Нижегородский государственный университет им. Н.И.Лобачевского

Факультет Вычислительной математики и кибернетики

Образовательный комплекс

Введение в методы параллельного программирования

Лабораторная работа 6.

Параллельные алгоритмы решения дифференциальных уравнений в частных производных

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

Содержание

Упражнения:

- Постановка задачи
- Реализация последовательного алгоритма Гаусса-Зейделя решения задачи Дирихле
- Разработка параллельного алгоритма Гаусса-Зейделя решения задачи Дирихле
- Реализация параллельного алгоритма Гаусса-Зейделя решения задачи Дирихле

Упражнение 1: Постановка задачи Дирихле

Проблема численного решения задачи Дирихле для уравнения Пуассона – 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)

В качестве области задания функции далее будет использоваться единичный квадрат:

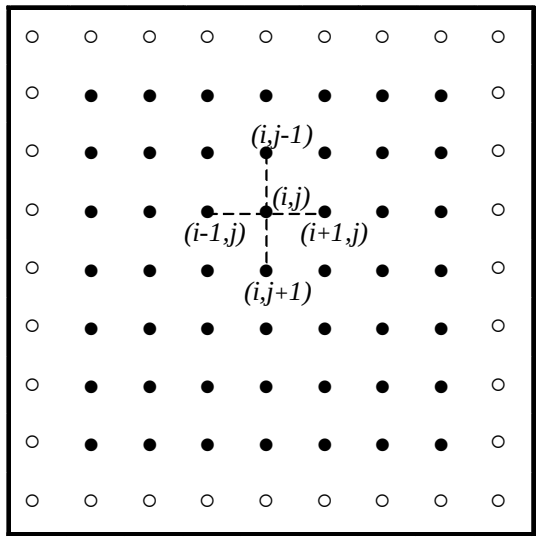
$$D = \{(x,y) \in R^2 : 0 \le x, y \le 1\}$$

Упражнение 1: Постановка задачи Дирихле

Метод конечных разностей является одним из наиболее распространенных подходов численного решения дифференциальных уравнений.

Используя этот метод, прямоугольная сетка в области может быть задана в виде:

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



Упражнение 1: Постановка задачи Дирихле

- В основе для построения различных *итерационных схем* решения задачи Дирихле используется представление уравнения Пуассона в конечно-разностной форме.
- *Метод Гаусса-Зейделя* для проведения итераций уточнения использует правило:

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

- 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.

Упражнение 1: Постановка задачи Дирихле

```
// 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 );
```

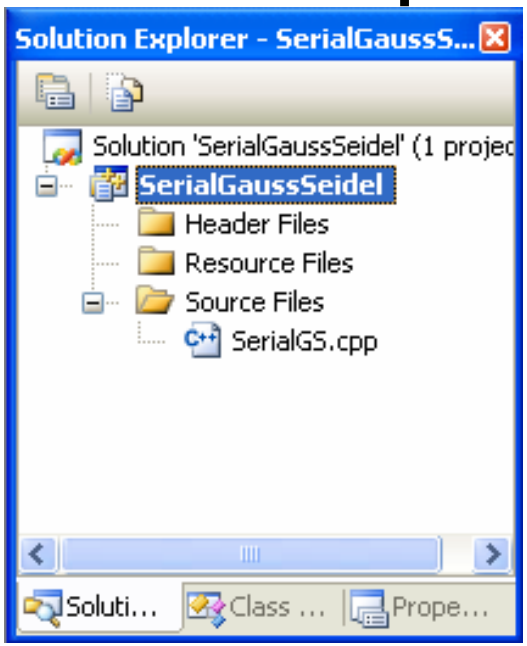
В дальнейшем при разработке программ для снижения сложности выполняемой лабораторной работы будем полагать, что функций *f* тождественно равно нулю, т.е. $f(x,y) \equiv 0$

Упражнение 2: Реализация последовательного алгоритма Гаусса-Зейделя решения задачи Дирихле

- **Поэтапная разработка последовательного алгоритма:**
  - Задание 1 – Открытие проекта **SerialGaussSeidel**
  - Задание 2 – Ввод исходных данных
  - Задание 3 – Задание начальных значений
  - Задание 4 – Завершение процесса вычислений
  - Задание 5 – Реализация алгоритма Гаусса-Зейделя
  - Задание 6 – Проведение вычислительных экспериментов

Упражнение 2: Реализация последовательного алгоритма Гаусса-Зейделя решения задачи Дирихле

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



Упражнение 2: Реализация последовательного алгоритма Гаусса-Зейделя решения задачи Дирихле

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

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

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

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

Упражнение 2: Реализация последовательного алгоритма Гаусса-Зейделя решения задачи Дирихле

- **Задание 2** – Ввод исходных данных:
  - Для задания исходных данных реализуем функцию *ProcessInitialization*:
    - определяет размеры сетки в области решения (размера матрицы pMatrix);
    - устанавливает требуемую точность Eps для решения задачи;
    - выделяет память для объекта *pMatrix;*
    - задания значений всех элементов матрицы *pMatrix*

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

Упражнение 2: Реализация последовательного алгоритма Гаусса-Зейделя решения задачи Дирихле

- **Задание 2** – Ввод исходных данных:
  - Для определения размера сетки в области поиска - необходимо задать значение переменной *Size*):

```
// 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);
}
```

Упражнение 2: Реализация последовательного алгоритма Гаусса-Зейделя решения задачи Дирихле

- **Задание 2** – Ввод исходных данных:
  - Выполним контроль правильности ввода данных.
  - Для этого добавим проверку размера сетки и, в случае ошибки (заданный размер меньше 3), продолжим запрашивать размер сетки до тех пор, пока не будет введено допустимое значение :

```
// 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);
```

Упражнение 2: Реализация последовательного алгоритма Гаусса-Зейделя решения задачи Дирихле

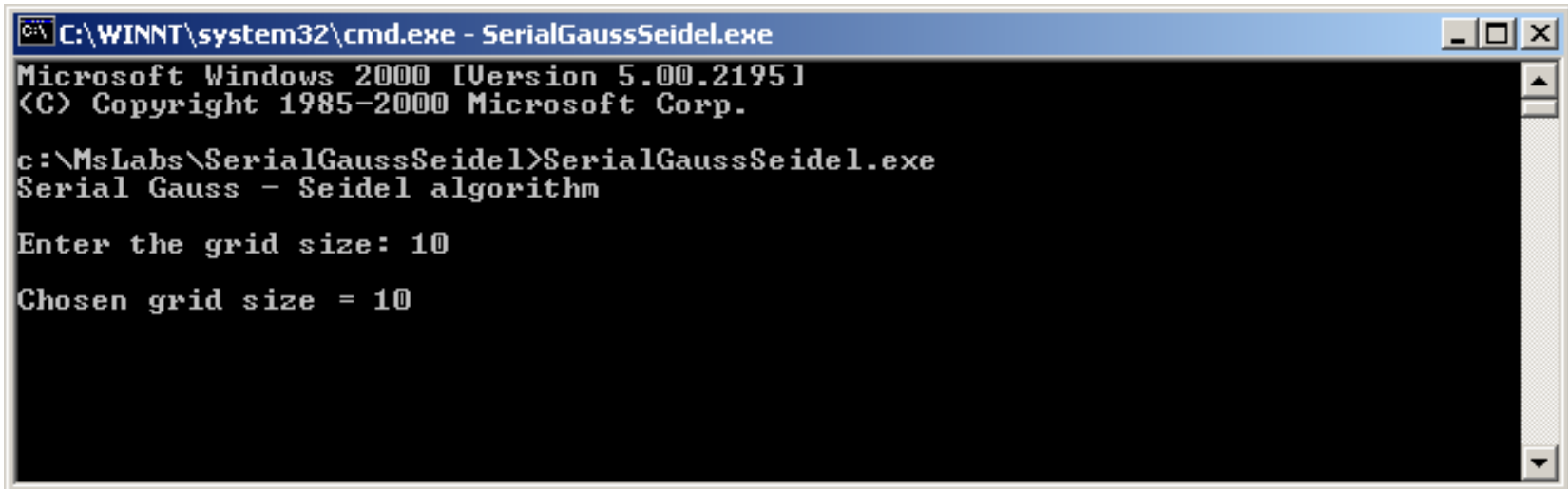
- **Задание 2** – Ввод исходных данных:
  - Добавьте вызов функции *ProcessInitialization* в основную функцию приложения:

```
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();
}
```

Упражнение 2: Реализация последовательного алгоритма Гаусса-Зейделя решения задачи Дирихле

- **Задание 2** – Ввод исходных данных:
  - Скомпилируйте и запустите приложение:



Упражнение 2: Реализация последовательного алгоритма Гаусса-Зейделя решения задачи Дирихле

- **Задание 3** – Задание начальных значений:
  - Выделение памяти для хранения данных:

```
// Function for memory allocation and initialization of grid
nodes
void ProcessInitialization (double* &pMatrix, int &Size) {
    <…>
  // Memory allocation
  pMatrix = new double [Size*Size];
}
```

Упражнение 2: Реализация последовательного алгоритма умножения матрицы на вектор…

- **Задание 3** – Задание начальных значений:
  - Определение значений элементов исходных матрицы по шаблону:
    - для всех внутренних узлов сетки устанавливается значение 0
    - для всех граничных узлов задается значение 100

$$pMatrix = \begin{pmatrix} 100 & 100 & 100 \\ 100 & 0 & 100 \\ 100 & 100 & 100 \end{pmatrix}$$

Упражнение 2: Реализация последовательного алгоритма Гаусса-Зейделя решения задачи Дирихле

- **Задание 3** – Задание начальных значений:
  - Реализуем функцию DummyDataInitialization для задания значений всех элементов матрицы pMatrix.

```
// 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;
  }
}
```

Упражнение 2: Реализация последовательного алгоритма Гаусса-Зейделя решения задачи Дирихле

- **Задание 3** – Задание начальных значений:
  - 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);
}
```

Упражнение 2: Реализация последовательного алгоритма Гаусса-Зейделя решения задачи Дирихле

- **Задание 3** – Задание начальных значений:
  - Для контроля правильности задания исходных данных необходимо разработать функцию.
  - *PrintMatrix* - функция форматированного вывода матрицы, в качестве аргументов передается указатель на одномерный массив и размеры матрицы по вертикали (количество строк *RowCount*) и горизонтали (количество столбцов *ColCount*).

Упражнение 2: Реализация последовательного алгоритма Гаусса-Зейделя решения задачи Дирихле

- **Задание 3** – Задание начальных значений:…
  - Форматированная печать матрицы:

```
// Function for formatted matrix output
void PrintMatrix (double* pMatrix, int RowCount,
  int ColCount) {
  int i, j; // Loop variables
  for (i=0; i<RowCount; i++) {
    for (j=0; j<ColCount; j++)
      printf("%7.4f ", pMatrix[i*ColCount+j]);
    printf("\n");
  }
}
```

Упражнение 2: Реализация последовательного алгоритма Гаусса-Зейделя решения задачи Дирихле

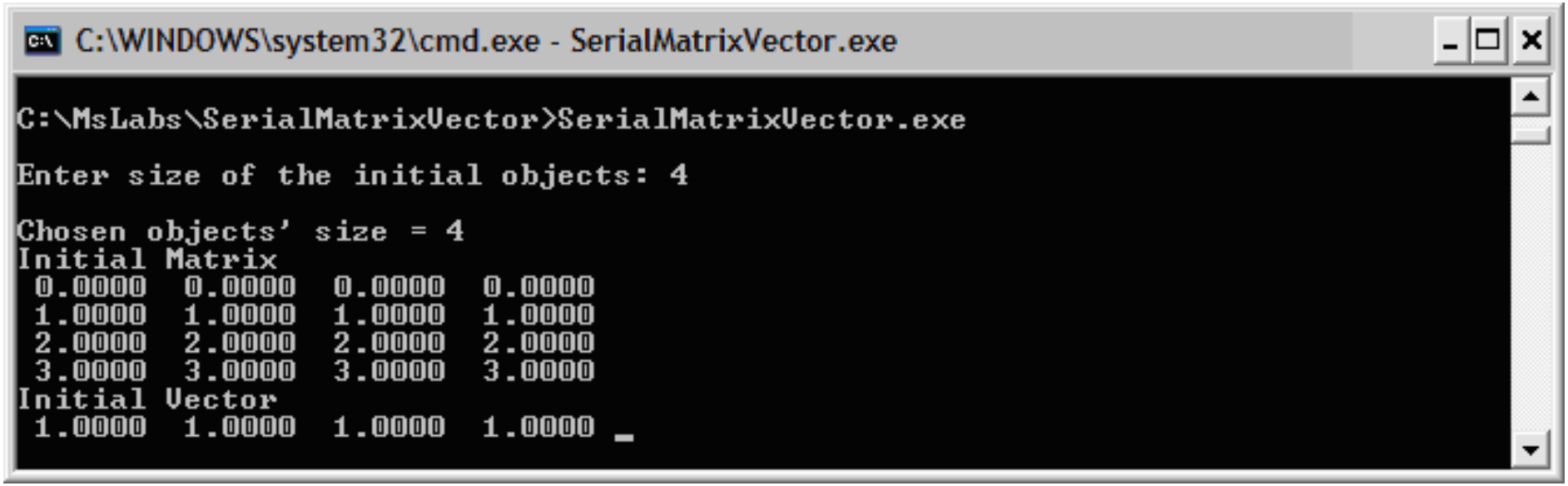
- **Задание 3** – Задание начальных значений:…
  - Контроль правильности выполнения этапа задания исходных данных:

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

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

Упражнение 2: Реализация последовательного алгоритма Гаусса-Зейделя решения задачи Дирихле

- **Задание 3** – Задание начальных значений:
  - Контроль правильности выполнения этапа задания исходных данных:



Упражнение 2: Реализация последовательного алгоритма Гаусса-Зейделя решения задачи Дирихле

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

Упражнение 2: Реализация последовательного алгоритма Гаусса-Зейделя решения задачи Дирихле

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

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

  - Вызов функции завершения вычислительного процесса:

```
// Matrix output
 <…>
// Computational process termination
ProcessTermination(pMatrix);
```

Упражнение 2: Реализация последовательного алгоритма Гаусса-Зейделя решения задачи Дирихле

- **Задание 5** – Реализация алгоритма Гаусса-Зейделя:
  - Для реализации алгоритма Гаусса-Зейделя для решения задачи Дирихле реализуем функцию *ResultCalculation*.
  - Как выходной параметр добавим переменную *Iterations*, в которой функция будет возвращать количество выполненных итераций алгоритма Гаусса-Зейделя до достижения требуемой точности.

```
// 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;
```

Упражнение 2: Реализация последовательного алгоритма Гаусса-Зейделя решения задачи Дирихле

```
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);
```

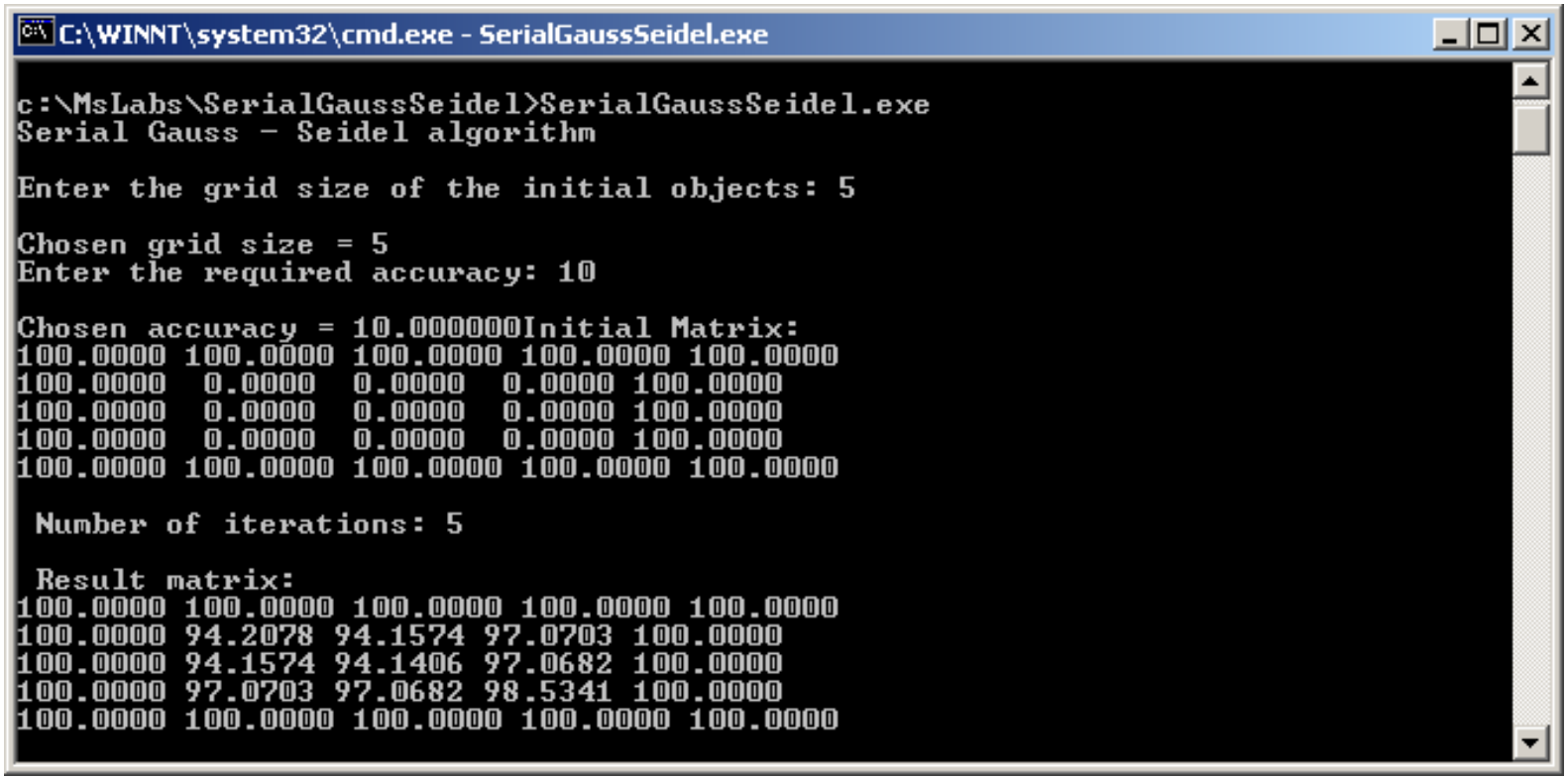
Упражнение 2: Реализация последовательного алгоритма Гаусса-Зейделя решения задачи Дирихле

- **Задание 5** – Реализация алгоритма Гаусса-Зейделя :
  - Добавьте вызов функции, реализующей алгоритм Гаусса-Зейделя в основную функцию программы,
  - Для контроля правильности работы алгоритма, распечатайте матрицу значений после выполнения алгоритма:

```
// Matrix and vector output
<…>
// 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);
```

Упражнение 2: Реализация последовательного алгоритма Гаусса-Зейделя решения задачи Дирихле

- **Задание 5** – Реализация алгоритма Гаусса-Зейделя:

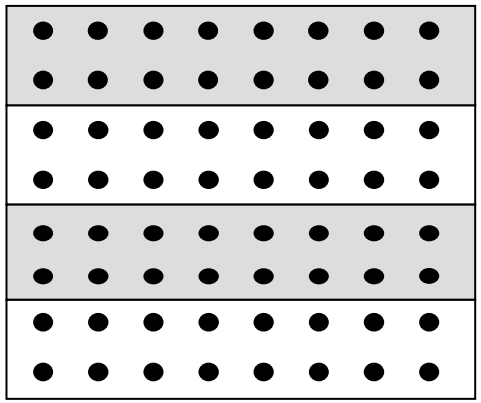


Упражнение 2: Реализация последовательного алгоритма Гаусса-Зейделя решения задачи Дирихле

- **Задание 6** – Проведение вычислительных экспериментов:
  - Для определения времени добавьте вызовы функций, позволяющие узнать время работы программы или ее части;
  - Проводите вычислительные эксперименты для достаточно больших размеров вычислительной сетки;
  - Измерьте времена работы алгоритма Гаусса-Зейделя;
  - Рассчитайте теоретическое время работы последовательного алгоритма;
  - Заполните таблицу результатов вычислений.

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

- При построении параллельных способов решения задачи Дирихле возможны два различных способа разделения данных:
  - одномерная или ленточная схема (разбиение матрицы по строкам)



  - двухмерное или блочное разбиение вычислительной сетки.

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

- **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:

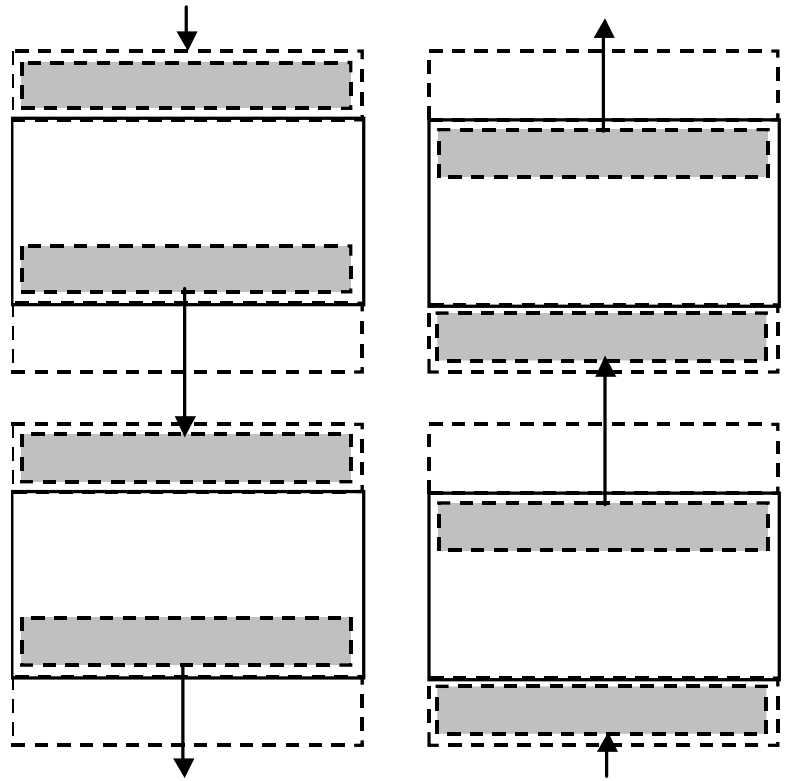
```
// Схема Гаусса-Зейделя, ленточное разделение данных
// действия, выполняемые на каждом процессоре
do {
  // <обмен граничных строк полос с соседями>
  // <обработка полосы>
  // <вычисление общей погрешности вычислений dmax>}
 while ( dmax > eps ); // eps - точность решения
```

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

- Процедура обмена граничных строк между соседними процессорами может быть разделена на две последовательные операции:

  - каждый процессор передает свою нижнюю граничную строку следующему процессору и принимает такую же строку от предыдущего процессора

  - процессоры передают свои верхние граничные строки своим предыдущим соседям и принимают переданные строки от следующих процессоров.



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

- 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:

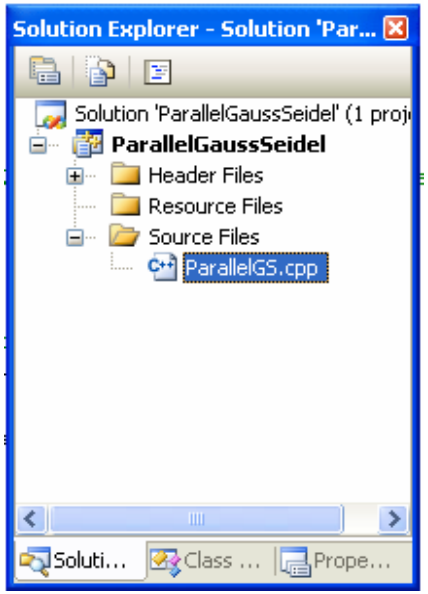
```
// Схема Гаусса-Зейделя, ленточное разделение данных
// действия, выполняемые на каждом процессоре
do {
  // обмен граничных строк полос с соседями
Sendrecv(u[M][*],N+2,NextProc,u[0][*],N+2,PrevProc);
Sendrecv(u[1][*],N+2,PrevProc,u[M+1][*],N+2,NextProc);
  // <обработка полосы с оценкой погрешности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

Упражнение 4: Реализация параллельного алгоритма Гаусса-Зейделя решения задачи Дирихле

- **Поэтапная разработка параллельного алгоритма:**
  - Задание 1 – Открытие проекта **ParallelGaussSeidel**
  - Задание 2 – Инициализация и завершение параллельной программы
  - Задание 3 – Определение количества процессов
  - Задание 4 – Ввод размера матрицы и точности для решения задачи
  - Задание 5 – Ввод исходных данных
  - Задание 6 – Завершение процесса вычислений
  - Задание 7 – Распределение данных между процессами
  - Задание 8 – Обмен граничных строк соседних процессов
  - Задание 9 – Реализация умножения матрицы на вектор
  - Задание 10 – Нахождение максимальной погрешности
  - Задание 11 – Сбор результатов вычислений
  - Задание 12 – Проверка правильности работы программы
  - Задание 13 – Реализация вычислений для любых размеров матрицы
  - Задание 14 – Проведение вычислительных экспериментов

Упражнение 4: Реализация параллельного алгоритма Гаусса-Зейделя решения задачи Дирихле

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



Упражнение 4: Реализация параллельного алгоритма Гаусса-Зейделя решения задачи Дирихле

- **Задание 1** – Открытие проекта **ParallelGaussSeidel** :
  - Проект содержит функции:
    - *DummyDataInitialization* – начальное задание исходных данных,
    - *ResultCalculation* – последовательный алгоритм Гаусса-Зейделя,
    - *PrintMatrix* – печать матрицы
  - В главной функции приложения объявлены переменные *pMatrix*, *Size, Eps*
  - Скомпилируйте и запустите приложение. Убедитесь, что на экран выводится сообщение

```
"Parallel Gauss-Seidel algorithm"
```

Упражнение 4: Реализация параллельного алгоритма Гаусса-Зейделя решения задачи Дирихле

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

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

Упражнение 4: Реализация параллельного алгоритма Гаусса-Зейделя решения задачи Дирихле

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

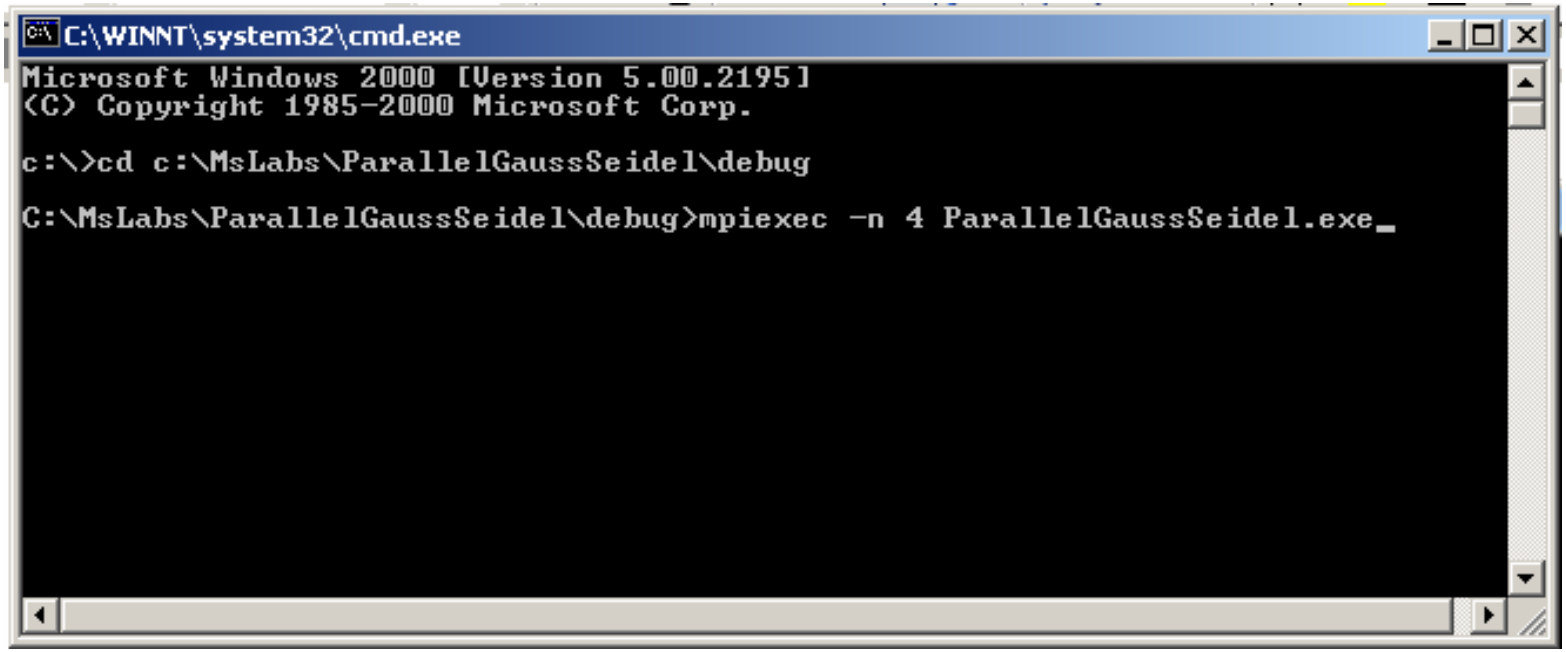
```
mpiexec –n <Num of processes> <Name of executive> <Arguments>
```

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

```
mpiexec –n 4 ParallelGaussSeidel.exe
```

Упражнение 4: Реализация параллельного алгоритма Гаусса-Зейделя решения задачи Дирихле

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



Упражнение 4: Реализация параллельного алгоритма Гаусса-Зейделя решения задачи Дирихле

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

```
int ProcNum;      // Number of available processes
int ProcRank;     // Rank of current process
void main(int argc, char* argv[]) {
  <…>
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
  MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
  printf ("Parallel Gauss-Seidel algorithm\n");
  MPI_Finalize();
}
```

Упражнение 4: Реализация параллельного алгоритма Гаусса-Зейделя решения задачи Дирихле

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

Упражнение 4: Реализация параллельного алгоритма Гаусса-Зейделя решения задачи Дирихле

- **Задание 4** – Ввод размера матрицы и точности для решения задачи:
  - Для ввода исходных данных, как и ранее, служит функция *ProcessInitialization*:

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

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

Упражнение 4: Реализация параллельного алгоритма Гаусса-Зейделя решения задачи Дирихле

- **Задание 4** – Ввод размера матрицы и точности для решения задачи:

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

Упражнение 4: Реализация параллельного алгоритма Гаусса-Зейделя решения задачи Дирихле

```
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);
}
}
```

Упражнение 4: Реализация параллельного алгоритма Гаусса-Зейделя решения задачи Дирихле

- **Задание 4** – Ввод размера матрицы и точности для решения задачи:
  - После того, как определены размер матрицы и точность для решения задачи, необходимо передать значение переменной Size и Eps на все процессы при помощи функции широковещательной рассылки MPI_Bcast:

```
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);
```

Упражнение 4: Реализация параллельного алгоритма Гаусса-Зейделя решения задачи Дирихле

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

Упражнение 4: Реализация параллельного алгоритма Гаусса-Зейделя решения задачи Дирихле

- **Задание 5** – Ввод исходных данных:
  - Согласно вычислительной схеме параллельного алгоритма, матрица разделяется между процессами на горизонтальные полосы: на каждом процессе содержится полоса матрицы *pProcRows*, содержащая *RowNum* строк,
  - 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;
 <…>
}
```

Упражнение 4: Реализация параллельного алгоритма Гаусса-Зейделя решения задачи Дирихле

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

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

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

Упражнение 4: Реализация параллельного алгоритма Гаусса-Зейделя решения задачи Дирихле

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

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

Упражнение 4: Реализация параллельного алгоритма Гаусса-Зейделя решения задачи Дирихле

- **Задание 7** – Распределение данных между процессами:…
  - Для распределения данных предназначена функция *DataDistribution*,
  - Матрица *pMatrix* разделяется между процессами при помощи функции обобщенной передачи данных от одного процесса всем процессам *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);
```

Упражнение 4: Реализация параллельного алгоритма Гаусса-Зейделя решения задачи Дирихле

```
  // 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);
}
```

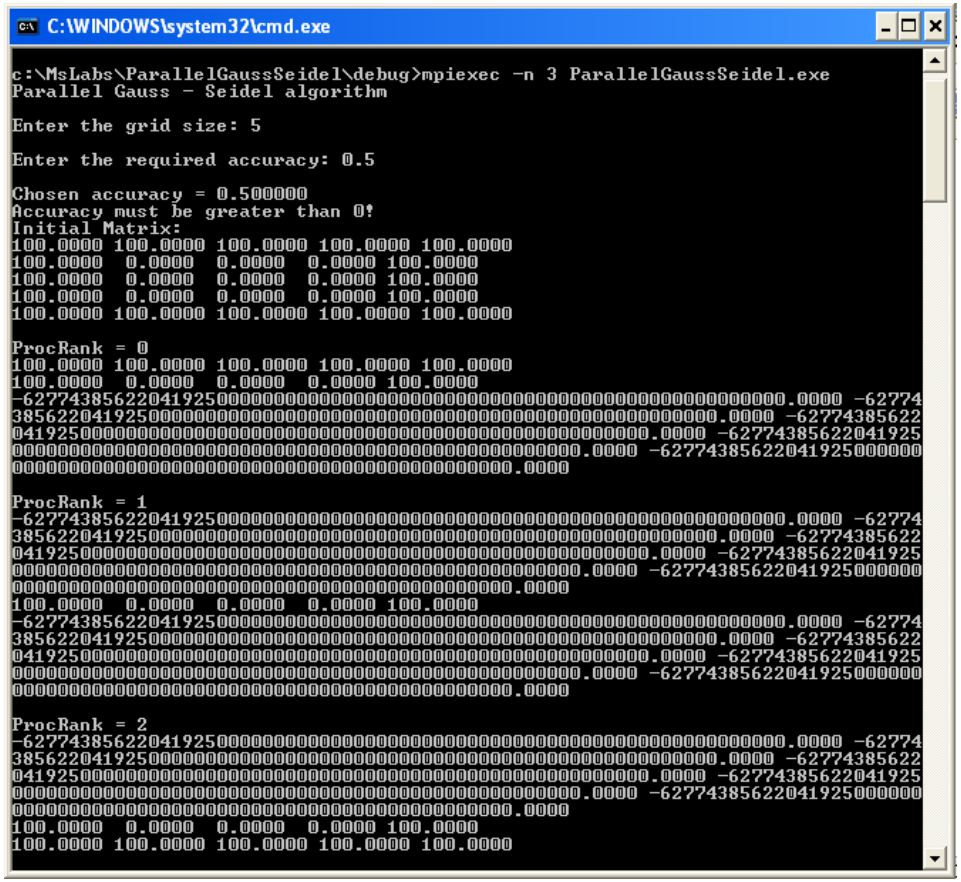
Упражнение 4: Реализация параллельного алгоритма Гаусса-Зейделя решения задачи Дирихле

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

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

Упражнение 4: Реализация параллельного алгоритма Гаусса-Зейделя решения задачи Дирихле

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



Упражнение 4: Реализация параллельного алгоритма Гаусса-Зейделя решения задачи Дирихле

- **Задание 8** – Обмен граничных строк соседних процессов:
  - Первое действие, которое должно выполняться на каждой итерации параллельного алгоритма Гаусса – Зейделя, состоит в обмене граничных строк между соседними процессами:

```
// 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);
}
```

Упражнение 4: Реализация параллельного алгоритма Гаусса-Зейделя решения задачи Дирихле

- **Задание 8** – Обмен граничных строк соседних процессов:
  - 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*:

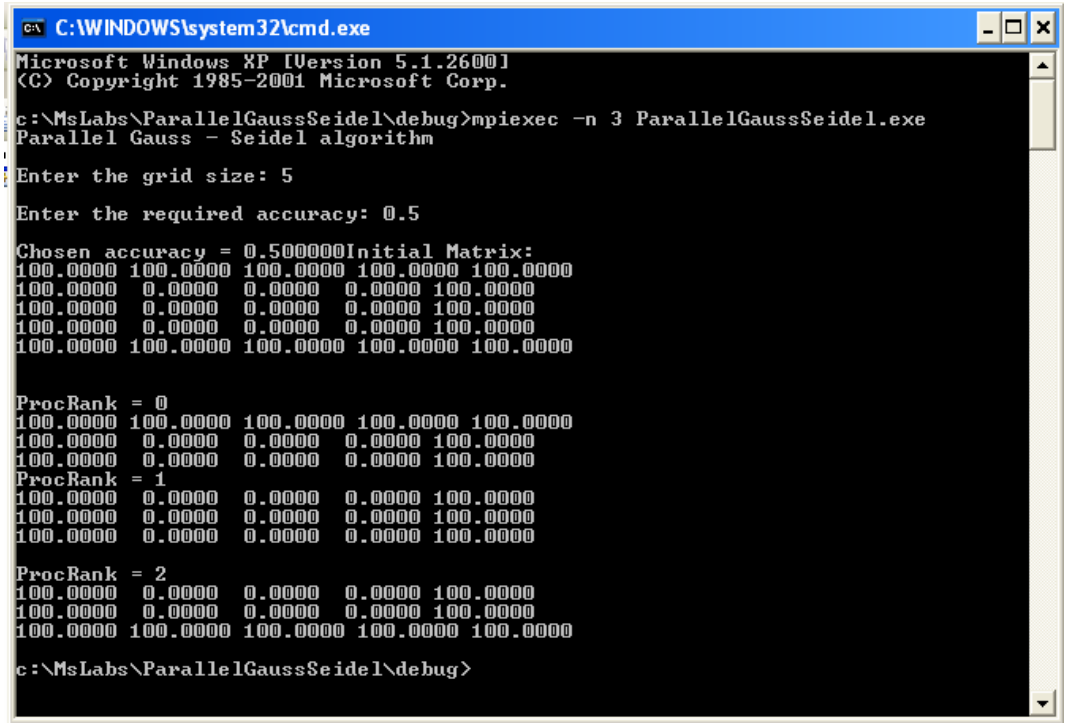
```
// Function for exchanging the boundary rows of the process
stripes
void ExchangeData(double* pProcRows, int Size, int RowNum) {
  MPI_Status status;
  int NextProcNum = (ProcRank == ProcNum-1)?MPI_PROC_NULL:
                                                ProcRank+1;
  int PrevProcNum = (ProcRank == 0)? MPI_PROC_NULL :  ProcRank-1;
```

Упражнение 4: Реализация параллельного алгоритма Гаусса-Зейделя решения задачи Дирихле

```
  // 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);
}
```

Упражнение 4: Реализация параллельного алгоритма Гаусса-Зейделя решения задачи Дирихле

- **Задание 8** – Обмен граничных строк соседних процессов:
  - Скомпилируйте приложение. Запустите приложение с использованием 3 параллельных процессов и установите размер сетки равный 6.
  - Убедитесь в том, что обмен граничных строк выполняется правильно



Упражнение 4: Реализация параллельного алгоритма Гаусса-Зейделя решения задачи Дирихле

- **Задание 9** – Выполнение итераций параллельного алгоритма:
  - Реализуем функцию *IterationCalculation* выполняющая итерации параллельного алгоритма Гаусса-Зейделя для

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

  - Для выполнения алгоритма Гаусса-Зейделя каждым процессом реализуем функцию *ParallelResultCalculation*

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

Упражнение 4: Реализация параллельного алгоритма Гаусса-Зейделя решения задачи Дирихле

- **Задание 9** – Выполнение итераций параллельного алгоритма:
  - Добавьте вызовы функции *IterationCalculation* и *ExchangeData* в функцию *ParallelResultCalculation*

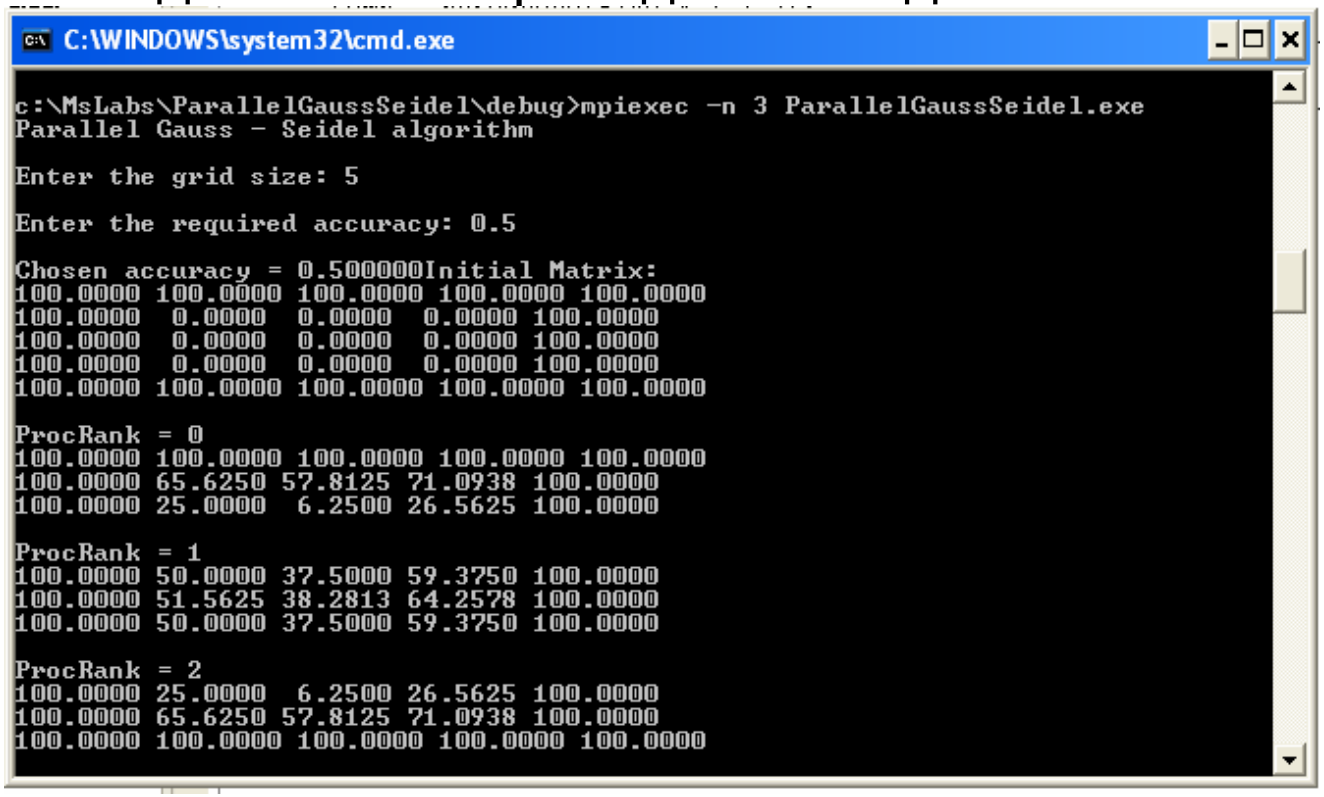
```
// Function for the parallel Gauss-Seidel method
void ParallelResultCalculation(double* pProcRows, int Size, int
RowNum,double Eps, int &Iterations) {
    <…>
//  do {
    Iterations++;
    // Exchanging the boundary rows of the process stripe
    ExchangeData(pProcRows, Size, RowNum);
    // The Gauss-Seidel method iteration
    ProcDelta = IterationCalculation(pProcRows, Size, RowNum);
    TestDistribution(pMatrix, pProcRows, Size, RowNum);
//  } while(Iteration < 2);
}
```

Упражнение 4: Реализация параллельного алгоритма Гаусса-Зейделя решения задачи Дирихле

- **Задание 9** – Выполнение итераций параллельного алгоритма:
  - Для контроля правильности выполнения алгоритма можно воспользоваться функцией *TestDistribution* и вызов следует переместить на строку после вызова вновь разработанной функции *IterationCalculation*
  - Добавьте вызов функцib *ParallelResultCalculation* в главную функцию параллельного приложения
  - Скомпилируйте и запустите приложение

Упражнение 4: Реализация параллельного алгоритма Гаусса-Зейделя решения задачи Дирихле

- **Задание 9** – Выполнений итерация параллельного алгоритма:
  - При запуске приложения с использованием трех процессов и размере сетки равном 5 результаты вычислений должны совпадать с нижеприведенными данными



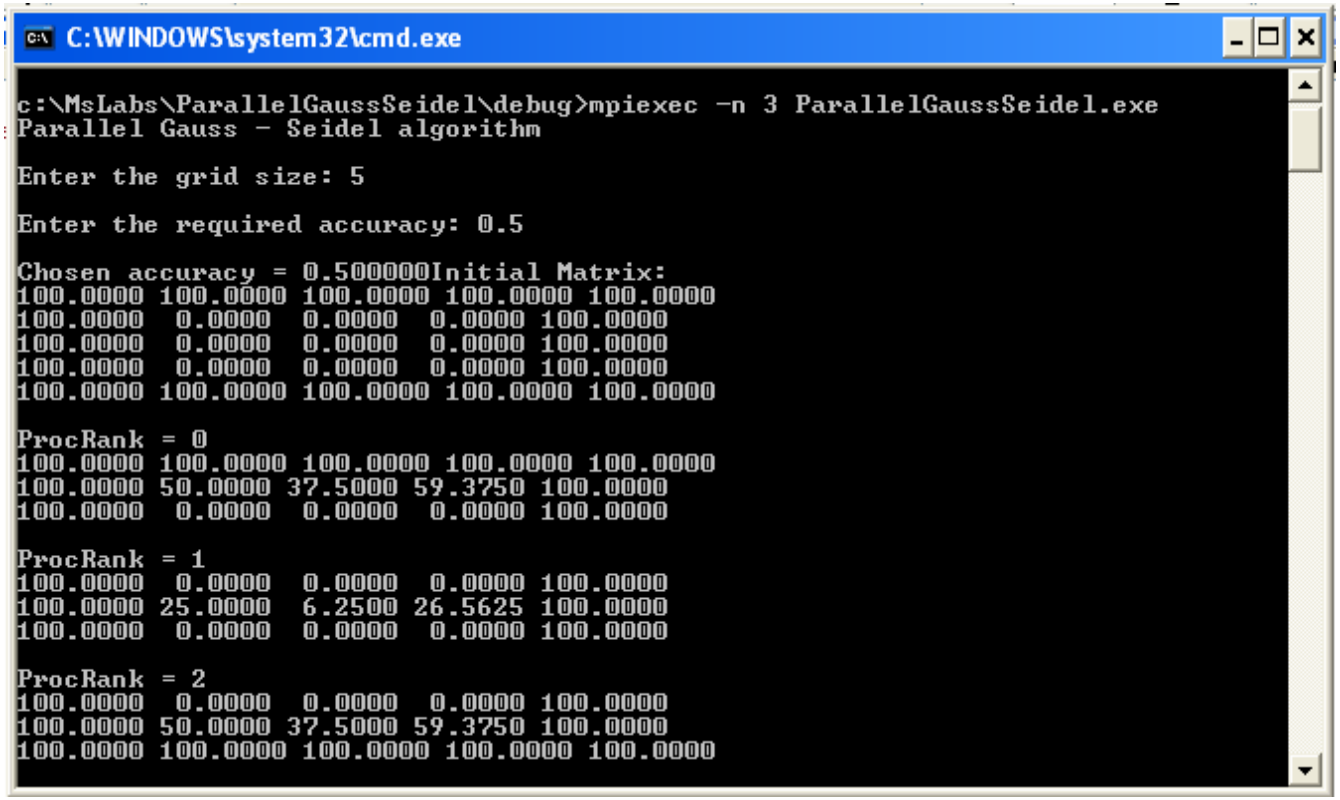
Упражнение 4: Реализация параллельного алгоритма Гаусса-Зейделя решения задачи Дирихле

- **Задание 10** – Нахождение максимальной погрешности:
  - Добавьте вычисление максимального изменения результатов расчетов, получаемых на итерации алгоритма.
  - Необходимые изменения функции *ParallelResultCalculation* состоят в следующем :

```
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 );
}
```

Упражнение 4: Реализация параллельного алгоритма Гаусса-Зейделя решения задачи Дирихле

- **Задание 10** – Нахождение максимальной погрешности:
  - При запуске приложения с использованием трех процессов и размере сетки равном 5 и требуемой точности 0.1 результаты вычислений должны совпадать с нижеприведенными данными



Упражнение 4: Реализация параллельного алгоритма Гаусса-Зейделя решения задачи Дирихле

- **Задание 11** – Сбор результатов вычислений:
  - На ведущем процессе необходимо собрать полосы сетки, расположенных на разных процессах, для это используем функцию *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);
}
```

  - Скомпилируйте и запустите приложение. Оцените правильность его работы.

Упражнение 4: Реализация параллельного алгоритма Гаусса-Зейделя решения задачи Дирихле

- **Задание 12** – Проверка правильности работы программы:
  - Для проверки данных необходимо, чтобы последовательный и параллельный алгоритм использовали одни и те же данные.
  - Реализуем функцию *CopyData* выполняющая копирование исходных данных

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

Упражнение 4: Реализация параллельного алгоритма Гаусса-Зейделя решения задачи Дирихле

- **Задание 12** – Проверка правильности работы программы:
  - Функция *TestResult* сравнивает результаты работы последовательного и параллельного алгоритмов путем поэлементного сравнения полученных результатов:

```
// Function for testing the computation result
void TestResult(double* pMatrix, double* pSerialMatrix,
                int Size, double Eps)
```

  - Результатом работы этой функции является печать диагностического сообщения. Используя эту функцию, можно проверять результат работы параллельного алгоритма независимо от того, насколько велики исходные объекты при любых значениях исходных данных.

Упражнение 4: Реализация параллельного алгоритма Гаусса-Зейделя решения задачи Дирихле

- **Задание 12** – Проверка правильности работы программы:
  - Закомментируйте использовавшуюся ранее отладочную печать,
  - Реализуйте функцию *TestResult*,
  - Добавьте вызов функции *TestResult* в функцию *main*,
  - Замените вызов функции *DummyDataInitialization* на вызов функции *RandomDataInitialization* в функции *ProcessInitialization*
  - Протестируйте работоспособность приложения

Упражнение 4: Реализация параллельного алгоритма Гаусса-Зейделя решения задачи Дирихле

- **Задание 13** – Реализация вычислений для любых размеров матрицы:
  - Внесите необходимые изменения в функции *ProcessInitialization*, *DataDistribution, ResultCollection*

Упражнение 4: Реализация параллельного алгоритма Гаусса-Зейделя решения задачи Дирихле

- **Задание 14** – Проведение вычислительных экспериментов:
  - Добавьте вычисление и вывод времени выполнения алгоритма Гаусса-Зейделя,
  - Протестируйте работоспособность приложения,
  - Проведите вычислительные эксперименты,
  - Измерьте времена работы алгоритма Гаусса-Зейделя при различных количествах исходных данных и различном числе процессов,
  - Определите получаемое ускорение,
  - Вычислите теоретическое время работы параллельного алгоритма,
  - Заполните таблицу результатов вычислений

Заключение

- Рассмотрен один из методов параллельного выполнения алгоритма Гаусса-Зейделя
- Разработаны приложения, реализующие последовательный и параллельный алгоритмы Гаусса-Зейделя
- Проведены вычислительные эксперименты, проведено сравнение последовательного и параллельного алгоритмов

Темы заданий для самостоятельной работы

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

Литература

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

Авторский коллектив

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

Гришагин В.А., доцент, к.ф.м.н.

Абросимова О.Н., ассистент (раздел 10)

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

Лабутин Д.Ю., ассистент (система ПараЛаб)

Сысоев А.В., ассистент (раздел 1)

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

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

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

Ливерко С.В. (система ПараЛаб)

О проекте

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

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

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