nEnter the required accuracy: ");
    scanf("%lf", &Eps);
    printf("\nChosen accuracy = %lf", Eps);
    if (Eps <= 2)
      printf("\nAccuracy must be greater than 0!\n");
  }   while (Eps <= 0);

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

  // Setting the grid node values
  DummyDataInitialization(pMatrix, Size);
}

void main() {
  double* pMatrix;    // Matrix of the grid nodes
  int     Size;       // Matrix size
  double  Eps;        // Required accuracy
  int     Iterations; // Iteration number
```

```
  printf ("Serial Gauss - Seidel algorithm\n");

  // Process initialization
  ProcessInitialization(pMatrix, Size, Eps);
  // Matrix output
  printf ("Initial Matrix: \n");
  PrintMatrix (pMatrix, Size, Size);

  // The Gauss-Seidel method
  ResultCalculation(pMatrix, Size, Eps, Iterations);

  // Printing the result
  printf("\n Number of iterations: %d\n",Iterations);
  printf ("\n Result matrix: \n");
  PrintMatrix (pMatrix, Size, Size);
  getch();

  // Computational process termination
  ProcessTermination(pMatrix);
}
```

Приложение 2 – Программный код параллельного приложения для алгоритма Гаусса-Зейделя

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


static int ProcNum = 0;   // Number of available processes
static int ProcRank = -1; // Rank of current process

// Function for distribution of the grid rows among the processes
void DataDistribution(double* pMatrix, double* pProcRows, int RowNum,
  int Size) {
  int *pSendNum; // Number of elements sent to the process
  int *pSendInd; // Index of the first data element sent to the process
  int RestRows=Size;
  // Alloc memory for temporary objects
  pSendInd = new int [ProcNum];
  pSendNum = new int [ProcNum];
  // Define the disposition of the matrix rows for current process
  RowNum = (Size-2)/ProcNum+2;
  pSendNum[0] = RowNum*Size;
  pSendInd[0] = 0;
  for (int i=1; i<ProcNum; i++) {
    RestRows = RestRows - RowNum + 2;
    RowNum = (RestRows-2)/(ProcNum-i)+2;
    pSendNum[i] = (RowNum)*Size;
    pSendInd[i] = pSendInd[i-1]+pSendNum[i-1]-Size;
  }
  // Scatter the rows
  MPI_Scatterv(pMatrix , pSendNum, pSendInd, MPI_DOUBLE, pProcRows,
    pSendNum[ProcRank], MPI_DOUBLE, 0, MPI_COMM_WORLD);
  delete []pSendInd;
  delete []pSendNum;
}
```

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

// Function for formatted matrix output
void PrintMatrix(double *pMatrix, int RowCount, int ColCount){
  int i,j; // Loop variables
  for(int i=0; i < RowCount; i++) {
    for(j=0; j < ColCount; j++)
      printf("%7.4f ", pMatrix[i*ColCount+j]);
    printf("\n");
  }
}
// Function for the execution of the Gauss-Seidel method iteration
double IterationCalculation(double* pProcRows, int Size, int RowNum) {
  int i, j;  // Loop variables
  double dm, dmax,temp;
  dmax = 0;
  for (i = 1; i < RowNum-1; i++)
    for(j = 1; j < Size-1; j++) {
      temp = pProcRows[Size * i + j];
      pProcRows[Size * i + j] = 0.25 * (pProcRows[Size * i + j + 1] +
                                        pProcRows[Size * i + j - 1] +
                                        pProcRows[Size * (i + 1) + j] +
                                        pProcRows[Size * (i - 1) + j]);
      dm = fabs(pProcRows[Size * i + j] - temp);
      if (dmax < dm) dmax = dm;
  }
  return dmax;
}

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

// Function for simple setting the grid node values
void DummyDataInitialization (double* pMatrix, int Size) {
  int i, j;  // Loop variables
  double h = 1.0 / (Size - 1);
  // Setting the grid node values
  for (i=0; i<Size; i++) {
    for (j=0; j<Size; j++)
      if ((i==0) || (i== Size-1) || (j==0) || (j==Size-1))
        pMatrix[i*Size+j] = 100;
      else
        pMatrix[i*Size+j] = 0;
```

```
  }
}

// Function for memory allocation and initialization of grid nodes
void ProcessInitialization (double* &pMatrix, double* &pProcRows,int &Size,
  int &RowNum, double &Eps) {
  int RestRows; // Number of rows, that haven't been distributed yet
  // Setting the grid size
  if (ProcRank == 0) {
    do {
      printf("\nEnter the grid size: ");
      scanf("%d", &Size);
      if (Size <= 2) {
         printf("\n Size of grid must be greater than 2! \n");
      }
      if (Size < ProcNum) {
        printf("Size of grid must be greater than"
               "the number of processes! \n ");
      }
    }
    while ( (Size <= 2) || (Size < ProcNum));

    // Setting the required accuracy
    do {
      printf("\nEnter the required accuracy: ");
      scanf("%lf", &Eps);
      printf("\nChosen accuracy = %lf", Eps);
      if (Eps <= 0)
        printf("\nAccuracy must be greater than 0!\n");
    }
    while (Eps <= 0);
  }
  MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);
  MPI_Bcast(&Eps, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

  // Define the number of matrix rows stored on each process
  RestRows = Size;
  for (i=0; i<ProcRank; i++)
    RestRows = RestRows-RestRows/(ProcNum-i);
  RowNum = (RestRows-2)/(ProcNum-ProcRank)+2

  // Memory allocation
  pProcRows = new double [RowNum*Size];
  // Define the values of initial objects' elements
  if (ProcRank == 0) {
    // Initial matrix exists only on the pivot process
    pMatrix = new double [Size*Size];
    // Values of elements are defined only on the pivot process
    DummyDataInitialization(pMatrix, Size);
  }
}
// Function for exchanging the boundary rows of the process stripes
void ExchangeData(double* pProcRows, int Size, int RowNum) {
  MPI_Status status;
  int NextProcNum = (ProcRank == ProcNum-1)? MPI_PROC_NULL : ProcRank + 1;
  int PrevProcNum    = (ProcRank == 0)? MPI_PROC_NULL : ProcRank - 1;
  // Send to NextProcNum and receive from PrevProcNum
  MPI_Sendrecv(pProcRows + Size * (RowNum - 2),Size, MPI_DOUBLE,
    NextProcNum, 4, pProcRows, Size, MPI_DOUBLE, PrevProcNum, 4,
    MPI_COMM_WORLD, &status);
  // Send to PrevProcNum and receive from NextProcNum
  MPI_Sendrecv(pProcRows + Size, Size, MPI_DOUBLE, PrevProcNum, 5,
    pProcRows + (RowNum - 1) * Size, Size,MPI_DOUBLE, NextProcNum, 5,
```

```
    MPI_COMM_WORLD, &status);
}

// Function for the parallel Gauss - Seidel method
void ParallelResultCalculation (double *pProcRows, int Size, int RowNum,
  double Eps, int &Iterations) {
  double ProcDelta,Delta;
  Iterations=0;
  do {
    Iterations++;
    // Exchanging the boundary rows of the process stripe
    ExchangeData(pProcRows, Size,RowNum);

    // The Gauss-Seidel method iteration
    ProcDelta = IterationCalculation(pProcRows, Size, RowNum);

    // Calculating the maximum value of the deviation
    MPI_Allreduce(&ProcDelta, &Delta, 1,MPI_DOUBLE, MPI_MAX,
      MPI_COMM_WORLD);
  } while ( Delta > Eps);
}
// Function for gathering the result vector
void ResultCollection(double *pMatrix, double* pProcResult, int Size,
  int RowNum) {
  int *pReceiveNum;  // Number of elements, that current process sends
  int *pReceiveInd;  // Index of the first element of the received block
  int RestRows = Size;
  int i;             // Loop variable

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

  // Define the disposition of the result vector block of current processor
  pReceiveInd[0] = 0;
  RowNum = (Size-2)/ProcNum+2;
  pReceiveNum[0] = RowNum*Size;
  for ( i=1; i < ProcNum; i++){
    RestRows = RestRows - RowNum + 1;
    RowNum = (RestRows-2)/(ProcNum-i)+2;
    pReceiveNum[i] = RowNum*Size;
    pReceiveInd[i] = pReceiveInd[i-1]+pReceiveNum[i-1]-Size;
  }

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

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

// Function for the serial Gauss – Seidel method
void SerialResultCalculation(double *pMatrixCopy, int Size, double Eps,
  int &Iter){
  int i, j;  // Loop variables
  double dm, dmax,temp;
  Iter = 0;
  do {
    dmax = 0;
    for (i = 1; i < Size - 1; i++)
      for(j = 1; j < Size - 1; j++) {
```

```
        temp = pMatrixCopy[Size * i + j];
        pMatrixCopy[Size * i + j] = 0.25 * (pMatrixCopy[Size * i + j + 1] +
                                            pMatrixCopy[Size * i + j - 1] +
                                            pMatrixCopy[Size * (i + 1) + j] +
                                            pMatrixCopy[Size * (i - 1) + j]);
        dm = fabs(pMatrixCopy[Size * i + j] - temp);
        if (dmax < dm) dmax = dm;
      }
      Iter++;
  }
  while (dmax > Eps);

}
// Function to copy the initial data
void CopyData(double *pMatrix, int Size, double *pSerialMatrix) {
  copy(pMatrix, pMatrix + Size, pSerialMatrix);
}

// Function for testing the computation result
void TestResult(double* pMatrix, double* pSerialMatrix, int Size,
  double Eps) {
  int equal = 0; // =1, if the matrices are not equal
  int Iter;

  if (ProcRank == 0) {
    SerialResultCalculation(pSerialMatrix, Size, Eps, Iter);
    for (int i=0; i<Size*Size; i++) {
      if (fabs(pSerialMatrix[i]-pMatrix[i]) >= Eps)
        equal = 1;break;
    }
    if (equal == 1)
      printf("The results of serial and parallel algorithms"
             "are NOT identical. Check your code.");
    else
      printf("The results of serial and parallel algorithms"
             "are identical.");
  }
}

// Function for setting the grid node values by a random generator
void RandowmDataInitialization (double* pMatrix, int Size) {
  int i, j;  // Loop variables
  srand(unsigned(clock()));
  // Setting the grid node values
  for (i=0; i<Size; i++) {
    for (j=0; j<Size; j++)
      if ((i==0) || (i== Size-1) || (j==0) || (j==Size-1))
        pMatrix[i*Size+j] = 100;
      else
        pMatrix[i*Size+j] = rand()/double(1000);
  }
}

void main(int argc, char* argv[]) {
  double* pMatrix;       // Matrix of the grid nodes
  double* pProcRows;     // Stripe of the matrix on current process
  double* pSerialMatrix; // Result of the serial method
  int     Size;          // Matrix size
  int     RowNum;        // Number of rows in matrix stripe
  double  Eps;           // Required accuracy
  int     Iterations;    // Iteration number
  double currDelta, delta;
```

```
  setvbuf(stdout, 0, _IONBF, 0);
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
  MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);

  if(ProcRank == 0) {
    printf("Parallel Gauss - Seidel algorithm \n");
    fflush(stdout);
  }
  // Process initialization
  ProcessInitialization (pMatrix,pProcRows,Size,RowNum,Eps);

  // Creating the copy of the initial data
  if (ProcRank == 0) {
    pSerialMatrix = new double[Size*Size];
    CopyData(pMatrix, Size, pSerialMatrix);
  }

  // Data distribution among the processes
  DataDistribution(pMatrix, pProcRows, Size,RowNum);

  // Paralle Gauss-Seidel method
  ParallelResultCalculation(pProcRows, Size,RowNum,Eps, Iterations);
  //TestDistribution(pMatrix, pProcRows, Size,RowNum);

  // Gathering the calculation results
  ResultCollection(pProcRows, pMatrix, Size, RowNum);
  TestDistribution(pMatrix, pProcRows, Size,RowNum);

  // Printing  the result
  printf("\n Iter %d \n", Iterations);
  printf("\nResult matrix: \n");
  if (ProcRank==0) {
    //TestResult(pMatrix,Size,pMatrixCopy,Eps);
    PrintMatrix(pMatrix,Size,Size);
  }

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